LEHRSTUHL FÜR FLUGANTRIEBE
TECHNISCHE UNIVERSITÄT MÜNCHEN
UNIV.-PROF. DR.-ING. HANS-PETER KAU

DIPLOMARBEIT

Implementation of Rotor Stator Interfaces for the Open Field Operation and Manipulation (OpenFOAM) CFD Toolbox

Verfasser: Franz Blaim
Matrikelnummer: 2520070

Betreuer: Dipl.-Ing. Oliver Borm
Dipl.-Ing. Tobias Fröbel

Ausgabe: 01.04.2008
Abgabe: 30.09.2008

Contents

List of Figures

List of Tables

Table of Symbols

The following symbols have been used. Throughout the document all italic printed words denote a C++ function, a class name or an application name.

Indices

1	denoting first cascade
2	denoting second cascade
360	regarding the whole circular ring
η	property related to the direction formed between two interface vertexes
ω	denoting a variable resulting from the angular frequency
Φ	denoting a term related to the state variable
$\rightarrow$	denoting the observation direction. The state is observed from the coordinate frame at the left and viewed in the coordinate frame at the right of the arrow.
φ	azimuthal vector component in case of a constant integer number it denotes the number of sector elements
ξ	property related to the direction formed between a nodal point and its neighbour
A	denoting a property of the boundary named A
$a-b$	azimuthal boundary of a passage
AP	denoting a value between the node P and the boundary A
BC	property denoting a boundary condition value. Or denoting a distance between a boundary face and the nodal point P.
$c-d$	azimuthal boundary of a passage
CCS	cylindrical coordinate system
cloned	cloned values
E	denoting property at the east nodal point
e	denoting the east face
$e-f$	azimuthal boundary of a passage
eE	denoting property between the east face and the east nodal point
EP	denoting distance between the nodal point P and its east neighbour
ex	denoting an extensive property
F	denoting the front running boundary
f	denoting a property at a face
ggi	denoting a general grid interface weighting factor

$h-g$	azimuthal boundary of a passage
i	denoting the node number
in	denoting an intensive property
inf	denoting a internal field component
M	variable related to the master patch
m	integer variable
map	directly mapped values onto the interpolation disk
n	denoting a neigbour cell property
P	denoting a property belonging to the node P
PE	denoting property between the east nodal point and the nodal point P
Pe	denoting property between the east face and the nodal point P
pl	phase lagged
r	radial vector component
rot	rotated property
S	variable related to the slave patch
s	denoting a term related to the shadow patch
$surf$	denoting a field evaluated at a surface
T	denoting the trailing running boundary
t	time
u	denoting a uniform variable
W	denoting the west node
w	denoting the west face
WE	denoting a property between the west and east nodal point
WP	denoting distance between the nodal point P and its west neighbour
wP	denoting property between the west face and the nodal point P
Ww	denoting property between the west nodal point and the west face
x	cartesian vector component in y direction
xyz	denoting a position or vector noted in cartesian rectangular coordinates
y	cartesian vector component in x direction
constants	
k	constant factor
K	constant integer factor to adjust the time-step for the phase-lag
N	constant integer number
Parameter	
i	integer run variable
$\bar{\omega}$	circumferential speed magnitude
$\Delta\varphi$	opening angle of the patch sector in degrees
ΔV	control volume around a node

Δ	geometric distance between nodes or face centers
δ	geometry dependent delta coefficient or a direction vector between two consecutive nodal points
η	direction formed between two interface vertexes
Γ	diffusion coefficient
Φ	flux variable
ρ	density of a fluid
$\mathscr{A}$	denoting the coefficient matrix
$\mathscr{g}$	denoting the covariant metric tensor
$\mathscr{I}$	identity tensor
$\mathscr{R}$	rotatory tensor
$\mathscr{T}$	stress tensor
$\vec{\omega}$	axis vector for the rotation
θ	azimuthal angle
ξ	direction formed between a nodal point and its neighbour
A	area
a	matrix coefficient
CV	control volume
cv	cross flux correction
E	denoting the east node
e	base vector component
F	mass flux through area
f	weighting factor
n	normal
R	radius value
S	source term
V	volume
w	without overhead weighting factor, with vector overhead denoting the relative velocity

Overheads

$'$	denoting a variable in the relative frame
$-$	averaged value
$\rightarrow$	denoting a vector
r	rank of the variable

Chapter 1

Introduction

The Institute for Flight Propulsion at the Technische Universität München uses numerical calculation methods especially suited for the turbo-machinery flow-field calculation. One focus lies on the numerical simulation of rotor stator interaction. These interactions are observed in compressor stages and turbine stages and have a major influence on the engine characteristic map. They also influence the noise generation of the whole jet engine in a significant way. In *fig. 1.1* this interaction is shown, the main source for the interaction between rotor and stator are the wakes produced by the blades. By calculating a stage flow field the governing equations are solved in the relative frame of reference. For the case of a rotating frame and a non rotating or counter rotating reference frame, those equations must be coupled in a conservative way.

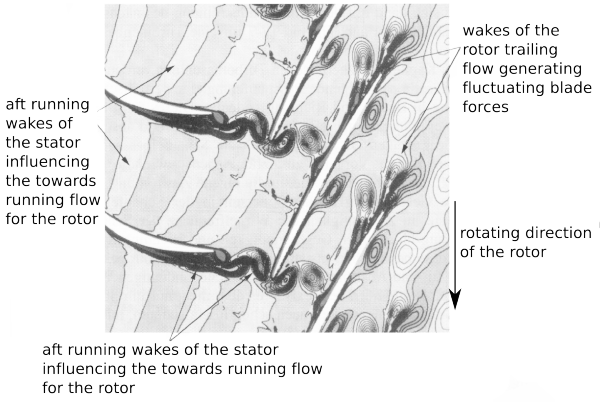


Figure 1.1: according to [4, p.1031] visualized rotor stator interaction

One approach is to use a full 360 degree stage simulation, which leads for a multistage case to an unacceptable increase of the computational effort caused by an increase of grid cells. To minimize the computational effort only a part of the whole blade number is used by commercial computation packages, which utilize several rotor-stator interfaces for the coupling between the different reference frames. Those are able to ensure the spatial and temporal cyclicity of the flow-field. Those rotor-stator interfaces can be divided into interfaces suited for steady and those suited for unsteady flow problems.

For steady simulations the frozen-rotor and the mixing-plane approach are widely used. In case of unsteady calculations the domain-scaling , phase-lag and non-linear-harmonics [25] methods are used.

1.1 Open-source CFD

Besides many commercial packages the open-source community provides also several codes to solve computational fluid dynamics. But until now, those cannot provide utilities to simulate rotor-stator interactions in a detailed and more important efficient way, because they are lacking the rotor-stator interfaces. Right now the most advanced and widely accepted CFD toolbox is the OpenFOAM CFD toolbox library. OpenFOAM has the big advantage that it originates from a commercial CFD code, which was one of the first written completely with a object orientated approach. This enables users to define new equations in an easy and elegant way. Therefore the user is not restricted to the existing solvers and boundary conditions, but has enough freedom to specify theoretically almost every kind of solution algorithm or boundary condition. Because of the object orientated approach and the usage of the C++ programing language it is very well suited for the scientific usage. It is also possible to use several commercial mesh formats by using the conversion tools included in the OpenFOAM toolbox, which makes it a versatile solving library.

A new general grid interface interpolation algorithm called *GGIinterpolation* has been introduced with the OpenFOAM 1.3 development version. This algorithm is claimed to be a conservative interpolation and enables the coupling of two planes (patches) belonging to different domains with two different mesh structures. Therefore the basic technology for developing rotor stator interfaces was founded. At the second OpenFOAM workshop in Zagrep 2007 the task for the OpenFOAM turbo-machinery group was set to develop a mixing-plane and a frozen rotor interface by the use of the *GGIinterpolation* algorithm see [20, pp.15]. Therefore the Institute for Flight Propulsion decided to participate by developing rotor-stator interface as a diploma-thesis. The used version is the OpenFOAM 1.4.1 development version which is available over svn at the sourceforge website. This version can be used for Linux, Unix operating systems but there is also a port available which makes it to run under cygwin.

1.2 Short work overview

The first step was to find out, if the development of rotor-stator interfaces has given a success at the time of April 2008. The first step was to get an overview about the development status. This showed that no rotor-stator interfaces were developed at that time. The next step was to find out, how the OpenFOAM toolbox can be used to implement boundary conditions and this knowledge should make it possible to implement rotor-stator interfaces. This is explained in chapter *chap. 2*. Because the OpenFOAM toolbox consists out of 662,231 lines of real source code, measured by the sloccount program, see [31], this was a very work intensive part. After this important

research has been successfully completed, the next step was to set up a steady state rotor stator interface. For this task the mixing-plane approach seemed to be well suited. The theory of the mixing-plane and the implementation is explained in *chap. 3*. After that, the necessary knowledge and experience for setting up a more complicated unsteady interface has been made. For that task the domain-scaling interface was implemented, see *chap. 4*. The last interface for which an implementation is needed is the phase-lag interface, which enables the user to handle rotor-stator setups having arbitrary pitch ratios. The theory and an idea for the implementation is explained in *chap. 5*.

Chapter 2

Solving Process in OpenFOAM

Because it is the task of this thesis to extend the OpenFOAM library with sophisticated boundary conditions like the mixing-plane, domain-scaling and phase-lag approach it is essential to understand how the theoretical computational fluid dynamics (CFD) solving process is working and in special, how the boundary conditions are treated in OpenFOAM. Therefore the first part of this chapter delivers a simple overview over an example problem. The gained insight through that, is used to explain in the second part of this chapter, how the OpenFOAM boundary conditions can be implemented and which part of the theory must be treated in special.

2.1 Theoretical solution of CFD problems

The general equations for a solving process for CFD problems are explained using an one dimensional example of a diffusion problem applied to a simple rod with two ends, see *fig. 2.1.* The governing equation for steady diffusion is expressed in *eq. (2.1)*, with Γ the diffusion coefficient, Φ the flux variable and S_Φ the source term. After integrating it over a control volume (CV) and applying the Gauss expression for vector integrals, it can be written as *eq. (2.2)*, see [30, pp.116].

$$\nabla \bullet (\Gamma \bullet \nabla \Phi) + S_\Phi = 0 \tag{2.1}$$

$$\int_{CV} div(\Gamma grad\Phi) dV + \int_{CV} S_\Phi dV = \int_A \overrightarrow{n} \bullet (\Gamma grad\Phi) dA + \int_{CV} S_\Phi dV = 0 \tag{2.2}$$

Because this example is dealing only with 1D *eq. (2.1)* can be simplified to *eq. (2.3).*

$$\frac{d}{dx}\left(\Gamma \frac{d\Phi}{dx}\right) + S = 0 \tag{2.3}$$

The first step in the finite volume method is the generation of discrete control volumes, which are used for the volume integration. For the example a simple 1D rod is used,

with constant values applied at the boundaries. In *fig. 2.1* the rod is divided into several equal spaced parts with faces positioned midway between consecutive points. Therefore each node is surrounded with a control volume forming a cell. Near the edge of the domain the control volumes are set up in a way that their faces align with the physical boundaries A and B of the computational domain.

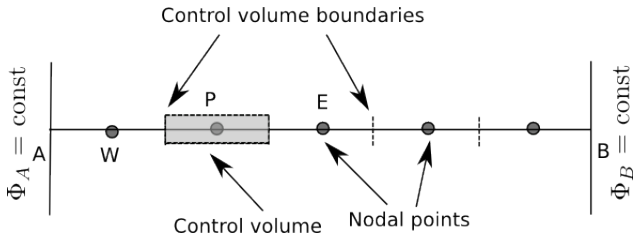


Figure 2.1: Example control volume discretisation for a 1D rod with boundaries A and B according to [30, pp.116]

A general node P is identified by its neighbors called W for west and E for east. The west side face of point P is named w and the east side face for P is named e, see *fig. 2.2*. The distance from the nodal point P to it's west neighbour node is denoted with Δ_{WP} and for the east neighbour node with Δ_{EP}.

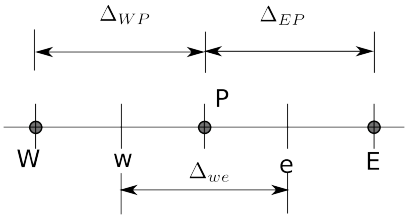


Figure 2.2: According to [30, pp.116] the distances between nodes and cell faces are shown.

After the grid has been established, the governing state formula has to be discretized. Therefore *eq. (2.3)* is integrated over an arbitrary control volume to derive the *eq. (2.4)* for every nodal point. The cross-sectional area of the control volume face is denoted with A. $\bar{S}$ is the average source term over the control volume ΔV of a cell around a node.

$$\int_{\Delta V} \frac{d}{dx}\left(\Gamma \frac{d\Phi}{dx}\right) dV + \int_{\Delta V} S dV = \left(\Gamma A \frac{d\Phi}{dx}\right)_e - \left(\Gamma A \frac{d\Phi}{dx}\right)_w + \bar{S}\Delta V = 0 \tag{2.4}$$

In order to derive an useful form for *eq. (2.4)* the diffusion coefficients at the west face Γ_w and at the east face Γ_e are required. Also the gradients $\frac{d\Phi}{dx}$ at the east and west faces are necessary. It seems to be appropriate to use linear approximations between the adjacent nodes to calculate Γ_w *eq. (2.5)* and Γ_e *eq. (2.6)*, see [30, p.117].

$$\Gamma_w = \frac{\Gamma_W + \Gamma_P}{2} \tag{2.5}$$

$$\Gamma_e = \frac{\Gamma_P + \Gamma_E}{2} \tag{2.6}$$

It has to be noted, that *eq. (2.6)* and *eq. (2.5)* are only valid for a regular grid, in case of an unstructured grid Γ_w and Γ_e are calculated by *eq. (2.7)* and *eq. (2.9)* according to [30, 21, pp.448, pp.44].

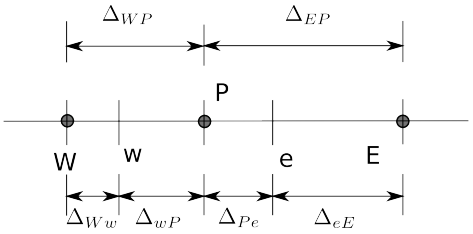


Figure 2.3: In case of an unstructured grid the distances between consecutive nodes are not uniform, which makes a linear weighting for the face values necessary.

The distances used in *eq. (2.7)* are visualized in *fig. 2.3*. The node which has a smaller distance contributes more to the interface value than the node with the greater distance. Therefore an inverse relation exists between distance and weighting factor. To account for that, the distance which is located next to the neighbour is used to calculate the weighting factor.

$$\Gamma_w = (1 - f_W)\Gamma_W + f_W\Gamma_P \tag{2.7}$$

The f_W is the weighting factor for the west fraction contributed by the point P. It can be calculated by dividing the distance from the west centroid to the west face Δ_{Ww} through the sum of the distance from the west face to the centroid of P Δ_{wP}. This is the same as the distance from the west centroid to the P centroid Δ_{WP}, see *eq. (2.8)*.

$$f_W = \frac{\Delta_{Ww}}{\Delta_{Ww} + \Delta_{wP}} = \frac{\Delta_{Ww}}{\Delta_{WP}} \tag{2.8}$$

The same is true for the east interface value Γ_e, see *eq. (2.9)*. In that case the weighting factor for the East centroid value is calculated with *eq. (2.10)*.

$$\Gamma_e = (1 - f_P)\Gamma_P + f_P\Gamma_E \tag{2.9}$$

The Δ_{Pe} in *eq. (2.10)* is the distance from the P centroid to the east face centre. The Δ_{eE} is the distance from the East centroid to the east face center. Again the sum in the divisor of *fig. 2.10* can be replaced with the distance from the centroid P to the centroid E symbolized with Δ_{PE}.

$$f_P = \frac{\Delta_{Pe}}{\Delta_{Pe} + \Delta_{eE}} = \frac{\Delta_{Pe}}{\Delta_{PE}} \tag{2.10}$$

The diffusive flux terms can be evaluated for the east face with *eq. (2.11)*, according to [30, p.117].

$$\left(\Gamma A \frac{d\Phi}{dx}\right)_e = \Gamma_e A_e \frac{1}{\Delta_{PE}} \left(\Phi_E - \Phi_P\right) \tag{2.11}$$

The same approach is used for the west face in *eq. (2.12)*.

$$\left(\Gamma A \frac{d\Phi}{dx}\right)_w = \Gamma_w A_w \frac{1}{\Delta_{WP}} \left(\Phi_P - \Phi_W\right) \tag{2.12}$$

In practical cases the source term S can be a function of the dependent variable, therefore the source term is approximated by the use of a linear form *eq. (2.13)*. The S_u part is the uniform part of the source term which is not dependent on the state variable Φ_P.

$$\bar{S}\Delta V = S_u + S_p \Phi_P \tag{2.13}$$

Substituting equations *eq. (2.11)*, *eq. (2.12)* and *eq. (2.13)* into *eq. (2.4)*, gives *eq. (2.14)*. In *eq. (2.14)* the δ symbol replaces the division of one through a geometric distance and is called in the following the delta coefficient.

$$\Gamma_e A_e \delta_{PE} \left(\Phi_E - \Phi_P\right) - \Gamma_w A_w \delta_{WP} \left(\Phi_P - \Phi_W\right) + \left(S_u + S_P \Phi_P\right) = 0 \tag{2.14}$$

The *eq. (2.14)* can be rewritten as *eq. (2.15)*.

$$\underbrace{\left(\delta_{PE}\Gamma_e A_e + \delta_{WP}\Gamma_w A_w - S_P\right)}_{a_P} \Phi_P = \underbrace{\left(\delta_{WP}\Gamma_w A_w\right)}_{a_W} \Phi_W + \underbrace{\left(\delta_{PE}\Gamma_e A_e\right)}_{a_E} \Phi_E + S_u \tag{2.15}$$

Now the coefficients of Φ_W, Φ_E and Φ_P in *eq. (2.15)* can be substituted with a_w, a_E and a_P, which leads to *eq. (2.16)*.

$$a_P \Phi_P = a_W \Phi_W + a_E \Phi_E + S_u \tag{2.16}$$

After rearranging *eq. (2.16)* to separate between source terms and flux terms it becomes to *eq. (2.17)*.

$$a_P \Phi_P - a_W \Phi_W - a_E \Phi_E = S_u \tag{2.17}$$

The coefficients are written in *tab. 2.1*. The coefficients in *tab. 2.1* are valid in every nodal point, except for the boundary nodes. For those *eq. (2.17)* has to be modified to incorporate the boundary conditions.

As mentioned before the boundary conditions must be integrated into the coefficients. The boundary at A has a fixed value Φ_A but the most important change is the distance

a_W	a_E	a_P
$\delta_{WP}\Gamma_w A_w$	$\delta_{PE}\Gamma_e A_e$	$a_W + a_E - S_P$

Table 2.1: Showing the derived discretisation coefficients for the east, west and the point P itself.

variation incorporated in δ_{AP}, see *fig. 2.4*. The Δ_{AP} is in case of an equally spaced grid equal to half of the distance between two consecutive nodes.

Because the boundary condition does not represent a node within the grid, but a face, it is incorporated in the matrix as a source term. Having *fig. 2.4* in mind the discretised equation at the boundary A can be written as *eq. (2.18)*.

$$\Gamma_E \left(\frac{\Phi_E - \Phi_P}{\Delta_{EP}} \right) - \Gamma_A \left(\frac{\Phi_P - \Phi_A}{\Delta_{AP}} \right) \tag{2.18}$$

This can be transformed into a form, which corresponds to the coefficient notation, see *eq. (2.19)*.

$$\Gamma_E \delta_{EP}(\Phi_E - \Phi_P) - (\Gamma_A \delta_{AP}(\Phi_P - \Phi_A)) + (S_u + S_P \Phi_P) = 0 \tag{2.19}$$

After sorting *eq. (2.19)* for the nodes, it becomes to *eq. (2.20)*.

$$\underbrace{(\Gamma_A \delta_{AP} + \Gamma_E \delta_{EP})}_{a_P} \Phi_P = \underbrace{0\Gamma_W}_{a_W} + \underbrace{\Gamma_E \delta_{EP}}_{a_E} \underbrace{\Phi_E + \Gamma_A \delta_{AP} \Phi_A}_{S_u} \tag{2.20}$$

Those coefficients are shown again in a structured way in *tab. 2.2*.

a_w	a_E	a_P	S_P	S_u
0	$\Gamma_E \delta_{EP} A$	$a_W + a_E - S_P$	$-\Gamma_A \delta_{AP} A$	$\Gamma_A \delta_{AP} A \Phi_A$

Table 2.2: Coefficients modified accounting for a fixed boundary condition.

For a system of m nodes the whole process will lead to a $m \times m$ matrix problem, shown in *eq. (2.21)*.

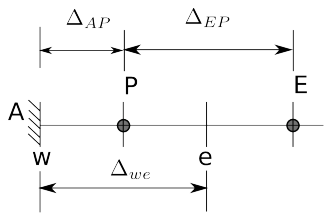


Figure 2.4: Boundary conditions change the distances used to calculate the delta coefficients.

$$\underbrace{\begin{bmatrix} a_{p,1} & a_{E,1} & 0 & 0 & 0 \\ a_{W,2} & a_{P,2} & a_{E,2} & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & a_{W,m-1} & a_{P,m-1} & a_{E,m-1} \\ 0 & 0 & 0 & a_{w,m} & a_{p,m} \end{bmatrix}}_{\mathcal{A}} \cdot \underbrace{\begin{bmatrix} \Phi_1 \\ \Phi_2 \\ \vdots \\ \Phi_{m-1} \\ \Phi_m \end{bmatrix}}_{\overrightarrow{\Phi}} = \underbrace{\begin{bmatrix} S_{u,1} \\ S_{u,2} \\ \vdots \\ S_{u,m-1} \\ S_{u,m} \end{bmatrix}}_{\overrightarrow{S}} \tag{2.21}$$

The *eq. (2.21)* can be written in compact notation as *eq. (2.22)*.

$$\mathcal{A}\,\overrightarrow{\Phi} = \overrightarrow{S} \tag{2.22}$$

This matrix equation can be solved for the governing state variable vector $\overrightarrow{\Phi}$ with every arbitrary matrix solving algorithm e.g. Jacobi or Gauß-Seidel-method, see [26, pp.137].

2.2 Solution process within OpenFOAM

OpenFOAM originates from the FOAM program, which should full fill the task to deliver an extensible software environment for developing new models and to experiment with model combinations easily and reliable [11, p.1]. OpenFOAM is following the concept to integrate every equation over the finite volume elements and solve them in a segregated manner, where every vector and any additional transport equation are solved sequentially, see [18, pp.62]. This also means that the vector coupling is lagged. But the equation segregation allows OpenFOAM to use memory-efficient solvers. To solve for the cross coupling of the vector field OpenFOAM uses intermediate solution, see [15, p.2].

2.2.1 Matrix representation

OpenFOAM enables the user to write the equation in tensor notation e.g the governing state equation *eq. (2.1)* for steady diffusion would result in *lst. 2.1.*

```
solve
(
    fvm::div(fvm::grad(phi)*gamma)==0
)
```

Listing 2.1: "Solving of the steady diffusion equation."

The *fvm::div* calls the divergence operator on the product between the gradient of the state variable, represented by *fvm::grad(phi)* and the diffusion coefficient, represented with *gamma*. Because the divergence and gradient operator are defined for every tensor field, the above equation assembles a matrix with a_P the matrix coefficient at the point

P and Φ_p the corresponding value, and the products of the neighbour matrix coefficients a_N with the corresponding neighbour values Φ_N are summed together. This leads with the source term S_P to a definition of a matrix line *eq. (2.23)*, see [11, p.3].

$$a_P \Phi_P + \sum_i a_i \Phi_i = S_P \tag{2.23}$$

As seen in *eq. (2.22)* a matrix A must be represented through the code. The base class for expanding every operation on tensors is the Vector-Space class. This class automatically expands the algebraic operators on tensors of different rank, this includes also the inner and outer product, see [11, p.3]. The next important object is the mesh object which is handled by the *polyMesh* base class, this stores the mesh as a list of cells. Every cell itself contains a list of faces and every face consists out of a list of points. But the information stored in this container does not necessarily suit the needs of the finite volume method. Therefore the *fvMesh* is derived from that base class, this holds also the sparse matrix addressing. The temporal dimension of the simulation is handled by the time class, which is integrated in a data-handling system. The tensor field values are handled by the two classes called *GeometricField* and *GeometricBoundaryField*, which is a list of patch fields. A patch is a boundary surface of the mesh. The operator discretisation produces a sparse matrix represented by the *lduMatrix* class, this also implements the necessary linear equation solvers, see [15, p.4]. To solve the equation only the method *solve* of the *fvMatrix* must be called, this call leads to the following execution of methods visualized in *fig. 2.5* using the UML 2.0 language standard, see [22, pp.107].

After the solving call the first is to add the boundary terms to the *source_* member variable of the matrix. After that all the interface coupling patches are extracted, which can be used for the following treatment of the internal-field matrix. A loop runs over all the components of the field, which leads to the segregated solving of fields of higher rank. In that loop a local variable *psiCmpt* is generated, which holds the component values of the internal-field. To that variable the source components are added into the corresponding diagonal of the internal-field values, by the *addBoundaryDiag* method. After that the interface values are updated, this treats those boundaries which couple two physically separated domains. The whole procedure can be seen as a general method for building the necessary matrix, which can be used for every kind of iterative solution method such as Gauss-Seidel or AMG.

2.3 Boundary conditions in OpenFOAM

Every fluid dynamic problem can be defined by initial and boundary conditions. Therefore it is important to understand how boundary conditions are treated in OpenFOAM. In general only two kinds of boundary conditions build the base for a diversity of boundary conditions.

- standard boundary condition
- coupled patch boundary condition

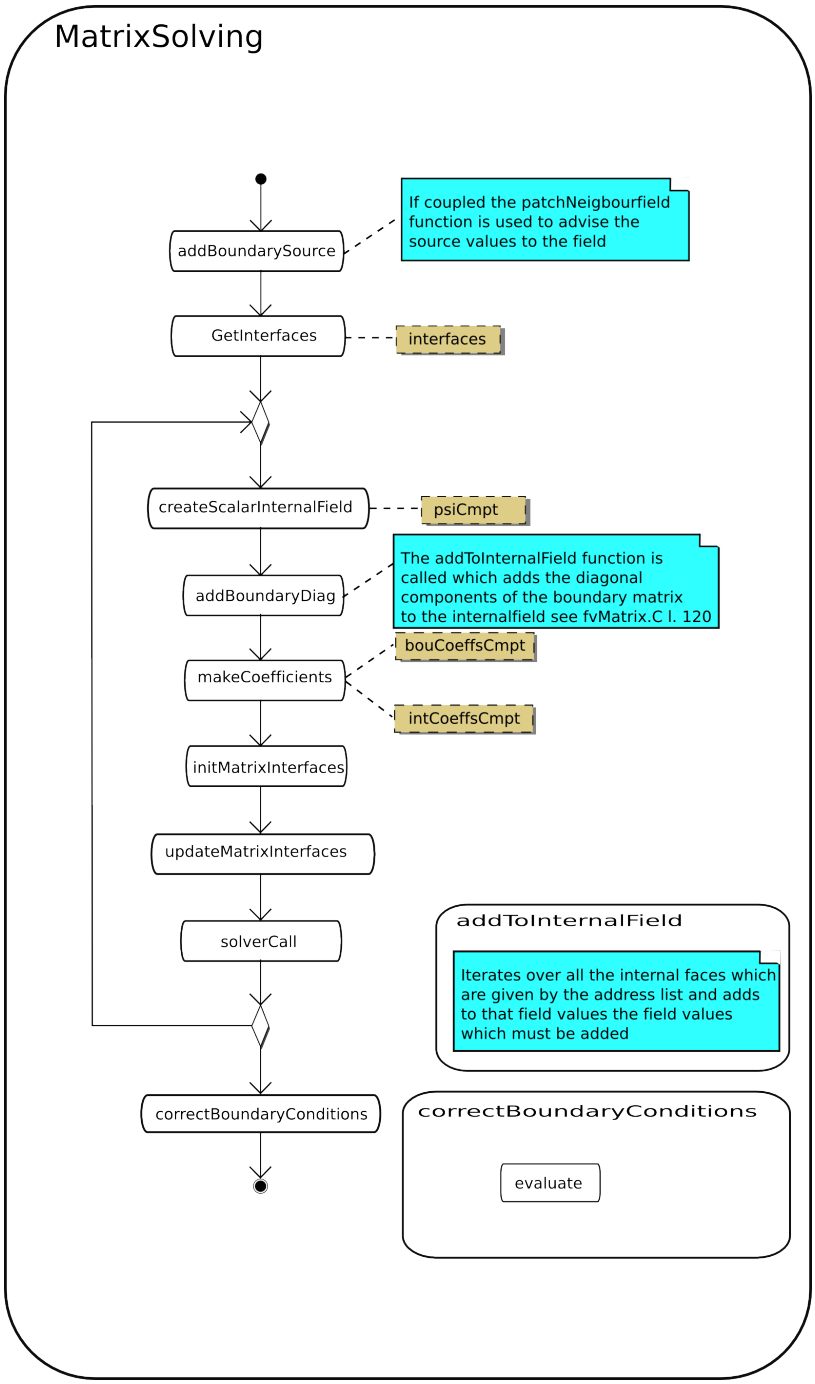


Figure 2.5: Showing the sequence diagram for the call of the solve method for the *fvMatrix* class. The UML 2.0 notation was used according to [22, pp.107]

Because every boundary condition is composed by either a mixture of the gradient and the value boundary condition, or by implementing a derived coupled patch boundary condition also known as interface boundary condition. Which files and classes are needed to implement boundary conditions is described in *sec. A.4*. Also a general idea of CFD is to split a boundary condition into a fixed value part, see *sub. 2.3.2.1* and a gradient part, see *sub. 2.3.2.2*. For handling of special boundary conditions, like the cyclic boundary condition, this approach is not applicable. For those an interface boundary condition must be implemented, see *sub. 2.3.1*.

2.3.1 Interface boundary condition

The most sophisticated boundary condition is the interface boundary condition. But it is also one of the most difficult things to implement in OpenFOAM. For understanding how an interface boundary condition is working a cyclic boundary condition was implemented utilizing the *GGIinterpolation* to be versatile enough to handle cyclic boundary conditions composed out of multiple patches, which have non conformal faces. Because OpenFOAM has already a cyclic boundary condition this can be used to validate the own implementation. For the solver this kind of boundary condition is not recognized as a boundary condition, because it seems to be connected with the neighbouring cells. The following steps must be performed within an interface boundary condition object.

1. Calculation of the cell face centres.
2. Calculation of the delta coefficients.
3. Computing a transformation which adopts the flow-field for the new spatial position.

For an interface boundary condition, there are two spatial boundaries which must be connected virtually within the matrix. This is done in OpenFOAM by three methods the *InitInterfaceMatrix* method, the *UpdateInterfaceMatrix* method and the *NeighbourPatchField* method.

2.3.1.1 Weights calculation

For calculating the flow field values at a surface it is necessary to calculate weights to build a weighted sum between the west and east node values. Therefore the surface patch field values Φ_{surf} can be calculated with *eq. (2.24)* by multiplying the neighbour field value Φ_n with the weighting factor w summed up with the corresponding weighting factor multiplied with the internal field value Φ_{inf} which belongs to the cell next to the patch.

$$\Phi_{surf} = w\Phi_n + (1 - w)\Phi_{inf} \tag{2.24}$$

The necessary elements to build the weighting factors for the interface boundary conditions are shown in *fig. 2.6*. The weights are calculated with *eq. (2.25)*, which corresponds very well to *eq. (2.8)* and *eq. (2.10)*. This is the basic formula for the 1D case.

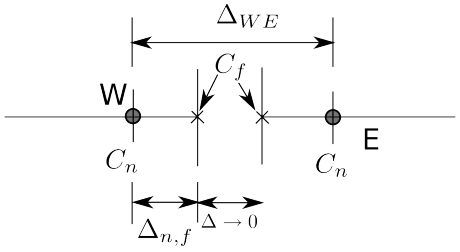


Figure 2.6: Visualized distances for weights building according to the OpenFOAM source code within the original cyclic *fvPatchField* class.

In 3D one has to take care about the face normals, the calculation of the distances is explained in *sub. 2.3.1.2.*

$$w = \frac{\Delta_{n,f}}{\Delta_{WE}} \tag{2.25}$$

2.3.1.2 Non conformal mesh handling

In case of a non conformal mesh at the interface patches, an interpolation must be made to generate matching delta lists, to enable the correct weights calculation. This is visualized in *fig. 2.7*, for easy explanation the big triangle shaped cell edge length is twice the edge length of the little triangle shaped cell. By interpolation with the *GGI-interpolation* algorithm the distance delta vectors are added together with the weights $w_{ggi,1}$, $w_{ggi,2}$. In that case both are equal to 1 because the face edge of one little cell fits into one big cell, see *sec. A.2.* In 3D and even 2D it is necessary to project the distance vector $\overrightarrow{d_1} + \overrightarrow{d_2}$ onto the face normal $\overrightarrow{n_f}$.

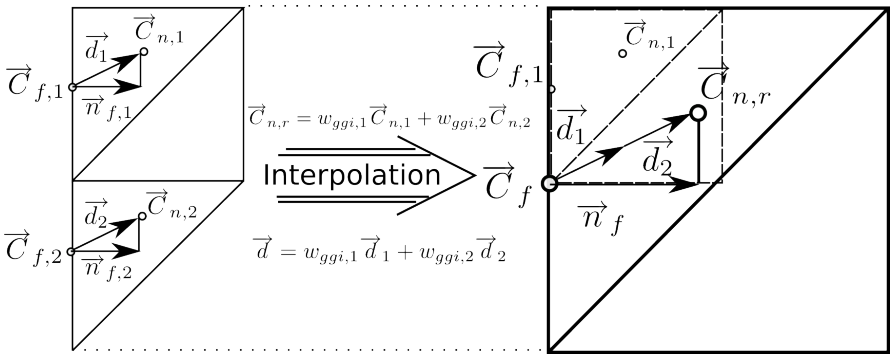


Figure 2.7: Different cell sizes need an interpolation to construct the correct distances necessary for the weights calculation.

Therefore the weights *eq. (2.27)* can be calculated by dividing the difference $\overrightarrow{d}_P$ *eq. (2.26)* between the patch face centres $\overrightarrow{C}_{f,P}$ and corresponding cell centres $\overrightarrow{C}_{n,P}$ projected onto the corresponding face normal $\overrightarrow{n}_f$ through the difference between the interpolated shadow cell centre see *fig. 2.7* and the patch cell centres also projected onto the face normal.

$$d_P = \overrightarrow{n_f} \bullet (\overrightarrow{C}_{f,P} - \overrightarrow{C}_{n,P}) \tag{2.26}$$

$$w = \frac{d_P}{\overrightarrow{n_f} \bullet (\overrightarrow{C}_{n,r} - \overrightarrow{C}_{n,P})} \tag{2.27}$$

The implemented cyclic boundary condition is a specialization for rotational geometry, therefore a rotational tensor is multiplied with the cell centre points before the deltas can be calculated, this is done with *eq. (2.28)*.

$$\overrightarrow{C}_{n,r} = \mathcal{R} \bullet \overrightarrow{C}_n \tag{2.28}$$

This is not the only rotational correction which must be applied. The same correction must be also applied for the field values in the *patchNeighbourField* function. To distinguish between vector-field, tensor-field and scalar-field a special transformation function is used in OpenFOAM.

2.3.1.3 Cross flux correction

Because the coefficients of the matrix are calculated with the distances in normal direction, this will lead to wrong delta coefficients. Especially for unstructured grids where the $\overrightarrow{\delta}$s are pointing in different directions than the face normal see *fig. 2.8(a)*. This wrong delta direction vector affects directly the flux value evaluated at the face surface see *eq. (2.29)*, see [30, pp.316].

$$\int_{\Delta A_i} \overrightarrow{n}_i \bullet (\Gamma grad\Phi) dA \approxeq \overrightarrow{n}_i \bullet (\Gamma grad\Phi) \Delta A_i \approxeq \Gamma \left(\frac{\Phi_A - \Phi_P}{\Delta_{PA}} \right) \Delta A_i \tag{2.29}$$

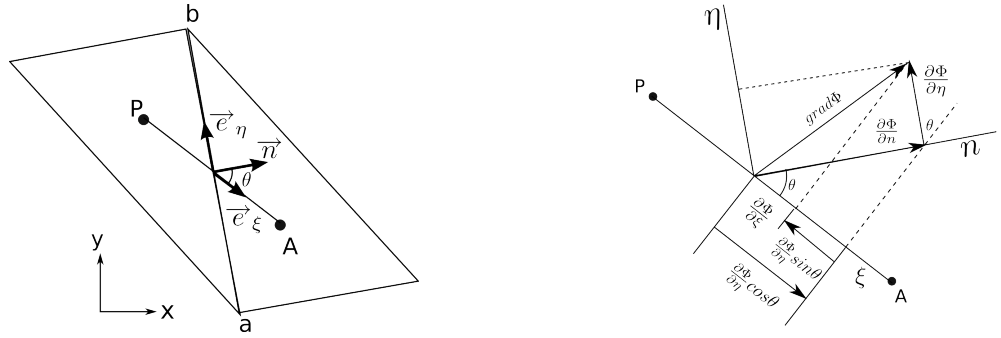
To deal with that wrong flux several correction methods can be found in [5], [24] and [16]. The different directions causing the splitting of the flux are shown in *fig. 2.8(a)*. The direction collinear to the vector between the two nodal points is called $\overrightarrow{e}_\xi$. The direction collinear to the vector between the two interface vertexes a and b is denoted with $\overrightarrow{e}_\eta$. It can be seen in *fig. 2.8(b)*, that the gradient of Φ is split up into a part co-linear to the normal direction $\frac{\partial \Phi}{\partial n}$ and a part $\frac{\partial \Phi}{\partial \eta}$ co-linear to the $\overrightarrow{e}_\eta$ base vector, which forms together with $\overrightarrow{e}_n$ a rectangular coordinate frame.

It can also be seen in *fig. 2.8(b)* that the sum of the two flux projections onto the $\overrightarrow{e}_\xi$ vector *eq. (2.31)*, *eq. (2.30)* gives the total flux in $\overrightarrow{\xi}$ direction *eq. (2.32)*.

$$\frac{\partial \Phi}{\partial n} \overrightarrow{n} \bullet \overrightarrow{e}_\xi = \frac{\partial \Phi}{\partial n} cos(\theta) \tag{2.30}$$

$$\frac{\partial \Phi}{\partial \eta} \overrightarrow{e}_\eta = -\frac{\partial \Phi}{\partial \eta} sin(\theta) \tag{2.31}$$

$$\frac{\partial \Phi}{\partial \xi} = \frac{\partial \Phi}{\partial n} cos(\theta) - \frac{\partial \Phi}{\partial \eta} sin(\theta) \tag{2.32}$$



(a) non normal direction vector

(b) correction strategy

Figure 2.8: According to [30, p.317] the cross flux problem, resulting from the deviation to the face normal direction, is visualized.

By again approximating the two gradients $\frac{\partial\Phi}{\partial n}$,$\frac{\partial\Phi}{\partial\xi}$ with central differencing schemes this leads to the *eqs. (2.33)*.

$$\begin{aligned} \frac{\partial\Phi}{\partial\xi} &= \frac{\Phi_A - \Phi_P}{\Delta\xi} \\ \frac{\partial\Phi}{\partial\eta} &= \frac{\Phi_A - \Phi_P}{\Delta\eta} \end{aligned} \tag{2.33}$$

The $\Delta\xi$ is the distance between the two Points P and A and the $\Delta\eta$ is the distance between the two vertexes a and b. The cross flux is corrected in OpenFOAM by the use of a correction vector $\overrightarrow{cv}$ which is the difference between the normal direction $\overrightarrow{n}$ and the scaled delta vector $\overrightarrow{\delta}_s$ *eq. (2.34)*. This formula can be found in the method *makeCorrVecs* of the class *coupledFvPatch*. Therefore there is no need to implement a custom cross flux correction.

$$\overrightarrow{cv} = \overrightarrow{n} - \underbrace{\frac{1}{\overrightarrow{n} \bullet \overrightarrow{\delta}}}_{cos\theta} \overrightarrow{\delta} \tag{2.34}$$

The *eq. (2.34)* is visualised in *fig. 2.9*, it can be seen that the projection of $\overrightarrow{\delta}$ onto the normal $\overrightarrow{n}$ is used to scale the length of $\overrightarrow{\delta}$, so that the difference vector $\overrightarrow{cv}$ is perpendicular to the normal direction and hence also parallel to the before mentioned $\overrightarrow{e}_\eta$ direction. Therefore *eq. (2.34)* defines the $\overrightarrow{e}_\eta$ direction, which is used to correct the flux.

2.3.1.4 Respecting the attitude of field components

Because a coupled field is most of the time used to connect fields with different attitudes, it is necessary to apply transformation operations onto the field components. This is done within a class derived of the *coupledFvPatchField* in the two methods *patchNeighbourField* and *updateInterfaceMatrix*. The *patchNeighbourField* delivers for the master

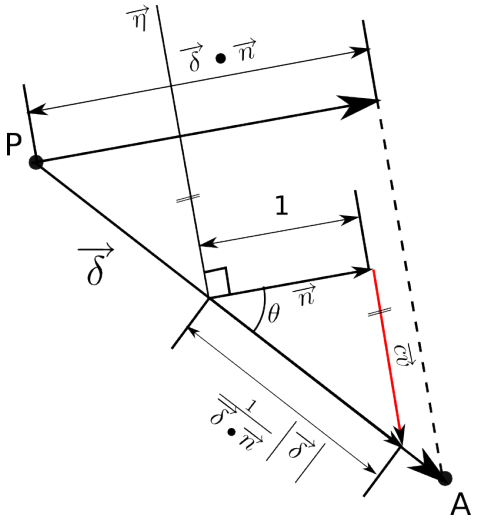


Figure 2.9: The geometric relationships, which define the flux correction vector for OpenFOAM.

side the transformed field corresponding to the cells at the shadow side, see *fig. 2.10* . The method *updateInterfaceMatrix* is called during every iteration step of the solver and delivers for the master side the seperate transformed field components corresponding to the shadow side.

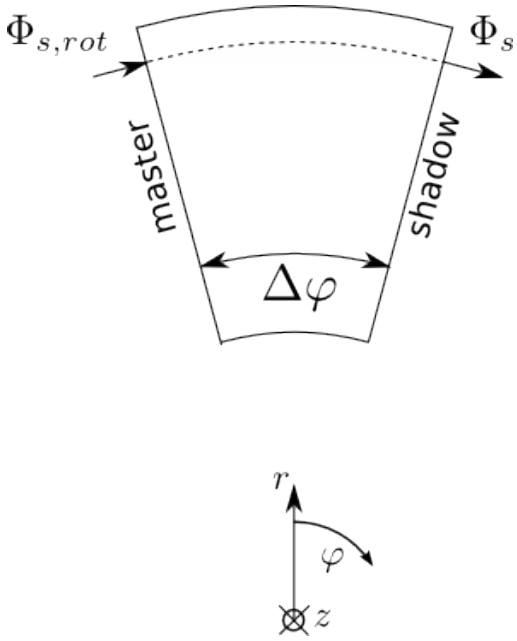


Figure 2.10: Need to apply rotation about $\Delta\varphi$ to the shadow state variable Φ_s to use it at the master side, forming a cyclic interface.

Because the *patchNeigbourField* delivers all field components e.g., a vector-field is not delivered component-wise. Therefore it is easy to rotate that delivered vector field by applying a rotation tensor operation. In case of a scalar field things are not so easy. For those problems a global function is embedded in OpenFOAM , called *transform*,

which takes a tensor and a arbitrary field as input. This is used to transform every kind of field (scalar, vector, tensor). But the method *updateInterfaceMatrix* has only access to the components of the state variable, to perform rotational scaling onto those components, the standard cyclic boundary condition is using *eq. (2.35)*.

$$\Phi_{c,rot} = \Phi_c diag\left(\mathcal{R}\right)^r \tag{2.35}$$

2.3.1.5 Validation of the implemented cyclic interface

Because the standard OpenFOAM library contains already a cyclic interface, this has been used as a reference to verify that the understanding of the interface boundary condition is deep enough for the implementation of new interface boundary conditions. Also the non conformal mesh handling mentioned in *sub. 2.3.1.2* can be checked to proof that the deltas and delta coefficients are handled correctly. To test all that a simple test case has been set up, one utilizing the original cyclic boundary condition and the other one utilizing the cyclic boundary condition capable handling the non conformal mesh. After the run has been performed on both cases they showed the same flow field distribution at the end of the simulated time and during all other time steps. The same was true for the course of the residuals. Therefore it can be stated, that the cyclic interface boundary condition was correctly adopted for the non conformal mesh handling.

2.3.2 Standard boundary conditions

In case of usual boundary conditions things are a little bit different, compared to the interface boundary conditions. First of all the node points are not treated as belonging to the internal grid, therefore there is no need to calculate the $\overrightarrow{\delta}$s and the $\overrightarrow{c}$s in an adopted way, those are calculated by the underlying *fvPatch* which is the geometric base class for all the boundary conditions. The calculation of those values is done according to [21, p.51] by computing only the distance from the boundary face centre to the cell centre, see *fig. 2.4*. The only tasks which must be performed in case of usual boundary conditions is to add the right coefficient values to the boundary and the internal field. This can be done by the following methods provided by the base class *fvPatchField*:

- The method *valueInternalCoeffs* sets the convective coefficients for the internal field value.
- The method *valueBoundaryCoeffs* sets the convective coefficients for the boundary field value.
- The method *gradientInternalCoeffs* sets the diffusive coefficients for the internal field.
- The method *gradientBoundaryCoeffs* sets the diffusive coefficients for the boundary field.

Those four functions are defining almost every kind of boundary field. The two methods starting with "value" are setting the boundary coefficients for the convective part and the two methods starting with "gradient" are setting the boundary coefficients for the diffusive part, see [10]. To illustrate how those functions are working, the fixed boundary condition and the gradient boundary condition are explained in the following.

2.3.2.1 Fixed value boundary condition

In OpenFOAM a fixed boundary condition can be implemented by setting the following return values of the before mentioned functions, see *tab. 2.3.2.1.* Comparing the two diffusive parts from *tab. 2.3* with S_P and S_u in *tab. 2.2* shows that only the δ values are used, no diffusion coefficient Γ is set in those functions. The *gradientBoundaryCoeffs* sets the diffusive S_u part, by multiplying the delta coefficient with the Φ value extracted from the boundary patch Φ_{BC}. The *gradientInternalCoeffs* sets the diffusive S_P part, by only returning the delta coefficient.

coefficient name	value
valueInternalCoeffs	0
valueBoundaryCoeffs	Φ_{BC}
gradientInternalCoeffs	-δ
gradientBoundaryCoeffs	$\Phi_{BC}\delta$

Table 2.3: Boundary condition method return values for a fixed gradient boundary condition.

The convective part can be derived by looking at the steady convection and diffusion equation for a property Φ, *eq. (2.36)* according to [30, p.135]. This equation describes the steady convection and diffusion of a property Φ in a given one dimensional flow-field u.

$$\frac{d}{dx}(\rho u \Phi) = \frac{d}{dx}\left(\Gamma \frac{d\Phi}{dx}\right) \tag{2.36}$$

After integrating this equation as described in *sec. 2.1* over a control volume, the *eq. (2.37)* is derived. The convective part stands on the left hand side of *eq. (2.37)*. It can be seen that no delta coefficient is involved within the convection part.

$$(\underbrace{\rho c A}_{F}\Phi)_e - (\underbrace{\rho c A}_{F}\Phi)_w = \left(\Gamma \frac{d\Phi}{dx}\right)_e - \left(\Gamma \frac{d\Phi}{dx}\right) \tag{2.37}$$

For a node with an adjacent boundary condition as shown in *fig. 2.4*, this integration transforms to *eq. (2.38)* with $A = BC$ denoting the boundary condition value.

$$\frac{F_e}{2}(\Phi_P + \Phi_E) - F_{BC}\Phi_{BC} = \delta_{PE}\Gamma_e(\Phi_E - \Phi_P) - \delta_{BC}\Gamma_{BC}(\Phi_P - \Phi_{BC}) \tag{2.38}$$

This can be rearranged to *eq. (2.39)*.

$$\left(\frac{F_e}{2} + \delta_{PE}\Gamma_e + \delta_{BC}\Gamma_{BC}\right)\Phi_P = \left(\delta_{PE}\Gamma_e - \frac{F_e}{2}\right)\Phi_E + \left(F_{BC} + \delta_{BC}\Gamma_{BC}\right)\Phi_{BC} \quad (2.39)$$

In *eq. (2.39)* the part in braces at the left side is the coefficient for the actual node P. This sum does not contain a convection term related to the boundary condition. Therefore the corresponding *valueInternalCoeffs* function must return zero, for a fixed value boundary condition. For the *valueBoundaryCoeffs*, the $F_{BC}\Phi_{BC}$ is the corresponding coefficient. Because the *fvPatchField* class does not know about the convection coefficient F_{BC}, the *valueBoundaryCoeffs* returns Φ_{BC}. Those values are also listed in *tab. 2.3*.

But additionally to the definition of the correct return values for the coefficients, it must be assured that the boundary value remains fixed during the solving process. This is done by disabling all the operators for the *fixedValueFvPatchField* class which would change the values of the boundary condition.

2.3.2.2 Gradient boundary condition

The gradient boundary condition in example the zero gradient boundary condition or the radial equilibrium boundary condition can be implemented by using a similar approach as in *sub. 2.3.2.1*. According to the source code of the zero gradient boundary condition the following coefficients are used see *tab. 2.4*. Because the value of the internal coefficient is set to 1 the solution can adopt freely at this boundary. For a fixed gradient boundary condition the value of the boundary coefficient is set to a value corresponding to the gradient multiplied with the delta coefficient. Therefore zero is returned for the zero gradient boundary condition.

coefficient function name	value
valueInternalCoeffs	1
valueBoundaryCoeffs	0
gradientInternalCoeffs	0
gradientBoundaryCoeffs	0

Table 2.4: Boundary condition method return values for a fixed gradient boundary condition.

Chapter 3

Mixing-Plane

The mixing plane is a rotor-stator interface for steady state flow solutions. It utilizes circumferential averaging to eliminate the time dependence of the rotor-stator interaction. This enables the mixing plane also to deal with arbitrary pitch relations between the rotor and the stator.

3.1 Theoretical Idea

The principal idea of the mixing-plane is, that the interface only sees averaged values over time and therefore a circumferential average at the interface can be used to generate a steady solution of the flow-field, see [6, pp.29]. The mixing-plane couples the flow-field solutions between two adjacent blade rows by averaging in circumferential direction the flow quantities necessary for the conservation law at an inter-row plane, called the mixing-plane interface. This averaging procedure is applied to the upstream exit-plane and the downstream inlet-plane see [27, pp.3]. The circumferential averaged radial profiles for the density, temperature, velocity (radial, tangential and axial), extracted from the upstream exit-plane are used to specify the downstream inlet boundary values. For the upstream outlet boundary condition the averaged pressure values extracted from the downstream inlet plane are used. The principal of this mixing-plane is not to connect both domains with a coupling interface but to transfer the values from one region averaged onto the other patch and use those values for a fixed value boundary condition.

In case of a supersonic flow-field the pressure is extracted from the upstream patch and the circumferential average is used for the downstream inlet plane boundary condition. To use the mixing-plane with transonic problems could cause problems. It can happen that the interface region is subsonic, but the mixing-plane was configured for supersonic flow. This type of interface can also be used for domains with different pitches, because the values are averaged in circumferential direction. This is also the reason why the solution obtained with the mixing-plane approach is independent on the azimuthal rotor position.

3.2 Implementation in OpenFOAM

Up until now no mixing-plane for OpenFOAM version 1.4.1 has been available for professionals in the turbo-machinery field. The following requirements should be met by the first version of implementation in OpenFOAM.

1. The theoretical considerations of a mixing-plane should be met.
2. The mixing-plane should be able to handle non conformal meshes.
3. The implementation should handle rotor and stator domains which posses different pitches.
4. The conservativeness must be met.
5. It should be able to handle radial, axial and mixed flow turbo machinery's.

3.2.1 Idea to solve the problem

The main problem in implementing a mixing-plane is how to determine which values lie on the same radial level. OpenFOAM has an interpolation class called *GGIinterpolation*, GGI means general grid interface. This *GGIinterpolation* class can interpolate conservatively between two patches, for an explanation of the working principal, see *sec. A.2*. Therefore the idea raised to use this interface to interpolate from an arbitrary structured grid boundary patch onto an ordered grid patch, which is structured to account for the need to map the interpolated values easily to radial levels. This idea is illustrated in *fig. 3.1*. It can also be seen that the structured patch, in the following called virtual patch, is ordered in a way, that every circumferential strip has a regular labeled number of faces to access the mapping between faces and values in an easy way.

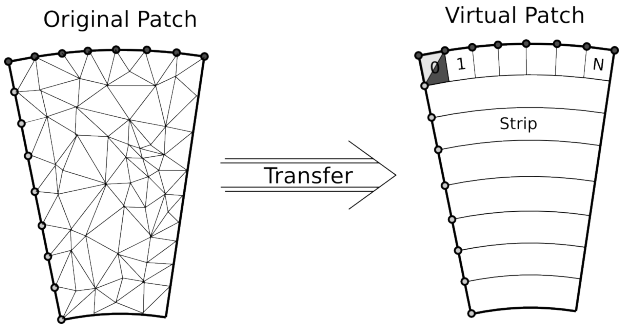


Figure 3.1: The values are transferred by a conservative interpolation from an arbitrary structured patch onto the regular structured patch.

The general averaging process is shown in *fig. 3.2*. First the values are transferred by the above mentioned interpolation onto the virtual patch VP-R in *fig. 3.2*, then the circumferential average is built on that virtual patch. After the averaging has been performed on the virtual patch, the averaged values are ordered in a structured list. Because every circumferential strip has only one averaged state variable value, this structured list is transferred to the neighbour patch and used to generate a averaged

field, *fig. 3.2* step two. Therefore it must be assured that the neighbour patch has the same number of circumferential strips with equal radial levels. This is assured by making one patch the master patch and the other the slave patch. The slave patch geometry adopts for the master patch radial levels, which are transferred by a list of starting points, defining the radial coordinates for the new structured slave patch.

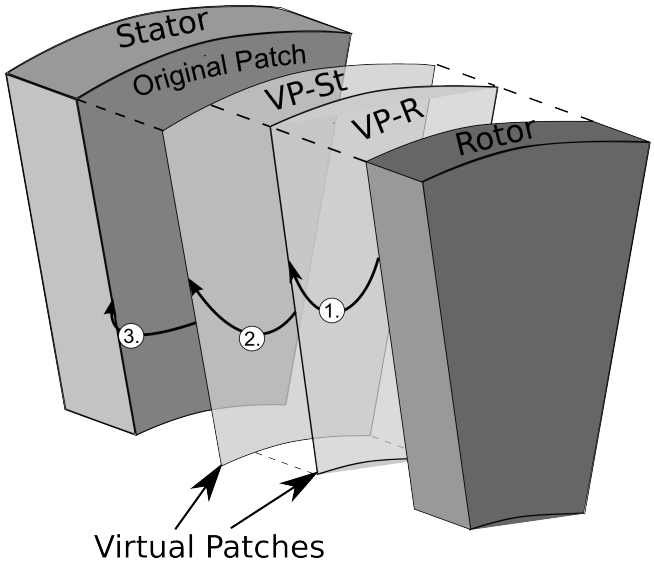


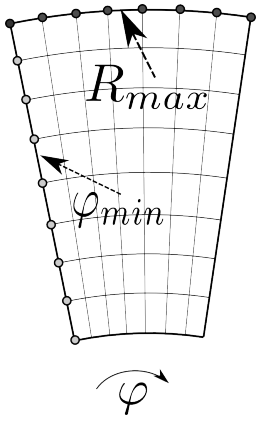
Figure 3.2: Basic idea for exchanging the averaged state variables between the different domains.

After the values are distributed on the neighbour virtual patch, in *fig. 3.2* VP-St, the values are interpolated with the *GGIinterpolation* onto the original neighbour patch in the third step. This procedure assures that the mixing-plane implementation can handle rotor-stator passages with differing pitch angles.

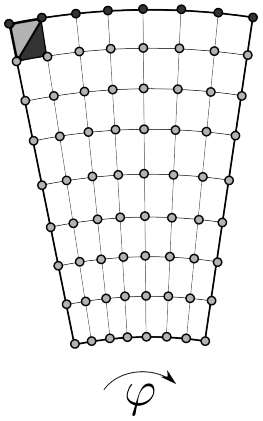
3.2.2 Geometry treatment

As explained before, the virtual patch geometry must be treated in a way that the circumferential average can be easily generated, after the original field has been interpolated onto the virtual patch. To build such a virtual patch the idea is to use the borders of the original patch to use the radius values on the border with the minimum averaged circumferential angle $\bar{\varphi}_{min}$ and the circumferential angle values φ at the border with the maximum radius value R_{max} see *fig. 3.3(a)* to generate the new points for the regular virtual patch see *fig. 3.3(b)* .

In OpenFOAM a *primitivePatch* is just a list of faces. Because every face in OpenFOAM contains a list of points and a list of edges, the *primitivePatch* can extract also a list of all its edges. From those edges a subset of edges denoting the internal edges can be retrieved from the *primitivePatch*. By knowing the internal edges, one can extract easily the external edges from the whole list of edges. Because the edge list returned by the *primitivePatch* is not ordered in a consecutive way, a sorting algorithm must be performed to order the edges. For that task, a merge sort algorithm has been used,



(a) R_{max} and φ_{min} points



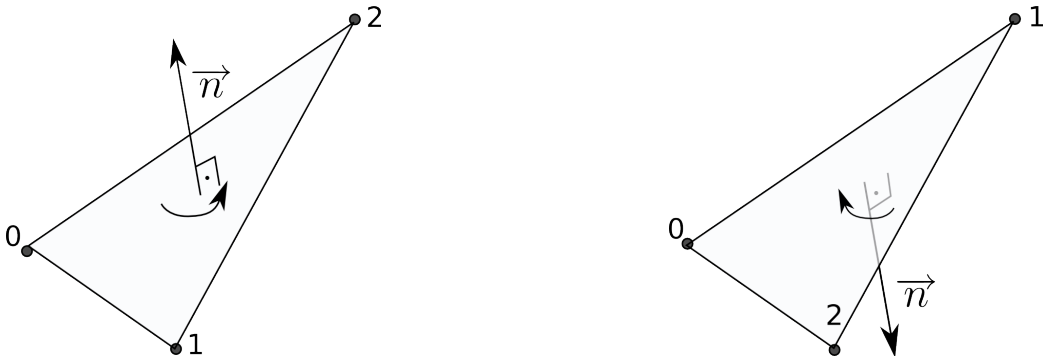
(b) newly constructed faces

Figure 3.3: (a) shows the points used for the construction of the new points in (b) which are used to built the new faces mapped to several stripes.

because of its stability and sorting speed. After this has been done, the last step for getting the necessary point coordinates from this edge list is, to advise the parts of the consecutive ordered edge list to the borders. To map edges to borders, it is necessary to find the corners, which connect two borders of the patch. This is done by calculating the angles between two consecutive edges and storing those in a list containing a corner data structure, which is also able to handle the mapping between corners and edges. This corner list is sorted for the absolute cosinus value of the angle between the two edges. This makes it possible to extract the four edge pairs which have the smallest cosinus value. After knowing those edge pairs one can easily extract the borders lying between them. After that, the borders can be sorted for $\bar{\varphi}_{min}$ and R_{max} to get the point coordinates forming the new points for the faces as shown in *fig. 3.3(a)*. To build the faces it is necessary to know that the *GGIinterpolation* is sensitive to the face normal direction of the participating patches. Therefore the faces must be built in a way, that the face normal directions are pointing into the opposite direction of the original face normals. This is assured by building an average face normal $\bar{\vec{n}}$ of the original patch over all it's incorporated faces *eq. (3.1)* with N the number of all faces.

$$\bar{\vec{n}}_{CCS} = \frac{\sum \vec{n}_{CCS,i}}{N} \tag{3.1}$$

This average face normal is used to check whether the newly generated faces are pointing in the opposite direction than the original patch face normals. This is done in the cylindrical coordinate system, because this average face normal can then be used to check if the turbo-machinery is a axial, radial or a mixed-flow machinery. To change the normal of a face one must just reorder it's point labels. As shown in *fig. 3.4(a)* every face is a list of point labels, the ordering of the points in that label list determines by the right hand rule into which direction the face normal will point. According to *fig. 3.4(b)* it is easy to change the direction of the face normal by just exchanging one point label with the consecutive label.



(a) original normal (b) reversed normal

Figure 3.4: Face normals are pointing into the direction given by the right hand rule orientated constrained by the point labels.

3.2.3 Average generation

After the virtual patch has been generated for every strip an average can be built by calculating either a mass weighted average for the extensive state variables Φ_{ex} and a area weighted average for the intensive state variables Φ_{in} [32, pp.52]. For every strip j the area weighted average can be built with *eq. (3.2)* with i the index of the face within the strip.

$$\bar{\Phi}_{in,j} = \frac{\sum A_i \Phi_{in,i}}{\sum A_i} \tag{3.2}$$

The mass weighted average can be calculated by *eq. (3.3)* $\overrightarrow{n}_i$ denotes the face normal of face i the dot product between this normal and the velocity $\overrightarrow{c}_i$ defines the mass-flow after it was multiplied with the density ρ_i. For the incompressible case with constant density, this equation reduces to *eq. (3.4)* by eliminating the constant ρ_i.

$$\bar{\Phi}_{ex,j} = \frac{\sum A_i \rho_i (\overrightarrow{n}_i \bullet \overrightarrow{c}_i) \Phi_{ex,i}}{\sum A_i \rho_i (\overrightarrow{n}_i \bullet \overrightarrow{c}_i)} \qquad compressible \tag{3.3}$$

$$\bar{\Phi}_{ex,j} = \frac{\sum A_i (\overrightarrow{n}_i \bullet \overrightarrow{c}_i) \Phi_{ex,i}}{\sum A_i (\overrightarrow{n}_i \bullet \overrightarrow{c}_i)} \qquad incompressible \tag{3.4}$$

For an averaged flow-field vector component, as it is the case for the averaged velocity, it is important to note that an average in cartesian coordinates also generates an average for the direction of the vector-field. For axial machines where the mixing-plane normal points into the z direction this does not have an influence on the solution. But for radial and mixed-flow machines it is important to treat the direction averaging in an appropriate way. The best solution is to transform the vector-field into the cylindrical coordinate system with *eq. (3.5)*, before the averaging is performed. For the derivation of *eq. (3.5)*, see *chap. A.1.1.*

$$\begin{bmatrix} \overrightarrow{\Phi}_{i,r} \\ \overrightarrow{\Phi}_{i,\varphi} \end{bmatrix} = \underbrace{\begin{bmatrix} cos(\varphi) & -sin(\varphi) \\ sin(\varphi) & cos(\varphi) \end{bmatrix}}_{T} \begin{bmatrix} \overrightarrow{\Phi}_{i,x} \\ \overrightarrow{\Phi}_{i,y} \end{bmatrix} \tag{3.5}$$

In *eq. (3.5)* $\overrightarrow{\Phi}_{i,x}$ and $\overrightarrow{\Phi}_{i,y}$ are the vector-field x and y components in the cartesian coordinate system and the $\overrightarrow{\Phi}_{i,r}$ and $\overrightarrow{\Phi}_{i,\varphi}$ are the vector-field components in the radial and azimuthal direction. Because the z component remains the same only the 2D transformation is shown. The cosinus and sinus are calculated by using the azimuthal component φ of the face center point corresponding to the vector-field component $\overrightarrow{\Phi}_i$, to adopt for the correct position in space. After the transformation of the flow-field has been performed it is averaged in the cylindrical coordinate system. The next step is to transform the average back to the original rectangular coordinate system by using *eq. (3.6)* and transform the values for every face back into the rectangular coordinate system.

$$\begin{bmatrix} \overrightarrow{\Phi}_{i,x} \\ \overrightarrow{\Phi}_{i,y} \end{bmatrix} = \underbrace{\begin{bmatrix} cos(\varphi) & sin(\varphi) \\ -sin(\varphi) & cos(\varphi) \end{bmatrix}}_{T^{-1}} \begin{bmatrix} \overrightarrow{\Phi}_{i,r} \\ \overrightarrow{\Phi}_{i,\varphi} \end{bmatrix} \tag{3.6}$$

Because the z coordinate axis of the rectangular coordinate system does not necessarily coincide with the z coordinate axis of the cylindrical coordinate system, a special treatment for those cases is necessary. To take that into account, the Rodriguez rotary tensor has been used to transform the vector-field between the different coordinate systems, see *sub. A.1.2*. This leads to a list of tensor objects which are mapped to the corresponding faces. This list of tensor objects is only build once for the transformation from the rectangular coordinate system into the cylindrical coordinate system. Because the inverse of a rotary matrix is the transposed tensor [19, pp.316] the same list can be used to transform from the cylindrical coordinate system to the rectangular coordinate system by just transposing the tensor list elements $\mathcal{R}_i$. The equation *eq. (3.7)* transforms every ith field component $\overrightarrow{\Phi}_i$ of the field by multiplying with the tensor list element $\mathcal{R}_i$ into field component $\Phi_{CCS,i}$ in the cylindrical coordinate system.

$$\overrightarrow{\Phi}_{CCS,i} = \mathcal{R}_i \bullet \overrightarrow{\Phi}_i \tag{3.7}$$

The *eq. (3.8)* is used to transform the component in the cylindrical coordinate system by multiplying it with the transposed rotary tensor element back into the rectangular coordinate system.

$$\overrightarrow{\Phi}_i = \mathcal{R}_i^T \bullet \overrightarrow{\Phi}_{CCS,i} \tag{3.8}$$

The success of this method is shown in *fig. 3.5*. The red vectors show the direction of an averaged field without respecting the cylindrical coordinate system. The blue highlighted vectors show the direction after the averaging in the cylindrical coordinate

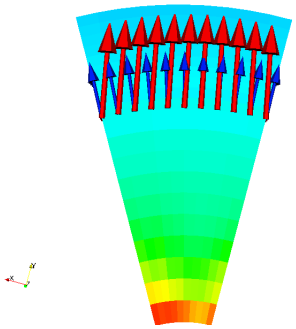


Figure 3.5: Showing the comparison between vectors averaged in the cylindrical coordinate system, small blue pointers, and vectors averaged in the rectangular coordinate system, big red pointers.

system has been used, those show a plausible direction distribution pointing normal to the interface surface.

3.3 Results

To show that the implementation of the mixing-plane was a success and also to prove that the procedure used is correct a test case has been set up to compare the computed results with results of commercial products. How to setup a mixing-plane case is described in *sub. A.5*. And at the end a future perspective of things, which can or should be improved is displayed.

3.3.1 Comparison with commercial codes

To make a validation of the results calculated with the implemented mixing plane approach a simple test case has been set up. This was used to compare it with a solution delivered by the Numeca solver. The flow-field of this test case was not turbulent and at a low Reynolds number $R = 1275$. To validate the upstream influence of the pressure field a simple geometric rhombus profile has been placed at the downstream domain shown in *fig. 3.6*. The magenta and red area are the inlet patches. For the magenta patch a velocity of $0.5\frac{m}{s}$ and for the red patch $1\frac{m}{s}$ was used to validate the value averaging in circumferential direction.

For the OpenFOAM solver no temperature was specified but for the Numeca solver a temperature of 293 K has been assumed and the fluid model incompressible air has been used. The kinematic viscosity for the *icoFoam* input file was set to $1.57 \cdot 10^{-5}\frac{m^2}{s}$. Right now the steady solver *simpleFoam* of the OpenFOAM library has not been given results in cooperation with the mixing-plane approach. Therefore a transient incompressible solver called *icoFoam* has been used, which requires only two fields to be specified. Termination criteria in that case was a residual under 10^{-4}. The commercial code used a solver especially suited for steady calculations. To compare the results a cut at the

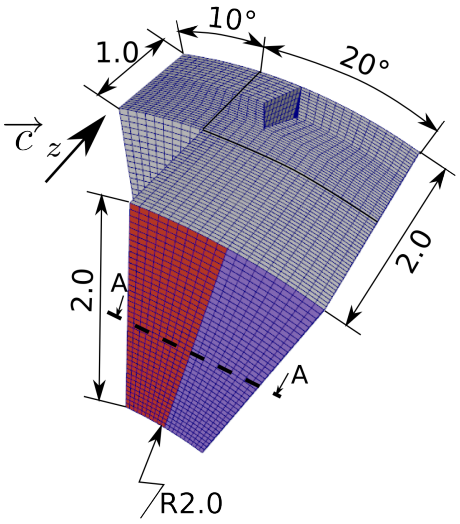
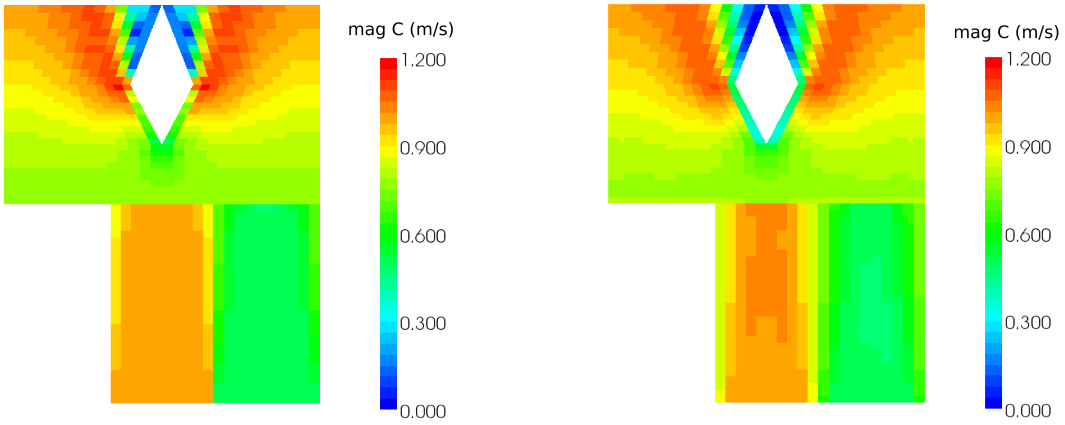


Figure 3.6: Showing the case setup for the mixing plane test case measurements are in centimeters.

half was made, which represents a plane with normal pointing into the y direction. This cut is marked in *fig. 3.6* with a thick dashed line denoted with A.

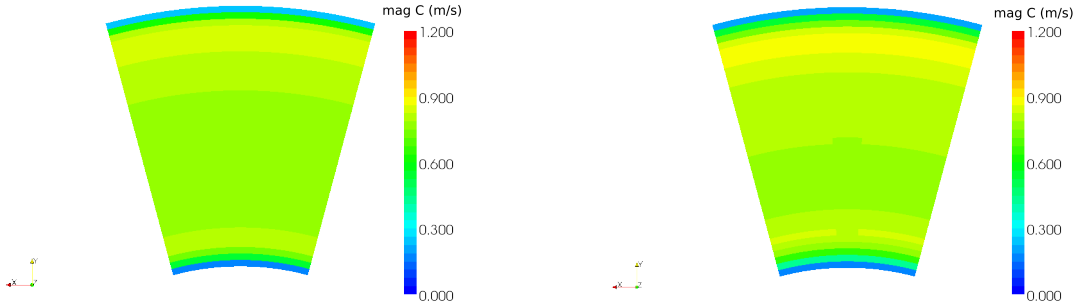


(a) OpenFOAM (b) Numeca

Figure 3.7: Mid cut, normal pointing into the y direction, through the calculation domain, displaying cell centered absolute velocity. The left calculated with OpenFOAM and the right calculated with Numeca.

The *fig. 3.7(a)* is the cut through the result calculated with OpenFOAM and the *fig. 3.7(b)* is the cut through the result delivered by the commercial code. Both show a high correlation in the gradients and the absolute velocity values. The magnitude of the OpenFOAM result is slightly higher than the result of the commercial code.

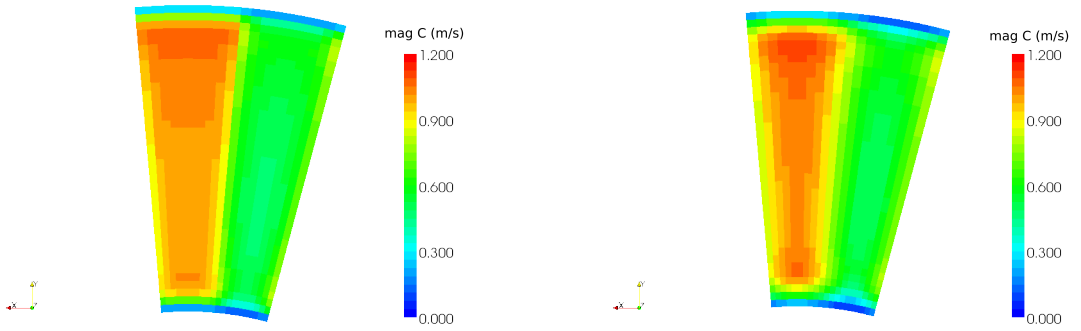
It can be seen in *fig. 3.7(a)*, that the velocity-field directly after the interface shows a completely averaged value. The Numeca case shows instead a small decrease of the velocity at the middle of the interface. This can be observed in the cuts *fig. 3.8(a)* and *fig. 3.8(b)* for the cell values directly after the rotor-stator interface. One explanation



(a) OpenFOAM (b) Numeca

Figure 3.8: Comparison of the stator cuts through the first cell next to the interface.

for that behavior, is a interpolation from originally noted node based values to cell based values.



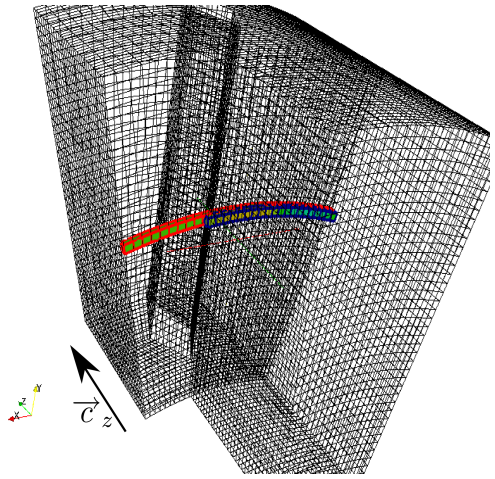
(a) OpenFOAM (b) Numeca

Figure 3.9: Comparison of the rotor cuts through the first cell next to the interface.

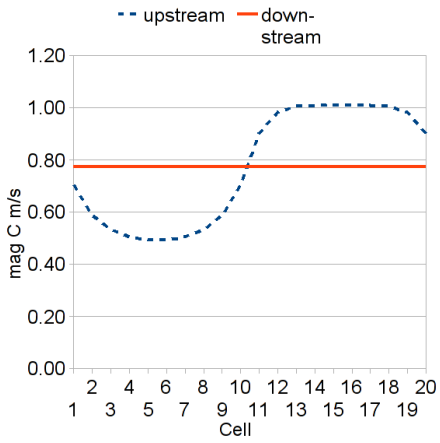
The cuts one cell upstream to the interface show for the OpenFOAM solution *fig. 3.9(a)*, for the side with the higher inlet velocity a little bit lower gradient in radial direction. It can be seen that the velocity reduces slightly slower from the maximum radius to the middle radius part. Instead the solution given by Numeca *fig. 3.9(b)* shows a higher gradient in azimuthal direction. But in general above results give a very high correlation in the absolute solution so it can be considered, that the implemented mixing-plane procedure treats both sides, upstream and downstream, in a correct way.

To show that the averaging is done correctly, a cell layer was extracted at the downstream and upstream region, both sharing the same radial level. The used cells are shown in *fig. 3.10(a)*. The values of the velocity magnitude in those cells are plotted in *fig. 3.10(b)*. The blue dashed line represents the upstream cell values and the red solid line represents the downstream averaged values. Because of the unsymmetrical velocity profile, the averaged value is between the low and the maximum of the blue dashed

line. Therefore it can be concluded, that the averaging is performed correctly and the mixing-plane behaves in the right way.



(a) side points adopted virtual patch



(b) side points adopted virtual patch

Figure 3.10: (a) shows the cells which are used to generate the plot shown in (b). Blue marks the downstream side and red the upstream.

3.3.2 Parallel computation

Right now the implemented mixing-plane is not able to handle parallel processing. This has to be implemented in the future. One way to handle the parallel processing is to virtually reconstruct the mixing-plane patches located at different processor regions by reading the information delivered at every processor directory for mapping between the processor regions and the original region. This would also require to access all the processor directories from every processor, which leads to the restriction of a shared data structure available at a network file systems. The other way would be to have an inter processor communication delivering the necessary information about points and faces and also face label addressing. The same procedure must be performed for the field values because those are also decomposed for every processor region.

3.3.3 Future improvements

The beforehand specified requirements, see *sec. 3.2* have been met by the current mixing-plane implementation. But the implemented approach has only a limited capability for geometry handling, which mainly results from the current building procedure for the virtual patches. One way to overcome that problem is illustrated in *fig. 3.11(a)*. It shows a way of building the patch by using not the original φ values delivered by the R_{max} border points as shown in *fig. 3.3(a)*, but using a fixed $\Delta\varphi$ for building the points on the different radial levels.

(a) side points adopted virtual patch

(b) second virtual patch

Figure 3.11: Alternate building of the regular patch which would improve the geometry compatibility.

This $\Delta\varphi$ is calculated by dividing the difference of the minimum φ value of the points lying on the R_{max} border line $\varphi_{min,R_{max}}$ and the corresponding maximum φ value $\varphi_{max,R_{max}}$ by the number of points lying on the R_{max} border line $N_{R_{max}}$, see *eq. (3.9)*.

$$\Delta\varphi = \frac{\varphi_{max,R_{max}} - \varphi_{min,R_{max}}}{N_{R_{max}}} \tag{3.9}$$

This procedure relies on the radial point coordinate values delivered by the φ_{min} border. This will result in a rotatory mapping of the φ_{min} border, by adding a constant φ to every azimuthal point coordinate on the φ_{min} border. To transfer the averaged values from that virtual patch to the neighbour virtual patch, both must have the same radial levels, to use a list of averaged values. But it can happen that the neighbour patch has a different distribution of border points, which will result in a different distribution of radial levels. In that case, no exchange by using a list of averaged values can be assured. To assure that both have the same radial levels, without changing the φ_{min} border curve course, may be difficult. One way would be to find the intersection of a constant radial level on the φ_{min} border and use the coordinates of that point to build the new points. The intersection of the φ_{min} border line with the radius levels of the neigbour domain seem to be possible by using the class *objectHit*.

But this procedure seems to be complicated compared to an approach, which uses two virtual patches for every side, one for the averaging and one for the interpolation of the received averaged values from the neighbour. This would lead to a weak coupling between the master and the slave patch, because only the patch for the interpolation will be built by receiving a list of radial levels from it's neighbour patch. The minimum and maximum azimuthal angles are determined by the points with extreme values, shown in *fig. 3.11(b)*. Those are the points which will lie on the borders of the newly generated virtual patch. The gray part in *fig. 3.11(b)* represents the original patch, and the white part represents the difference between the original and the virtual patch. The virtual patch used for the averaging will be built by the original patch values as explained in *sub. 3.2.2*. Referring to *fig. 3.12*, the first step is to interpolate the values from the original patch to the virtual patch, which shares the same border topology as the original patch.

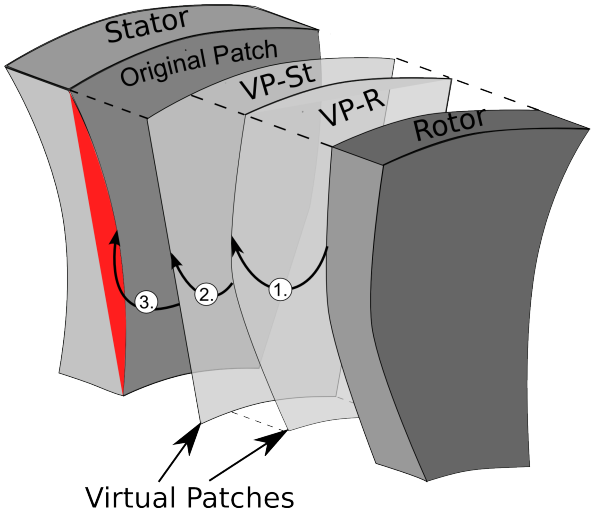


Figure 3.12: Showing the idea to handle different geometries by using a virtual patch with straight lines.

This virtual patch VP-R is used to average the values and generate the list of averaged values. The virtual patch of the neighbour is built in the before described way, therefore it shares the same radial levels and can distribute the averaged values from the received list over its surface. The third step interpolates the averaged values from the virtual VP-ST onto the original patch belonging to the stator. The red marked region on the stator surface represents the difference between the original stator patch and its virtual patch VP-ST. For other values which must be transferred from the stator to the rotor the same procedure is used, therefore every original patch will have two virtual patches one sharing the original border topology and one with straight borders.

3.3.4 Restrictions

The actual mixing-plane approach has some restrictions to the usage. If a solver utilizes fields, which are generated by internal operations through the solver, those fields must be exposed to the user. Otherwise it is not possible to set the correct parameters for the mixing-plane approach. Therefore the solver must be rewritten in a way that every used field has a file representation, in which the necessary parameters can be set.

Right now it must be noted that the mixing-plane approach can only be used to handle absolute frame values. If it is necessary to use the mixing-plane approach with a solver, which uses field values noted in the relative frame, a special treatment for those fields must be implemented. For a velocity noted in the relative frame $\overrightarrow{w}$, before the averaging can be performed a transformation into the absolute coordinate frame must be executed. This is done by adding the rotational velocity component $\overrightarrow{u}$ to the relative velocity component, see *eq. (3.10)*.

$$\overrightarrow{c} = \overrightarrow{w} + \underbrace{\overrightarrow{\omega} \times \overrightarrow{r'}}_{\overrightarrow{u}} \tag{3.10}$$

After the averaging was performed the averaged components must be transferred back into the relative coordinate frame by subtracting the rotational velocity component.

Chapter 4

Domain-Scaling

To cover unsteady solutions for turbo-machinery efficiently with OpenFOAM it is necessary to implement an interface capable to transfer the flow-field values from the rotating reference frame to the non rotating or counter rotating frame. Because unsteady simulations require a very fine mesh to resolve flow features with very small time scales and to accurately capture wakes [27, p.5], it is important to reduce the necessary memory by limiting the computational domain to one or two blade passages. One famous method for dealing which such a problem is the domain-scaling method, which is able to deal with computational regions possessing the same pitch.

4.1 Theoretical idea

The upstream boundary and the downstream boundary connecting the rotor and stator domain or the rotating and counter rotating domain can be described as a function of its spatial and temporal periodicity [27, p.6]. For equal pitches the state variable Φ at the upstream boundary is identical with that at the downstream periodic boundary at the same time. According to [23, p.8] this relation between the state variable of the second domain at the azimuthal position θ is equated with the corresponding azimuthal position in the first domain *eq. (4.1)*. The $\Delta\theta$ denotes the pitch angle and m is an arbitrary integer number.

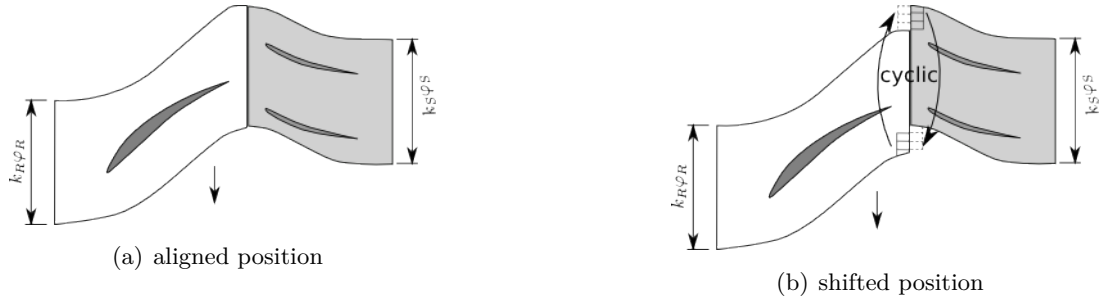
$$\Phi_i(\theta_i, t) = \Phi(\theta_i + m\Delta\theta_i, t) \tag{4.1}$$

The *eq. (4.1)* also holds for rotor-stator passages with different pitch angles if they can be related by two constant factors k_1 for the first passage and k_2 for the second cascade leading to *fig. 4.1*.

$$k_1\varphi_1 = k_2\varphi_2 \tag{4.2}$$

In that case a spatial periodicity condition shown in *fig. 4.1* can be used. The *fig. 4.1(a)* shows the starting position where both passages are aligned completely. After a time

step the position shown in *fig. 4.1(b)* can be observed. The two way pointer symbolize the cyclic exchange at the rotor-stator interface.



(a) aligned position

(b) shifted position

Figure 4.1: (a) shows a perfect alignment of the whole calculation domain and (b) shows the cyclic exchange to simulate the spatial periodicity. Virtual boundary cells are depicted with slashed lines.

4.2 Implementation in OpenFOAM

Until now the OpenFOAM CFD tools library does not offer the ability to calculate passages by using the above explained spatial cyclic boundary condition. Therefore it was the task to implement such a cyclic spatial interface to enable the OpenFOAM tools library to calculate passages with rotor-stator pitches restricted to *eq. (4.2)*. In the following the idea of the implementation will be explained.

4.2.1 Idea to solve the problem

The idea is to have two auxiliary patches constructed from the two original patches one lying on the master and one on the slave side. Because the starting part of the auxiliary patches is constructed by the same primitive geometry values (point, edges, faces) as the original patches they also share the same field value addressing with the original patch. After those are constructed the values are mapped onto them, fulfilling the condition described in *eq. (4.1)*. After that the exchange to the shadow patch is done by interpolation using the *GGIinterpolation* method. The whole interface should be implemented in a similar manner as a cyclic boundary condition. To implement that two implementations seem to be possible. One is using an interpolation sector *chap. 4.2.1.2* and the other is using an interpolation disk *chap. 4.2.1.1*. Both assume that the rotor domain is moving physically by changing the topology of the mesh, but should be able to handle still standing domains by moving the interface virtual patches.

4.2.1.1 Use of an interpolation disk

The easiest idea to implement is to have an auxiliary patch forming a whole disk out of the original parts. After the disk has been generated using the original patch primitive geometric values (points, edges, faces) as the first part of the disk, see *chap. 4.2.2*

for a detailed description of the geometry generation of the auxiliary patch, the state variable values from the original patch can be directly mapped onto the first part of the disk Φ_{map} *fig. 4.2* . The remaining parts of the disk can then be filled with cloned state variable values Φ_{cloned} of the first part, those cloned values must also correct for their orientation, as described in *chap. 2.3.1.4*. The approach for the handling of this correction is described in *chap. 4.2.2*.

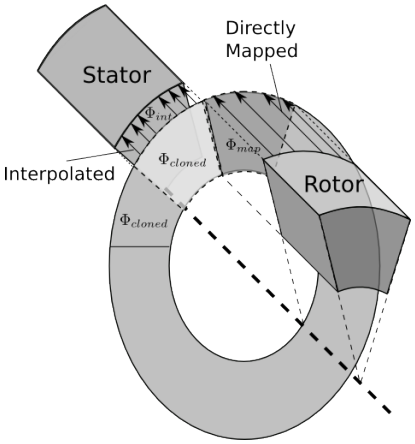


Figure 4.2: Showing the principal of the interpolation disk.

It can be seen in *fig. 4.2* that only the part of the auxiliary patch which lies collinear to the neighbour patch is used to build the interpolation weights. Because every rotation changes the aligned geometric parts the *GGIinterpolation* object must be updated after every physical time step.

4.2.1.2 Use of an interpolation sector

One idea would be to use only a part of a disk which covers the other patch to interpolate the mapped values and cloned values onto the shadow patch, see *fig. 4.3*. The cloned

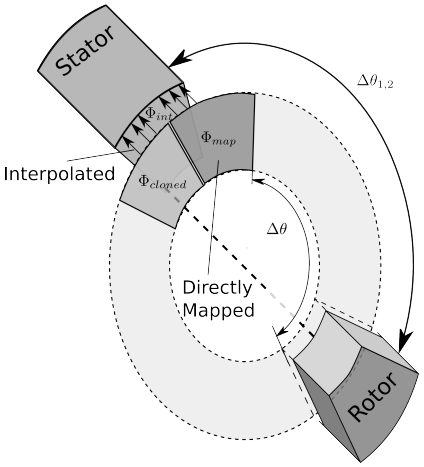


Figure 4.3: Showing the principal of the interpolation sector.

values are duplicated from the original field values and are mapped to a second patch-

part which has the identical structure as the original patch. In *fig. 4.3* the light gray part of the disk is not filled with values nor does it contain primitive geometry information. Only the two darker gray parts of the disk represent the auxiliary patch. This approach would enable the interface to reduce memory cost but an additional logic must be implemented to assure that the interpolation disk sector is always transformed to the right angular position. This rotation must be performed about the angle $\Delta\theta$ calculated with *eq. (4.3)*.

$$\Delta\theta = -\left\lfloor \frac{\Delta\theta_{1,2}}{\Delta\varphi_1} \right\rfloor \Delta\varphi_1 \tag{4.3}$$

Because the floor of the quotient between $\Delta\theta_{1,2}$, see *fig. 4.3* and the opening angle of the interface patch belonging to the first domain gives the number of virtual patches, which should be added to the original patch, to generate a sector, which reaches until the next overlapping patch part. This logic component must also assure that the vector-field components are transformed about the correct angle. Because this method seemed to be more difficult to implement, the interpolation disk method has been chosen.

4.2.1.3 Relative moving frame

Because it is possible with OpenFOAM to simulate a moving reference frame without moving the mesh, it must be possible to use the domain-scaling interface for this approach, too. For that alternative approach with non rotating computational domains, the rotation can be simulated by counter rotating the interface interpolation disks. To account for that, the current implementation made the interface position independent from the original patch position, by getting the position information indirectly from the time-step value. This is done by retrieving the actual time-step value from the data base and multiplying it with the corresponding constant angular frequency giving the angle about the interpolation disk has to be rotated. In this case the angular frequency must be carefully calculated for the upstream and downstream interface part. Imagine a non rotating downstream passage denoted in the following with 2 and a upstream passage, denoted with 1, rotating with the angular frequency ω_1. In that case, an observer in the relative frame of passage 1 will see the downstream passage rotating with the angular frequency $\omega_{1\to2}^{'}$ computed with *eq. (4.4)*.

$$\omega_{1\to2}^{'} = -\omega_1 \tag{4.4}$$

For counter rotating domains a relative angular frequency for every relative frame must be computed. Defining the relative angular frequency for passage 2, seen from an observer located in the upstream frame $\omega_{1\to2}^{'}$ with *eq. (4.5)*.

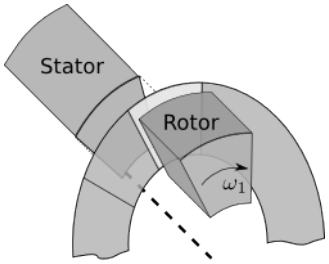
$$\omega_{1\to2}^{'} = \omega_2 - \omega_1 \tag{4.5}$$

With ω_1 the angular frequency of the upstream domain and ω_2 the angular frequency of the downstream domain. Substituting zero as value for ω_2 makes the result of *eq. (4.5)*

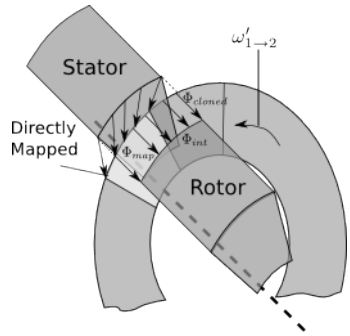
equal to *eq. (4.4)*. The relative angular frequency for the upstream domain observed from downstream domain can then be computed by *eq. (4.6)*.

$$\omega'_{2\to1} = \omega_1 - \omega_2 \tag{4.6}$$

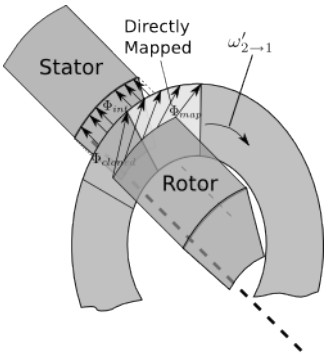
Because the relative position of the two domains must be the same as in case of a moving mesh, the angular frequencies for the interpolation disks are the same as the above calculated relative angular frequencies.



(a) moving domains



(b) interpolation from stator to rotor



(c) interpolation from rotor to stator

Figure 4.4: In case of non rotating domains the rotor and the stator interpolation disks must be rotated against each other with the resulting relative angular velocities $\omega'_{1\to2}$ and $\omega'_{2\to1}$, to obtain the same relative positions as for moving domains.

Therefore the interpolation disk for the interpolation from the rotor to the stator must be rotated with the angular frequency of $\omega'_{2\to1}$ shown in *fig. 4.4(c)*. Opposite to that the interpolation disk for the interpolation from the stator to the rotor must be rotated with $\omega'_{1\to2}$ shown in *fig. 4.4(b)*. In comparison to the positions of the domains shown in *fig. 4.4(a)* overlapping regions at both sides are the same as generated with the relative approach. Therefore this approach will also deliver a valid state variable exchange between two different non rotating domains. Until now the angular frequencies of the interpolation disks must be set directly by the user.

4.2.2 Geometry treatment

As mentioned in *sub. 4.2.1.1* the basic idea is to clone the field values onto a disk formed patch. To build this patch the first thing to be determined is the opening angle $\Delta\varphi$ of the original patch. By using this opening angle the number N_P of necessary patch parts can be determined by *eq. (4.7)*. This is done with *eq. (4.7)*. It is important to know that this equation must return an integer, otherwise an error must be thrown of the program.

$$N_P = \frac{360}{\Delta\varphi} \tag{4.7}$$

After that the original patch points are cloned by running through a for loop which sets the circumferential angle of the cloned point $\varphi_{cloned,i}$ which is calculated by multiplying the integer counter i times $\Delta\varphi$ *eq. (4.9)*. By running i from 0 to $N-1$ this will lead to a disk shaped patch consisting out of N patches which all share the same structure as the original patch.

$$\varphi_{cloned,i} = \varphi_{orig} + \Delta\varphi i \tag{4.8}$$

This procedure is visualized in *fig. 4.5*. It can be seen that one original point is shifted to the same relative position in the next sector part of the interpolation disk and that the radius R is equal to the original radius value R_{orig} of the corresponding original point.

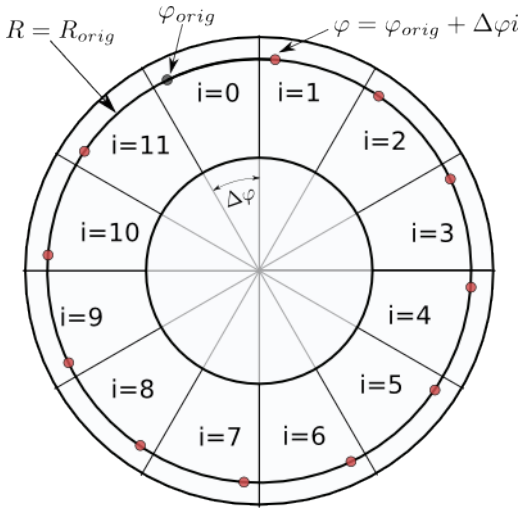
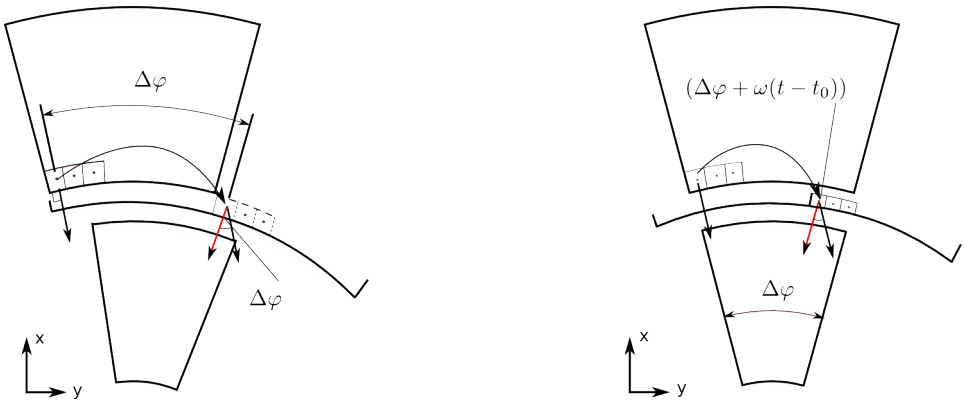


Figure 4.5: Example numbering for a $\Delta\varphi = 30$ which leads to a number of 12 patch parts which construct the whole disk.

After the points have been generated it must be noted that the faces have to be created in the same way as the original patch. The state variables are cloned in a similar way because the face labels of a disk part share the same consecutive order with the original patch, the field can be cloned again using a for loop. This is displayed in *eq. (4.9)* with $\Phi_{cloned,i}$ the state variable value at the ith face.



(a) moving domains (b) still standing domains

Figure 4.6: Direction distribution in case of relative frame approach and a moving domain approach.

$$\Phi_{cloned,i} = \Phi_{i-k} \tag{4.9}$$

With k being calculated by the floor of the counter variable i divided through N the number of values stored in the original field see *eq. (4.10)*. This corresponds to the number of already advised patch parts.

$$k = \left\lfloor \frac{i}{N} \right\rfloor N \tag{4.10}$$

For those cases in which the surface normal of the interface plane is pointing into the rotation axis direction it seems to be obvious that no necessity for a direction correction of all kind of tensor fields exists. But for those cases in which the interface plane normal is not collinear with the rotating axis a direction correction for fields with a rank greater than zero must be applied. For the case of a moving mesh a deviation of the direction is shown in *fig. 4.6(a)*, because the original vector is transferred about $\Delta\varphi$ during the cloning process this vector must be rotated about this angle to reach the correct direction marked red. For the relative case with a moving interpolation disk and a still standing calculation domain, the correction angle has to be increased about the traveled angle of the interpolation disk. This is shown in *fig. 4.6(b)*.

A more straight forward way would be to transform the field into cylindrical coordinates as it was done for the mixing-plane in *sub. 3.2.2*. Both methods were implemented and tested with a small test-case. After a certain angle was reached, a blew up of the solution was encountered. Therefore no radial or diagonal handling with the domain-scaling interface can be done until now. The presumable reason is a difficulty with transforming the segregated components into cylinder coordinates. The segregation takes place before the *updateInterfaceMatrix* function is called, see *fig. 2.5*. This function exchanges the segregated components from one domain to the other and therefore calls the interpolation strategy of the domain-scaling interface implementation. This limitation is caused by the program structure of OpenFOAM. From the current perspective only a new

written matrix handling class could solve that problem. This matrix class could give the domain-scaling interface implementation access to all the tensor-field components necessary to make the coordinate transformations possible.

4.3 Results

A domain-scaling interface has been implemented, which can handle even non rotating calculation domains and is also capable to be used as a frozen-rotor interface in case of a steady state calculation. Because it is not possible to run in parallel, it can only be used for small cases. Two test cases have been calculated. One for a rotating mesh and one with a non rotating mesh as a frozen-rotor test case.

4.3.1 Unsteady calculation with the domain-scaling interface

To show the effect of an unsteady simulation for a simple test case, a passage with two double wedge profiles was set up. The first domain was rotated by using the dynamic mesh approach delivered with OpenFOAM . To use the dynamic mesh for this test-case a topology changer called *turbFvMesh* was implemented. To observe a clear interface exchange an unsymmetrical velocity profile has been used, magenta 10 and red 20 $\frac{m}{s}$. This is shown in *fig. 4.7*, the measurements are in millimeters and the mesh had only 100.000 cells. The rotation speed of the rotor was chosen to be 2500 rpm, with moving wall boundary conditions applied to the rotor boundaries. The azimuthal boundaries were modeled as cyclic boundary. The setup of a domain-scaling interface is described in *sec. A.6*.

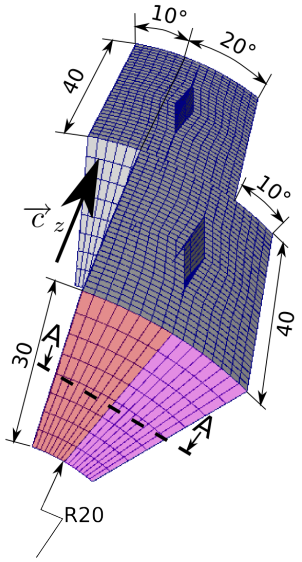
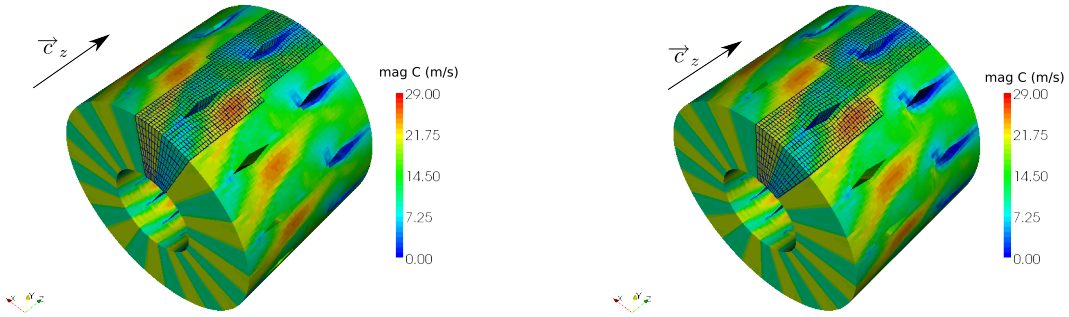


Figure 4.7: Case set up for the domain-scaling calculation, the thick dashed line marked with A represents the cutting plane.

Because it was the target to show the fluid transfer at the interface, the grid near the upper boundary was chosen to be very coarse. A finer grid would resolve the boundary layer and would not be able to show transfer gradients at the interface near the upper boundary.



(a) first observation time

(b) second observation time

Figure 4.8: (a) showing the solution after 100 time-steps, (b) showing the flow-field after 120 time-steps.

In *fig. 4.8(a)* a first position is shown, it can be seen that the values at the rotor-stator interface align pretty well. To show the full 360 degrees the single passages were copied and rotated to generate the full annulus. The same result in the alignment can be seen after an increase about several time steps, in *fig. 4.8(b)*. The important result which can be seen is that the transfer of the fluid can be considered as correct, because there are no distortions at the interface in the velocity field. To show that the variable transfer at the interface is performed correctly several cell layers next to the interface at the upstream and downstream side were extracted, see *fig. 4.9*.

The comparisons of the different cell layers are shown in *fig. 4.10*. All plots show a shift in x direction. This is caused by the shift in azimuthal position between the upstream and downstream domain *fig. 4.9*. But the absolute values and the course of the curves correspond very well in *fig. 4.10(b)* till *fig. 4.10(h)*. A higher difference in the absolute value and course can be observed for the cell layer located next to the upper boundary. This can be explained by the different treatment of the boundary conditions in the moving domain and the stationary domain. For the stationary domain the boundary values at the upper and lower boundary condition is equal to zero, but for the moving domain those boundary values are equal to the circumferential velocity. Because the circumferential velocity is proportional to the radius, a higher influence can be observed at the cells next to the upper boundary than to the lower boundary, see*fig. 4.10(h)*.

4.3.2 Alternative usage as Frozen-Rotor

Because the domain-scaling interface enables the user to rotate the interface only about a previous specified angle. This can be used as a frozen rotor approach for steady calculations. To setup a frozen rotor interface by using the domain-scaling implementation

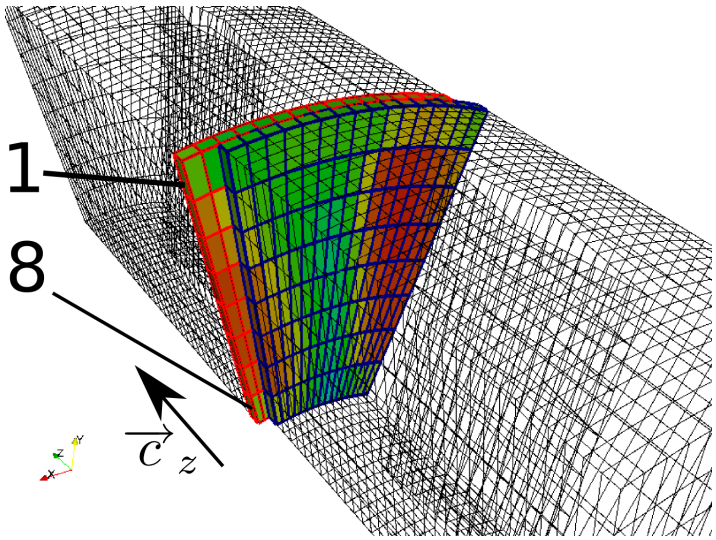
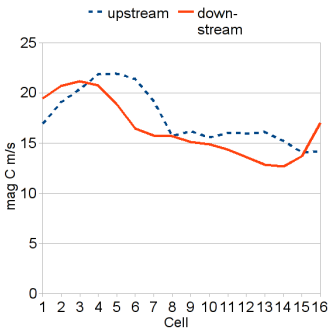


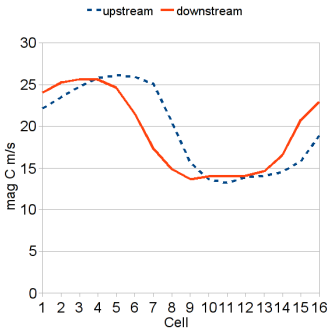
Figure 4.9: Showing the cells which were used to generate the plots in *fig. 4.10*. There were 8 cell layers extracted every cell extraction shared one radial level and the first cell extraction is at the upside.

the only thing which must be done is to add an entry called fixed angle to the boundary dictionary file and the domain-Scaling interface will stay fixed. As in the case of stationary domains one has to rotate both the stator domain interface and the rotor domain interface about the same angle. Alternatively a topology changer can be used to rotate the rotor domain physically about a specified amount, then the domain-scaling interface does not need a fixed angle parameter, only the angular velocity of both interface sides must be set to zero. The basic setup is also shown in *fig. 4.7*, the measurements are in millimeters. The magenta part has an inlet velocity of $10\frac{m}{s}$, the red part has an inlet velocity of $20\frac{m}{s}$. This unsymmetrical velocity profile has been chosen to see a clear velocity distribution at the interface to validate whether the flow values were transferred correctly or not. The incompressible solver *simpleFoam* has been used to calculate the flow-field. To compare the obtained results with a commercial solver the same case was calculated with Numeca. For Numeca the laminar navier stokes equation solver has been used, utilizing the incompressible air model. To compare the results a cut was made, this cut is represented by a thick dashed line marked with A in *fig. 4.8*.

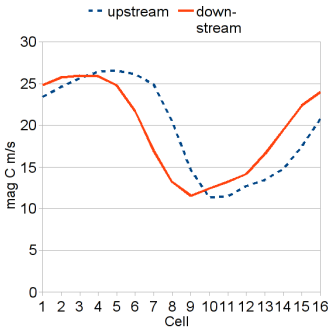
The OpenFOAM results in *fig. 4.11(a)* are, beside a lower velocity at the double wedge profile, comparable to the Numeca results *fig. 4.11(b)*. Another difference, which may have caused the higher velocity for Numeca is the low velocity for the first cells near the diamond profile. But the main focus lies on the interface which shows for the newly implemented domain-scaling interface for OpenFOAM a good agreement with the velocity distribution delivered by the Numeca calculation. Therefore it can be stated that the newly implemented domain-scaling interface can simulate a frozen-rotor approach. It can be used to easily evaluate a frozen-rotor calculation at every angular position by rotating only the interface. Therefore it also gives a main advantage about those implementations, which rely on the topological relation of the mesh.



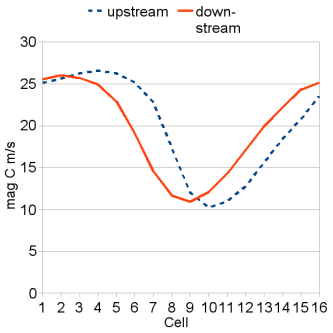
(a) 1. Cell Layer



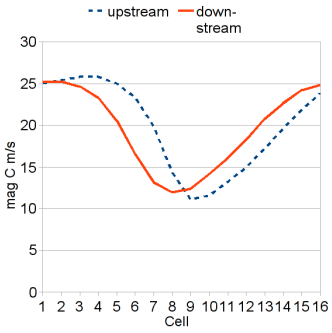
(b) 2. Cell Layer



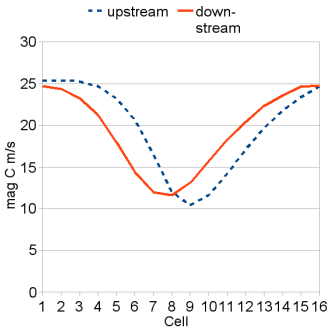
(c) 3. Cell Layer



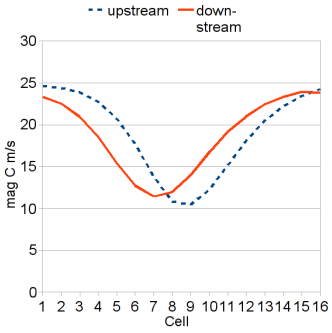
(d) 4. Cell Layer



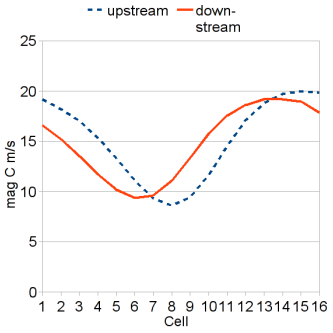
(e) 5. Cell Layer



(f) 6. Cell Layer

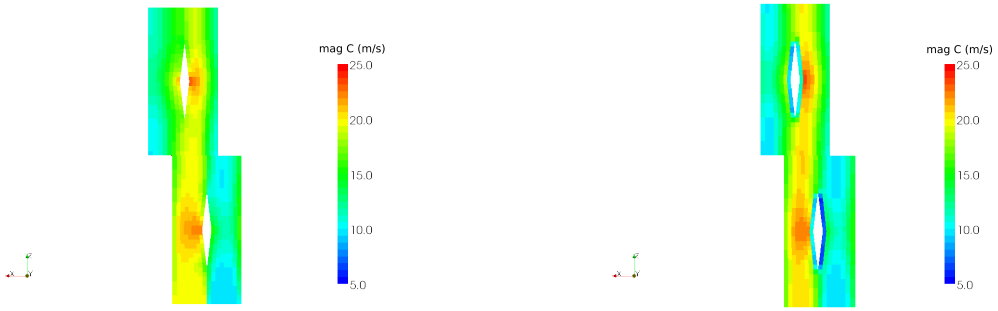


(g) 7. Cell Layer



(h) 8. Cell Layer

Figure 4.10: Showing the values within a cell layer next to the interface for the upstream and downstream part. Because the domains are rotated about an azimuthal angle, the curves are also shifted about a constant offset in x direction.



(a) OpenFOAM (b) Numeca

Figure 4.11: Cuts through the frozen-rotor case setup (a) shows the OpenFOAM result and (b) the result of Numeca

4.3.3 Future Improvements

The first problem which must be solved is to add the support for parallel calculations. This problem is just the same as for the mixing-plane and is described in detail at *sub. 3.3.2*. The other improvement is to generate a stable and fast unsteady solver which is capable to utilize the pseudo time-step method described in [14]. The other problem which is left to solve is the handling of radial and mixed flow cases by utilizing a coordinate transformation into the cylindrical coordinate system.

Chapter 5

Phase-Lag

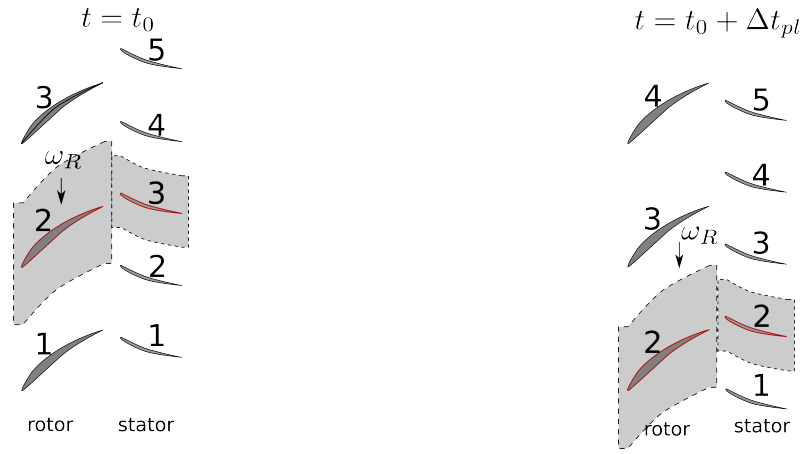
Real rotor-stator configurations most of the time do not allow the use of the domain-scaling approach for the calculation of passages with unequal pitches. Because the pitch relations do not fit the condition described with *eq. (4.2)*. To overcome that, a periodicity in time is assumed. The approach which respects this time periodicity is called phase-lag boundary condition. The name phase-lag or phase lagged boundary condition is used as a generalization for two different instantiations of a time shifted boundary condition. The phase-lag condition for which implementation ideas in OpenFOAM will be presented, is called direct storage method, see [27, pp. 12]. The other called chorochronic phase lag boundary condition, see [9, pp.69] describes the phase lag boundary condition by a fast Fourier transformation. The direct storage method was first mentioned by Erdos and Alzner see [7, pp. 46] and solves the problem of the phase lagged boundary condition by storing the calculated values of previous time-steps to use those as boundary condition for the current time-step. The phase-lag boundary condition is divided into two parts, the rotor stator interface and the azimuthal boundary conditions. How the time shift is exactly performed is illustrated in the following sections.

5.1 Theoretical idea

For rotor stator combinations with an unequal pitch the flow field has a combined spatial and temporal periodicity which can be expressed with *eq. (5.1)*, see [23, 27, pp.7, pp.7]. Compared to *eq. (4.1)* an additional time shift or phase lag is introduced Δt_{pl}.

$$\Phi_i(\theta_i, t) = \Phi(\theta_i + m\Delta\theta_i, t + \Delta t_{pl}) \tag{5.1}$$

This is illustrated in *fig. 5.1*, which shows the fact that the spatial relative position of the passage stator 3 to passage rotor 2 at initial time t_0, see *fig. 5.1(a)* is identical with the relative position of stator 2 to rotor 2 at a time incremented about the phase lagged time t_{pl}, see *fig. 5.1(b)*. Because the relative positions of the blade rows are identical, the flow-field can also be approximated as identical at those time steps.



(a) first time-step

(b) phase shifted position

Figure 5.1: According to [27, p.23] showing the identical relative positions at time t_0 and $t_0 + \Delta t_{pl}$.

5.1.1 Azimuthal boundary condition

The basic idea is to divide the whole domain into several discrete parts to have a time grid to which discrete relative positions can be mapped enabling the correct storing. The number of those parts is calculated by multiplying the number of the first cascade blades N_1 with the number of blades N_2 in the second cascade according to [7, p.99]. By introducing a refinement factor K to adopt for finer meshes, or to respect the courant number criteria by decreasing the time-step, *eq. (5.2)* delivers the number of sub parts.

$$N_{360} = N_1 N_2 K \tag{5.2}$$

Because both cascades can have an angular frequency *eq. (5.3)* is used to calculate the relative circumferential speed magnitude $\bar{\omega}_{rel}$. Then the time step for the phase-lag setup can be calculated by *eq. (5.4)*. Otherwise there is no way to retrieve the correct stored values.

$$\bar{\omega}_{rel} = |\omega_1 - \omega_2| \tag{5.3}$$

$$\Delta t = \frac{2\pi}{\bar{\omega}_{rel} N_{360}} \tag{5.4}$$

The circumferential angle change per time step can then be easily calculated with *eq. (5.5)*.

$$\Delta\varphi = \Delta t \bar{\omega}_{rel} = \frac{2\pi}{N_{360}} \tag{5.5}$$

The number of discrete azimuthal positions per blade passage in the first $i = 1$ or the second $i = 2$ cascade is calculated by *eq. (5.6)* .

$$N_{\varphi,i} = \frac{N_{360}}{N_{3-i}} = \begin{cases} \frac{N_{360}}{N_2} \text{ for } i = 1 \\ \frac{N_{360}}{N_1} \text{ for } i = 2 \end{cases} \tag{5.6}$$

Every passage marked with dashed lines in *fig. 5.2*, has two neighbour passages one is running ahead, denoted with F for forward running and the other is running behind, denoted with T for trailing running. To determine, which is running ahead and which is running behind the relative circumferential speed direction $\omega'_{i\rightarrow(3-i)}r_i$ is used. How the relative angular frequencies $\omega'_{i\rightarrow(3-i)}$ are calculated is explained in *sub. 4.2.1.3.* This can be seen in *fig. 5.2*, the pointer labeled with $\omega'_{1\rightarrow2}r_1$ shows the direction of the circumferential speed. Between one passage of the first cascade and the trailing passage is a time difference with the value $N_{\varphi,1}\Delta t$, see *fig. 5.2*. Because of that the state variable Φ at the azimuthal boundary condition is described with *eq. (5.7).*

$$\Phi_{a-b}(t) = \Phi_{c-d}(t - N_{\varphi,1}\Delta t) \tag{5.7}$$

The time difference between the ahead running passage is $(N_{360} - N_{\varphi,1})\Delta t$, therefore the state variable at the azimuthal boundary condition can be calculated for time t at the boundary c-d with *eq. (5.8).*

$$\Phi_{c-d}(t) = \Phi_{a-b}(t - (N_{360} - N_{\varphi,1})\Delta t) \tag{5.8}$$

The azimuthal boundary condition of the passage in the second cascade at the boundary h-g can be calculated with *eq. (5.9)*, because it is again the trailing boundary of the blade.

$$\Phi_{h-g}(t) = \Phi_{e-f}(t - N_{\varphi,2}\Delta t) \tag{5.9}$$

The corresponding boundary values for the line e-f is calculated with *eq. (5.10)*, because it runs in front of the blade respecting the direction of the circumferential speed a greater amount back in time must be traveled.

$$\Phi_{h-g}(t) = \Phi_{e-f}(t - (N_{360} - N_{\varphi,2}\Delta t) \tag{5.10}$$

The term $N_{360} - N_{\varphi,1}$ is a very big integer number, therefore the need for storing of relatively much values is founded. To reduce the amount of values need to be stored, this term can be simplified and it will be graphically shown that the same relative position of the cascades will be reached with *eq. (5.12).* The new number of phase lagged time-steps for the first cascade of the trailing cascade $N_{T,1}$ is calculated by subtracting the original $N_{\varphi,2}$ multiplied with the off rounded fraction of $N_{\varphi,1}$ through $N_{\varphi,2}$. This off rounded fraction can be considered as a switch which determines how many $N_{\varphi,3-i}$ sector parts must be subtracted from $N_{\varphi,i}$.

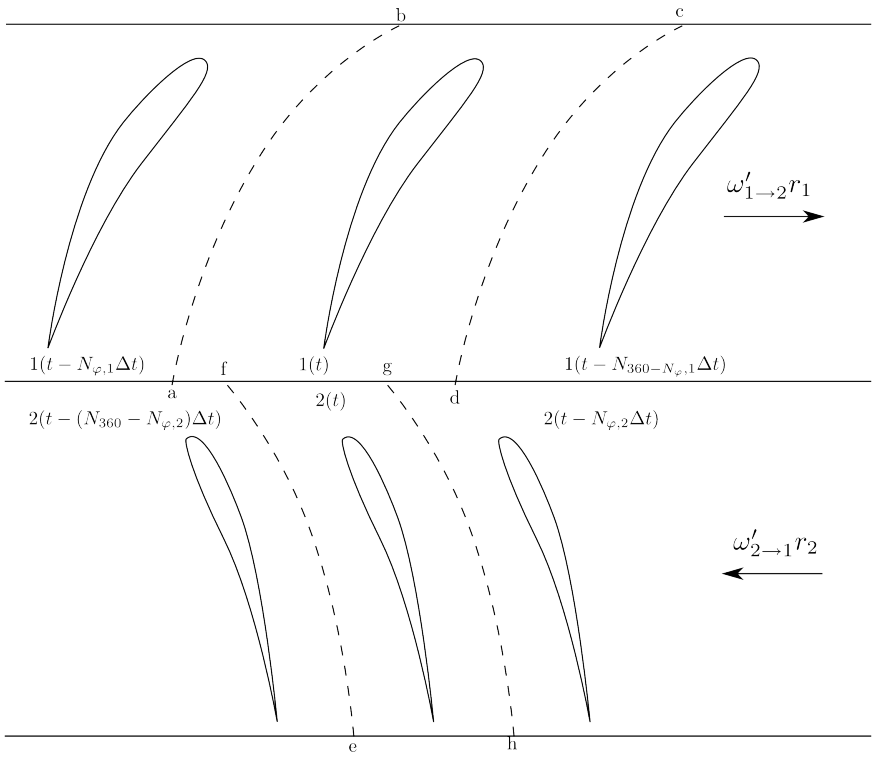


Figure 5.2: Azimuthal phase lagged boundary conditions with the time shifted values, according to [3].

$$N_{\varphi,i}\hat{=}N_{T,i} = N_{\varphi,i} - N_{\varphi,3-i}\underbrace{\left\lfloor \frac{N_{\varphi,i}}{N_{\varphi,3-i}} \right\rfloor}_{switch} \tag{5.11}$$

After $N_{T,1}$ has been calculated, the term $N_{360} - N_{\varphi,2}$ can be replaced by $N_{F,1}$ with *eq. (5.12)*, which corresponds to *eq. (5.11)*. The idea for the switch was given by [3].

$$(N_{360} - N_{\varphi,i})\hat{=}N_{F,i} = (N_{360} - N_{\varphi,i}) - N_{\varphi,3-i}\underbrace{\left\lfloor \frac{N_{360} - N_{\varphi,i}}{N_{\varphi,3-i}} \right\rfloor}_{switch} \tag{5.12}$$

To check the validity of the derived formulas, one can make a thought experiment by imagine a rotor ($i = 1$) which has 3 passages, a stator ($i = 2$) with 4 passages and a K value with one. Then the following relations for the azimuthal boundary condition can be calculated, see *tab. 5.1*.

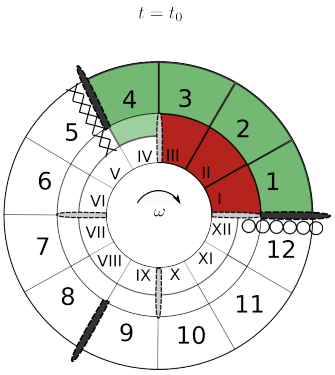
To proof the validity of the derived relations in *tab. 5.1*, one just constructs two rings, which are divided into the discrete parts. The first domain is shaded in green and the disk parts are numbered with arabic numerics and the second domain is shaded in red, the numbering is done with roman numerics see *fig. 5.3*.

In *fig. 5.3(a)* the position of the first domain and the second domain is shown for the current time t_0. In *fig. 5.3(b)* the relative position of the both sectors is shown for the

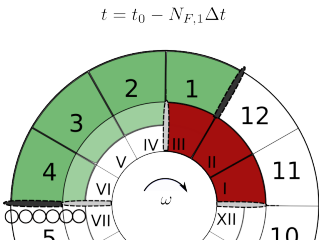
Name	value
N_{360}	12
$N_{\varphi,1}$	$\frac{12}{3} = 4$
$N_{\varphi,2}$	$\frac{12}{4} = 3$
$N_{T,1}$	$4 - 3\left\lfloor \frac{4}{3} \right\rfloor = 1$
$N_{F,1}$	$(12-4) - 3\left\lfloor \frac{12-4}{3} \right\rfloor = 8 - 6 = 2$
$N_{T,2}$	$3 - 4\left\lfloor \frac{3}{4} \right\rfloor = 3$
$N_{F,2}$	$(12-3) - 4\left\lfloor \frac{12-3}{4} \right\rfloor = 9 - 8 = 1$

Table 5.1: Showing the phase lag example values

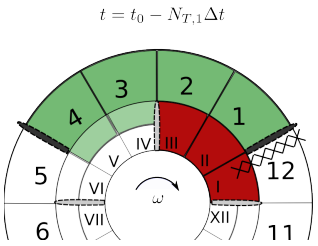
time step which is used to retrieve the state variables for the boundary condition at the forward running azimuthal boundary of the rotating domain. The circles in *fig. 5.3(b)* mark the side from which the values are extracted and used as boundary condition at the side marked with circles at the actual time step in *fig. 5.3(a)*. that the sectors IV to VI of the stator are aligned with the sectors 2 till 4, which corresponds to the relative position of the sectors X till XII to 10 till 12 in *fig. 5.3(a)*. A similar relation can be observed for the trailing running boundary of the first domain. In *fig. 5.3(c)* it can be seen that the sector I is aligned with sector 12 at the forward running boundary, which



(a) current time



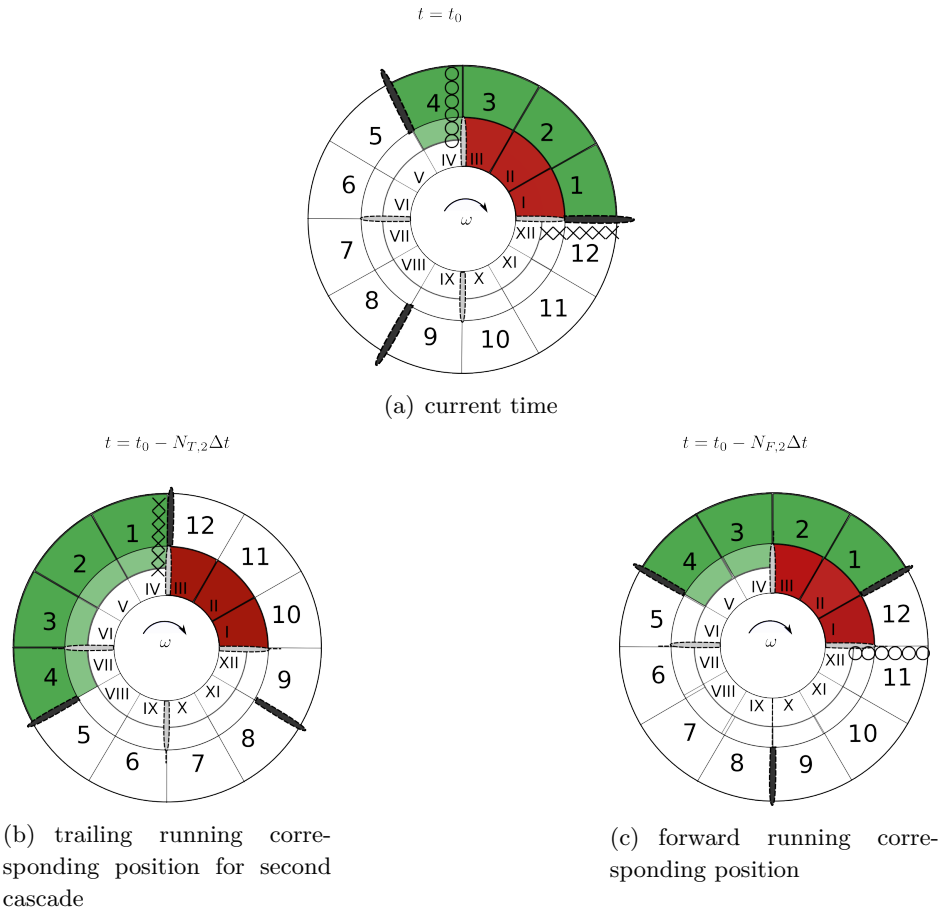
(b) forward running corresponding position for first cascade



(c) trailing running corresponding position for first cascade

Figure 5.3: Showing the example phase lag azimuthal boundary positions for the first passage.

again corresponds to the situation observed at the trailing boundary in *fig. 5.3(a)* for the sectors IV and 4. Therefore the values are extracted from the side marked with crosses in *fig. 5.3(c)* and are used as boundary condition at the side marked with crosses at the actual time step in *fig. 5.3(a)*.



(a) current time

(b) trailing running corresponding position for second cascade

(c) forward running corresponding position

Figure 5.4: Showing the example phase lag azimuthal boundary positions for the second passage.

For the second domain, the azimuthal boundary relations are displayed in *fig. 5.4*. The forward and trailing running boundaries are lying on the opposite side compared to the first domain, because of the relative angular frequency going into the opposite direction for that reference frame. For the trailing running side of this domain, *fig. 5.4(b)* delivers the position which is observed by shifting the time back about $N_{T,2}$ time steps. The side marked with crosses is used to extract the values for the use as a boundary condition at the equal marked side for the actual time-step displayed in *fig. 5.3(a)*. It can be seen that sector I till III align now with sector 10 till 12 in *fig. 5.4(b)*, which corresponds to the position of sector X till 12 and 10 till 12 in *fig. 5.3(a)*. For the forward running boundary *fig. 5.4(c)* shows that sector I and 12 correspond to the position of sector IV and 4 in *fig. 5.3(a)* for the current time step. The values for the forward running boundary are extracted from the side marked with circles in *fig. 5.4(c)* and are used as boundary conditions in *fig. 5.3(a)* at the similar marked side. It was shown that the relative positions at the time shifted boundaries agree with the relative positions at the

actual time-step. Therefore the introduced method to calculate the time shift can be considered as proofed and valid.

5.1.2 Rotor-Stator interface

For the rotor-stator interface a similar approach can be derived from the azimuthal boundary relations. The only difference is, that the interface between the two domains has an overlapping and a non overlapping region, because of the different pitch angles. Therefore the state variables from the non-overlapping region must be replaced by the time lagged state variables. In case of the first domain with three blades, green part, of the disks in *fig. 5.3(a)* no sector is in the non overlapping region of the neighbour disk red in *fig. 5.3(a)*. Therefore all the values extracted from the interface at domain one, are directly exchanged to the corresponding overlapping sectors of domain two. But for the second domain not all sector parts of one passage find a overlapping region on the neighbour passage, see *fig. 5.3(a)* sector IV and 4, therefore the values of the non overlapping sectors must be shifted in time and transferred to the first domain. To reach a corresponding position domain two must be shifted about $N_{F,2}$, which is in case of the before mentioned thought experiment equal one. This leads to the position shown in *fig. 5.3(c)*. The sector I and 12 are aligned and have a similar position than the non overlapping region formed by sector IV and 4 at the actual time-step, shown in *fig. 5.3(a)*. Therefore the values extracted from sector 4 at time $t_0 - N_{F,2}\Delta t$, see *fig. 5.3(c)*, are used for the boundary condition in sector 4 at the actual time-step, see *fig. 5.3(a)*. To reach those values a logic must be implemented to handle the time shift management at the interface. Especially care must be taken to separate between time shifted values and directly exchanged values.

5.1.3 Initial conditions

For the phase-lag boundary conditions, the initial conditions must be used until enough time steps are calculated to be used for the time shifted boundary conditions. For the azimuthal boundary conditions, the best approximation is to use a standard cyclic boundary condition. But for the rotor-stator interface only the non overlapping sector must be filled with an approximated initial condition. Right now to use a kind of a cyclic spatial boundary condition as described in *chap. 4* seems to be the most applicable way.

5.2 Idea for the implementation in OpenFOAM

Right now only an idea can be presented how one could implement a phase-lag boundary condition in OpenFOAM. During the current thesis no time was left to complete the started implementation of the phase-lag boundary condition. Major problems with the exchange strategy caused a high delay. In the following, two ways of handling the

rotor-stator interface are presented and one idea for the simpler azimuthal boundary condition.

5.2.1 Idea to implement the azimuthal boundary conditions

Because the initial condition of the azimuthal boundaries is generated by using the cyclic interface it was the idea to implement a cyclic boundary condition by extending the first implementation described in *sub. 2.3.1*. The only extension, which must be made is to use the time shifted values after enough time steps are available. The idea was to store the values in an appropriate way during the evaluate method call. During that call, it must be proofed that the added field is the value produced by the last iteration step in the current time step. Because this was not applicable during the evaluate method call, the value of the last time step is added to the list. Because the evaluate method is called before the other iteration steps, it can be assured that during the *updateMatrixInterface* method the old time step values are available. But a main problem raised, which made it impossible to store the values as a member variable. Therefore the values must be stored on the hard disk by using the *IOField* function. To know which time values have been stored in the file, an additional list which contains the corresponding time-step values has to be stored on the hard disk. It must be also assured that the stored file is not erased by killing the run. This can only be assured by introducing a backup file which is written before the original file is written.

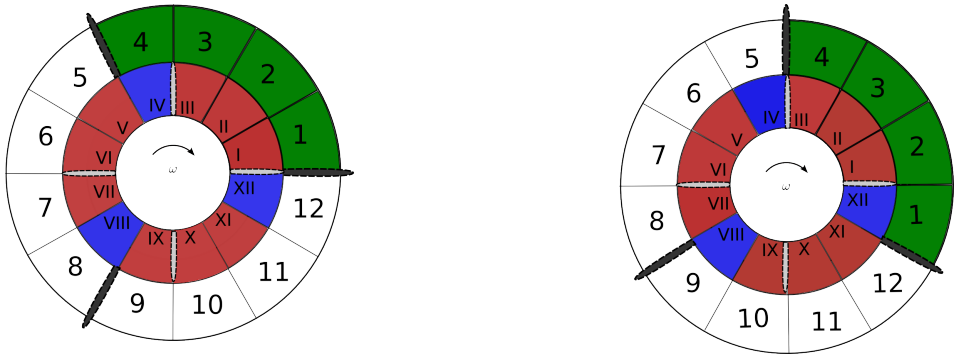
5.2.2 Idea to implement the rotor-stator interface

Two ideas for the rotor-stator interface implementation were found. One will use a complete disk, the other will use only a sector of a disk. The principal idea is similar to the domain-scaling approach, which used a disk to distribute the values on the rotor stator interface. But the phase lagged interface must also respect the time shift for the parts which do not have a direct overlap with the neighbour patch. Therefore specific parts of the disk will contain the phase-lagged values, which have to be used as a fixed value boundary condition.

5.2.2.1 Using of an interpolation disk

When looking at the parts of the interface, which are covered by current values, one can imagine that for the thought experiment only one part must be filled with the time lagged values. To compensate for the rotation and respecting that the flow field has a periodicity in space, the idea for the construction of an interpolation disk rises. This disk is shown in *fig. 5.5(a)*. The green part is representing the values of the first domain and the red part states the values of the second domain. The blue sector represents the time shifted part. This time shifted part is filled with values shifted about the $N_{F,2}$ value, because they are located on the forward running part of the domain two.

The time relation between the numbered parts of the domain 1 and domain 2 are listed in *tab. 5.2*. It can be seen how the value distribution would be for the whole disk.



(a) first position

(b) second position

Figure 5.5: Showing the approach of an interpolation disk with partitioned sectors.

After the first time has been incremented about one time step the disk is rotated around one sector part. This is shown in *fig. 5.5(b)*, it can be seen that the time shifted part in sector XII of the second domain will be advised by interpolation to the sector 1 of the first domain. Again it has to be noted that this is the time shifted value of the first sector, which leads to a comparable situation as one time step before, where the sector 1 was also aligned with sector I.

5.2.2.2 Geometry treatment for the interpolation disk

To use an interpolation disk the original patch must be split up into several sectors. This can be done by a similar procedure as described in *sub. 3.2.2*. First the opening angle of one sector is calculated by *eq. (5.5)*. After this is known the splitting can

domain 1	values for domain 2	domain 2	values for domain 1
1(t)		I(t)	
2(t)		II(t)	
3(t)		III(t)	
4(t)	1(t)	IV(t)	$I(t - N_{F,2}\Delta t)$
5(t)	2(t)	V(t)	I(t)
6(t)	3(t)	VI(t)	II(t)
7(t)	1(t)	VII(t)	III(t)
8(t)	2(t)	VIII(t)	$I(t - N_{F,2}\Delta t)$
9(t)	3(t)	IX(t)	I(t)
10(t)	1(t)	X(t)	II(t)
11(t)	2(t)	XI(t)	III(t)
12(t)	3(t)	XII(t)	$I(t - N_{F,2}\Delta t)$

Table 5.2: Showing the distribution of the values on the rotor stator interface.

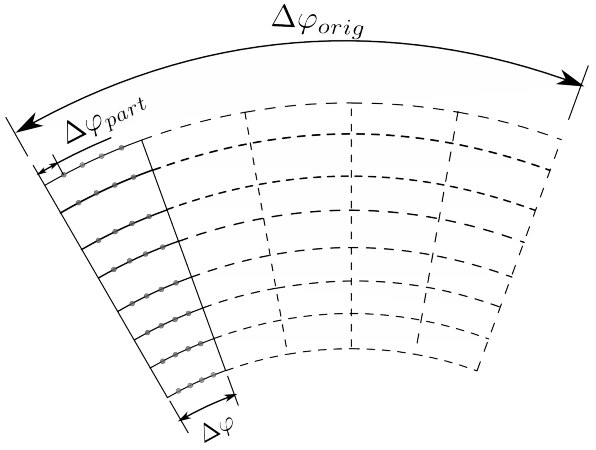


Figure 5.6: Showing the geometric relations for the building of the partitioned virtual patch.

be started. The next question, which rises, is how many faces this part must contain. Because the patch will be build in a similar way as for the mixing-plane, which means the side points of the original patch deliver the radial values for the newly generated partitioned patch, the only thing which must be determined are the points lying on the maximum radius values. This is done by scaling the total number of points $N_{R_{max}}$ lying on the original patch R_{max} line through the fraction between the $\Delta\varphi$ calculated with *eq. (5.5)* and the opening angle of the original patch, see *fig. 5.13*, which returns the number of necessary angle parts N_{part}.

$$N_{part} = \left\lceil \frac{\Delta\varphi}{\Delta\varphi_{orig}} \right\rceil N_{R_{max}} \tag{5.13}$$

The divisor $\Delta\varphi$ can be calculated with *eq. (5.14)*.

$$\Delta\varphi_{orig} = \frac{2\pi}{N_i} \tag{5.14}$$

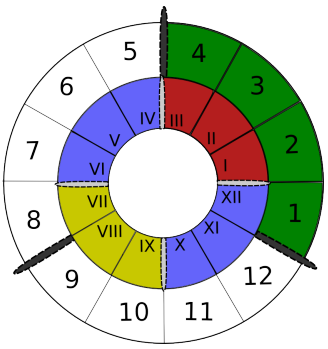
The angular distance between two consecutive points $\Delta\varphi_{part}$ on one line with constant radius is calculated with *eq. (5.15)* by dividing the passage angle $\Delta\varphi$ through the number of parts N_{part}. Those geometric relations are also shown in *fig. 5.2.2.2.*

$$\Delta\varphi_{part} = \frac{\Delta\varphi}{N_{part}} \tag{5.15}$$

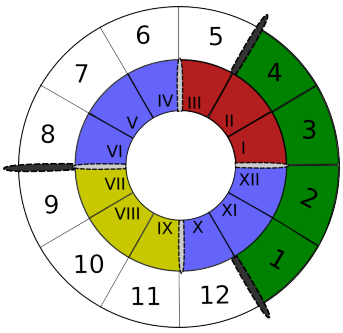
After those points have been generated, they are used to build the faces of a sector part, in the same way as in *sub. 3.2.2.* The first part of the patch is now built, the remaining patch elements are generated by transforming the first with a tensor rotation about the $\Delta\varphi$. During that operation the labeling of the first patch part faces are stored to identify the relative positions of the patch elements. This component of the phase lag interface has been implemented.

5.2.2.3 Using of an interpolation sector

By using an interpolation sector, the two disadvantages are that an additional logic must be implemented, which generates the correct attitude corresponding to the current time step. The second disadvantage is, that a whole passage sector must be stored. But from a different point of view this could be seen as an advantage for that method. Because it is not so complicated to generate the interpolation sector geometry primitives. No splitting of the sector must be performed. As seen in *fig. 5.7* the red sector of the stator which is denoted with roman numerics has a direct overlap with the 3 sectors of the rotor, for the fourth the values one time-step before are used for the boundary nodes. After the rotor has turned around one time-step, only two parts of the rotor are covered directly by the original values of the stator. But the stator sector III in *fig. 5.7(b)* is covered directly with the values of sector 5 which corresponds to the sector 1. But the sector 1 on the rotor is not covered with the values of the current time step of sector III. But assuming temporal periodicity this has maybe only a small effect on the final converged solution, which includes no periodicity assumption.



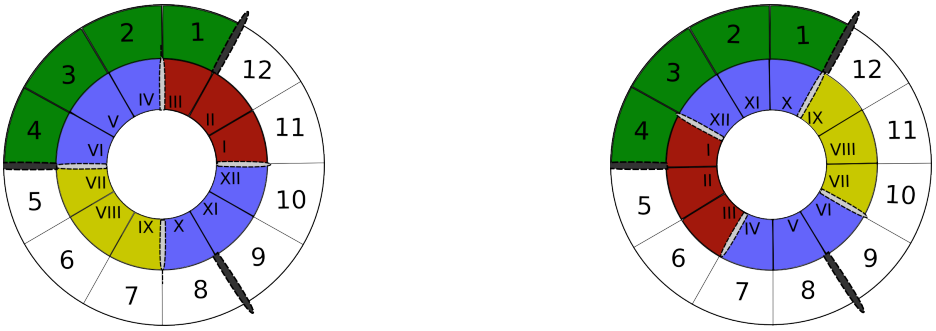
(a) position corresponding to first time-step



(b) position corresponding to consecutive time-step

Figure 5.7: Showing the interpolation sector strategy, blue sectors are filled with time shifted values. The yellow sector does not exist physically.

After enough time steps are calculated, the rotor will reach a position where it is covered by only one sector of the original values as shown in *fig. 5.8(a)*. The next time step would rotate the rotor into a position where it has no coverage with the original values anymore. To avoid that situation the stator disk has to be rotated, to get into the position shown in *fig. 5.8(b)*. The physical or original stator patch does not need to be rotated because the values of the rotor are distributed over the whole rotor interpolation disk. Therefore the interpolation from the rotor interpolation disk will always interpolate the correct values on the stator patch.



(a) end of interpolation sector

(b) jump to the beginning of interpolation sector

Figure 5.8: Showing how the sector must be rotated after the end has been reached to generate correct phase lagged values.

5.2.2.4 Geometry treatment for the interpolation sector

Compared to the interpolation disk geometry treatment the geometry generation for the interpolation sector is very easy. Because the original patch must just be duplicated in a similar way as described in *sub. 4.2.2*. The only difference is, that only two cloned patches are generated. One by adding the $\Delta\varphi$ and the other by subtracting this angle from the original azimuthal point coordinates φ_{orig}. This is shown in *fig. 5.9*.

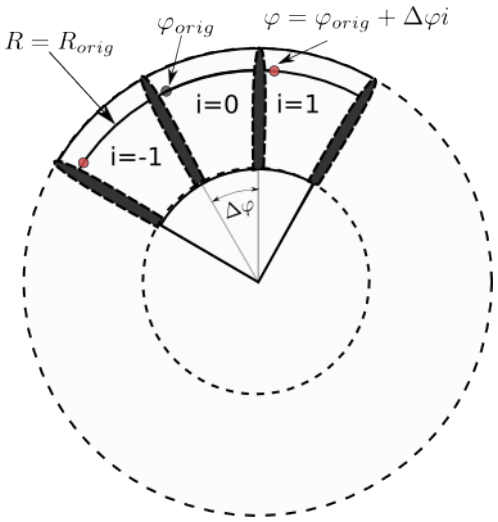


Figure 5.9: Showing how the sector parts are copied. i is an integer number which can be either 1, 0 or -1, this leads to the generation of points which share the same radius levels but different azimuthal coordinates.

5.3 Results

Right now two things have been implemented. The geometry handling for the rotor-stator interface, with the correct mapping of field values to sector parts. The basic

handling for the azimuthal boundary condition. Unfortunately the first test with the azimuthal boundary condition showed that the member variables cannot be stored during run-time, because the patch field class is built in every solution step by utilizing a blank constructor. Therefore the next task is to implement the correct handling of the storing of field variables. But there is still a high doubt that this method can lead to success, because a test with a cyclic interface, which only transferred constant values between the coupled regions, resulted in a solution blew up. Therefore the strategy is maybe completely wrong and must be replaced by a fixed value/fixed gradient boundary condition approach for the non connected sector parts, which seems right now absolutely difficult to implement.

Chapter 6

Conclusion

Before this diploma thesis there was no basic way to simulate rotor stator interactions with the OpenFOAM CFD toolbox by not modeling the full 360 degree cascades. After an intensive time of literature and source code research a good fundament for further developments was developed. Because the two implemented rotor-stator interfaces rely on the *GGIinterpolation* algorithm, it is essential to improve this algorithm, to make it more stable and more versatile in the handling of unstructured and not aligned meshes. Until now both implemented rotor-stator interfaces cannot be used for parallel calculations.

The implemented mixing-plane interface can be set up in an easy and familiar way. It delivers the ability to solve rotor-stator interactions for steady flow fields. The current implementation can deal with basic turbo-machinery geometries. To overcome that, an improvement strategy to handle all surfaces of revolution has been delivered. The mixing-plane is capable to couple the rotor and stator domain by respecting the conservativeness of the flow-field. The comparison between commercial products showed that the solution by OpenFOAM is comparable to the solutions of the commercial products. But during testing, the need for a special suited steady solver, which is able to handle supersonic and transonic flow-fields respecting the needs of the mixing plane implementation has been raised.

The domain-scaling interface enables the OpenFOAM users to simulate unsteady flow fields for rotor-stator interactions, by respecting the spatial periodicity at an interface between two passages, which have the same pitch. To improve the calculation speed, a new solver, especially suited for the calculation of unsteady flows within rotating reference frames, should be implemented in the future. Alternatively this implementation of a domain-scaling interface can also be used as a frozen-rotor which enables OpenFOAM users to vary the angular position of the rotors and stators in an easy way.

For the phase-lag interface, a different picture must be painted. During the implementation it showed that the developed idea could right now not be implemented mainly because the mixing between fixed and coupled boundary conditions was not possible. Therefore the complete idea has been illustrated in this thesis to spread the gained knowledge to other members of the OpenFOAM user community.

After this thesis has been finished, it is now possible to calculate passages located in two different reference frames by the use of OpenFOAM CFD tools library. Those two rotor stator interfaces make the OpenFOAM CFD tools library one of the first open source projects able to simulate cases specially suited for the simulation of turbo machinery. But the current implementation is just a good starting for future improvements and extensions for the OpenFOAM turbo-machinery calculation capabilities.

Appendix A

Appendix

A.1 Useful tensor calculus

Because OpenFOAM is based on tensor calculus a short overview of the used tensor calculus is given.

A.1.1 Cylindrical coordinates

The transformation from a vector noted in cartesian coordinates into cylindrical coordinates is derived in the following. The covariant base tensors $\mathscr{g}$ of the cylindrical coordinate system are given in the following, see [17, pp.140].

$$\mathscr{g}_1 = \overrightarrow{e}_1 cos\theta + e_2 sin\theta \tag{A.1}$$

$$\mathscr{g}_2 = -\overrightarrow{e}_1 r\, sin\theta + e_2 r\, cos\theta \tag{A.2}$$

$$\mathscr{g}_3 = \overrightarrow{e}_3 \tag{A.3}$$

This leads to the covariant metric tensor displayed in *eq. (A.4)*.

$$\mathscr{g}_{ij} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.4}$$

This defines the physical base of the tensor coordinates. By following [12, pp.221], those can be calculated by dividing the base tensor components through the corresponding metric tensor component as done in *eq. (A.5)* .

$$\mathscr{y}_i^* = \frac{\mathscr{y}_i}{\sqrt{\mathscr{y}_{ij}}} \tag{A.5}$$

This leads to following physical base coordinates for the cylindrical coordinate system *eq. (A.8)* .

$$\mathscr{y}_1^* = \vec{e}_1 cos\theta + e_2 sin\theta \tag{A.6}$$

$$\mathscr{y}_2^* = -\vec{e}_1 sin\theta + e_2 cos\theta \tag{A.7}$$

$$\mathscr{y}_3^* = \vec{e}_3 \tag{A.8}$$

Therefore the transformation of a vector in rectangular coordinates to one vector in cylindrical coordinates is just a matrix multiplication as shown in *eq. (A.9)*

$$v_{CCS} = \underbrace{\begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathscr{Z}} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{A.9}$$

The transformation from the cylindrical coordinate system to the rectangular coordinate system is done by inverting the matrix, because it is a symmetric matrix this can be done by transposing the matrix.

$$v_{xyz} = \underbrace{\begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathscr{Z}^T} \begin{bmatrix} v_r \\ v_\theta \\ v_z \end{bmatrix} \tag{A.10}$$

A.1.2 Rodriguez tensor

In most of the time, rotation about an arbitrary axis must be performed. Either for generating rotated point coordinates or for transforming vector-fields from one coordinate system to another. Therefore the common known Euler transformation matrices are not very useful, but in [1, pp.29] the Rodriguez's tensor $\mathscr{R}$ is derived.

$$\mathscr{R} = \mathscr{I} + \mathscr{w} \cdot sin(\theta) + \mathscr{w}^2 \cdot cos(\theta) \tag{A.11}$$

With *eq. (A.11)* a rotation tensor can be easily calculated by knowing the identity tensor $\mathscr{I}$, the axis vector $\mathscr{w}$ and the angle argument θ for the rotation. In full length this leads to *eq. (A.12)*

$$\mathcal{R} = \begin{bmatrix} c + \not{w}_x^2 \hat{c} & \not{w}_x \not{w}_y \hat{c} - \not{w}_z s & \not{w}_y s + \not{w}_x \not{w}_z \hat{c} \\ \not{w}_z s + \not{w}_x \not{w}_y \hat{c} & c + \not{w}_y^2 \hat{c} & -\not{w}_x s + \not{w}_y \not{w}_z \hat{c} \\ -\not{w}_y s + \not{w}_x \not{w}_z \hat{c} & \not{w}_x s + \not{w}_y \not{w}_z \hat{c} & c + \not{w}_z^2 \hat{c} \end{bmatrix} \tag{A.12}$$

With c the cosinus of θ, s the sinus of θ and $\hat{c} = 1 - cos\theta$.

A.2 Short overview GGIinterpolation

To interpolate between two non conformal patches, it is necessary to use an area weighted interpolation. This is done by the *GGIinterpolation*, which was introduced with OpenFOAM 1.3-dev. There are two patches, one is named slave and the other master, this is dependent from the interpolation direction. If the original values are originated from patch A and should be interpolated to patch B, then patch A is called master and patch B slave.

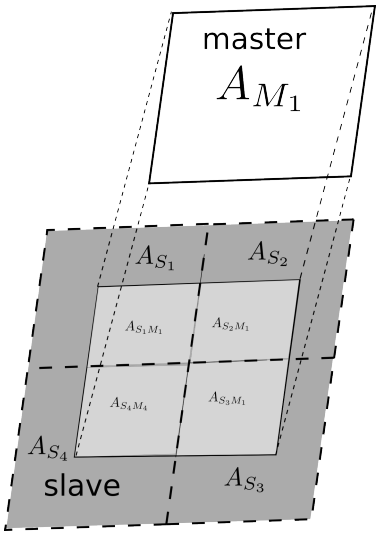


Figure A.1: Face intersection generating the weights for the master and shadow.

In *fig. A.1* the original face size is denoted as A_{M_1}, those original faces are intersected by the slave faces marked with light gray. In *fig. A.1* only one slave face is shown for simplicity. According to [2], by building the quotient between the slave face area, which covers a part of the master area, and the whole master area, the factor for the interpolation weight is calculated, see *eq. (A.13)*.

$$w_{M_i,S_j} = \frac{A_{S_j,M_i}}{A_{M_i}} \tag{A.13}$$

The conservativeness is automatically assured by this procedure because the sum of all i weighting factors belonging to one slave face full-fill *eq. (A.14)*.

$$\sum_j w_{M_j,S_i} = 1.0 \tag{A.14}$$

After those factors have been calculated they can be used to transfer the state variable values from the master Φ_M to the ith slave $\Phi_{S,i}$ see *eq. (A.15)*.

$$\Phi_{S_i} = \sum_j w_{M_j,S_i} \Phi_{M_i} \tag{A.15}$$

The reverse procedure must be applied for the interpolation from the slave to the master. In this case the master to slave weights cannot be inverted but must be calculated new, by again dividing the slave area covered by the master face through the slave face area magnitude.

$$w_{S_j,M_i} = \frac{A_{M_i,S_j}}{A_{S_j}} \tag{A.16}$$

Again the conservativeness is assured by *eq. (A.17)*.

$$\sum_n w_{S_j,M_i} = 1.0 \tag{A.17}$$

The interpolated values at the master face can then be calculated with *eq. (A.18)*, this is also shown in *fig. A.2*.

$$\Phi_{M,i} = \sum_n w_{S_j,M_i} \Phi_{S_j} \tag{A.18}$$

During work with the *GGIinterpolation* algorithm several things of interest have been discovered. It is important to know, that the current native implementation of the *GGIinterpolation* algorithm is sensitive to two important parameters. The intersection direction and the intersection algorithm. For the intersection direction two different directions can be chosen. One is called *VISIBLE* and the other *FULL_RAY*, both have not shown differences in the result. Therefore they can be chosen totally freely.

Not only the intersection direction can be specified but also the intersection algorithm can be selected for the *GGIinterpolation*. Right now only two algorithms were available, the *CONTACT_SPHERE* and the *VECTOR* algorithm. Both use a different approach, the *VECTOR* algorithm assumes that, by walking into the direction, given by the master face normal, an intersection with the opposite face can be made. The *CONTACT_SPHERE* algorithm generates a sphere around the point which should be checked for a hit. Therefore the *CONTACT_SPHERE* algorithm is very suitable for the case of radial or mixed flow machinery. The *VECTOR* algorithm has not delivered a successful interpolation for the radial and mixed flow machines, but was much more stable in case of an axial machinery than the *CONTACT_SPHERE* approach. With the *CONTACT_SPHERE* algorithm some axial cases caused the solver to end with a floating point exception.

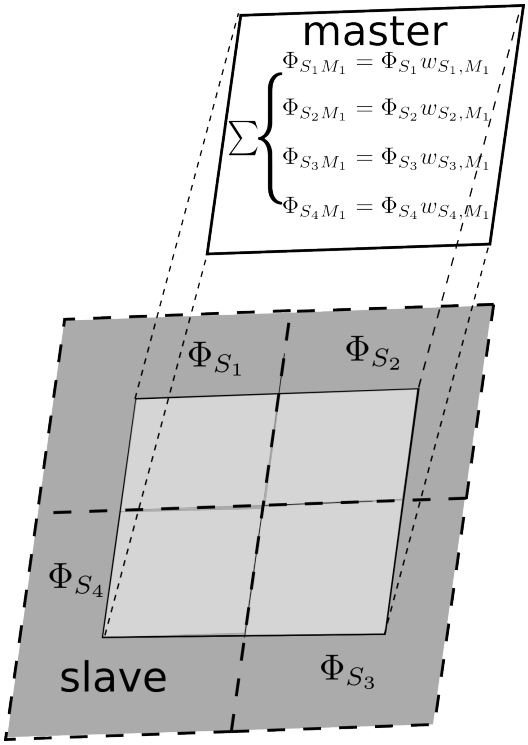


Figure A.2: Summations of the state variable multiplied with the face weights, generating the interpolated state variable.

A.3 Rotating reference frames in OpenFOAM

There are three possibilities to simulate the rotating frames in OpenFOAM.

A.3.1 Absolute velocity in inertial reference frame

This approach uses the momentum and continuity equations noted in the inertial reference frame, solving for the absolute velocity $\overrightarrow{c}$. The continuity equation is displayed in *eq. (A.19)*.

$$\frac{\partial \rho}{\partial t} + \nabla \bullet (\rho \overrightarrow{c}) = 0 \tag{A.19}$$

The momentum equation is noted by *eq. (A.20)* with $\overrightarrow{k}$ the body forces and the stress tensor noted as $\tilde{\mathcal{J}}$.

$$\frac{\partial}{\partial t}(\rho \overrightarrow{c}) + \nabla \bullet (\rho \overrightarrow{c}\,\overrightarrow{c}) = \nabla \tilde{\mathcal{J}} + \rho \overrightarrow{k} \tag{A.20}$$

The mesh is modified by topological changers, which generate the rotation of the different domains. The effects on the fluid caused by the rotating blades are inserted by unsteady boundary conditions. For turbo-machinery the most important movement is the rotation around one axis, especially for this thesis a new topology-changer was

written to rotate the mesh around one axis. It is also important to choose the right boundary conditions for the rotating blades. For those the *movingWallVelocity* boundary field type must be used otherwise no rotational velocity component, resulting from the rotating blades is impressed into the flow-field. For a detailed description of the moving mesh solving procedures, see [8, pp.669].

A.3.2 Relative velocity in relative reference frame

In that model the momentum equation is noted in the relative reference frame and it is solved for the relative velocity $\overrightarrow{w}$. The relative velocity is derived from the absolute velocity $\overrightarrow{c}$ by subtracting the rotational velocity component $\overrightarrow{u}$. This rotational velocity component is calculated by the cross product between the angular frequency vector $\overrightarrow{\omega}$ and the position vector in the relative reference frame $\overrightarrow{r}'$, see *eq. (A.21)*.

$$\overrightarrow{w} = \overrightarrow{c} - \underbrace{\overrightarrow{\omega} \times \overrightarrow{r}'}_{\overrightarrow{u}} \tag{A.21}$$

The continuity equation is displayed in *eq. (A.22)*.

$$\frac{\partial \rho}{\partial t} + \nabla \bullet (\rho \overrightarrow{w}) = 0 \tag{A.22}$$

Because the momentum equation is solved in the relative reference frame, a fluid particle will encounter fictitious forces caused by the rotation of the relative reference frame. Therefore the coriolis and centrifugal forces are added in the momentum equation. But also the unsteady euler part must be added, which is the tangential acceleration caused by the temporal change of the angular frequency. This equation is displayed in *eq. (A.23)*, $\overrightarrow{w}$ is the relative velocity, compare [28, pp.45].

$$\frac{\partial}{\partial t}'(\rho \overrightarrow{w}) + \nabla \bullet (\rho \overrightarrow{w}\,\overrightarrow{w}) + \underbrace{2\rho(\overrightarrow{\omega} \times \overrightarrow{w})}_{Coriolis} + \underbrace{\rho(\overrightarrow{\omega} \times (\overrightarrow{\omega} \times \overrightarrow{r}'))}_{Centrifugal} + \underbrace{\rho \frac{\partial \overrightarrow{w}}{\partial t} \times \overrightarrow{r}'}_{Euler} = \nabla \overrightarrow{\mathcal{J}} + \rho \overrightarrow{k} \tag{A.23}$$

This approach is used by the single reference frame model (SRF) released in the OpenFOAM 1.5 version. This calculates all the state variables in one reference frame and therefore applies the coriolis force and centrifugal force as a source term, because it solves the momentum equation noted in the relative frame. This solver can only handle one reference frame. Therefore no rotor-stator interaction can be simulated with this method right now. But it should be possible to extend this solver, to be able to simulate two different $\overrightarrow{\omega}$ field regions.

A.3.3 Absolute velocity in relative reference frame

Another possibility is to solve the momentum equation noted in the relative frame for the absolute velocity. The continuity equation *eq. (A.22)* is applied, because the relative

reference frame is used. The momentum equation *eq. (A.23)* can be rearranged to *eq. (A.24)*. This is done by substituting $\overrightarrow{w}$ in *eq. (A.23)* with the right hand side of *eq. (A.21)*, see [13, p.259].

$$\frac{\partial}{\partial t}^{'}(\rho\overrightarrow{c}) + \nabla \bullet (\rho\overrightarrow{w}\overrightarrow{c}) + \rho(\overrightarrow{\omega} \times \overrightarrow{c}) = \nabla\overrightarrow{\mathcal{J}} + \rho\overrightarrow{k} \tag{A.24}$$

In OpenFOAM the multiple reference frame (MRF) uses this approach. A major advantage of the MRF is the possibility to split the calculation domain in different reference frames by using different cell zones. Therefore the MRF approach can be used to simulate rotor stator interactions. Normally this approach is used to simulate so called mixed out flow problems. But it could be used for the mixing-plane approach by using the *MRFZones* within a solver derived from the *icoFoam* solver.

A.4 How to implement a boundary condition

To implement a basic boundary condition in OpenFOAM a new class must be derived from the template base class *patchField*. To understand how such a *patchField* derivation is working it is necessary to have a deep knowledge about template classes and template function in general see [29, pp.349]. A template class can take a template parameter for example most of the fields will be either a scalar field e.g pressure or a vector field e.g velocity. Instantiation of this template classes is done in the fvPatchFields.C declaration file. In general situations it is totally sufficient to implement only a derivation of the *fvPatchField* class. To implement boundary conditions which need to manipulate and use the primitive geometry information a single derivation of the following classes is needed.

- *fvPatchField* handles all the value calculation and adds the appropriate coefficients into the matrix. This is also what is represented in the boundary field dictionary.
- *fvPatch* is responsible for the geometric handling needed for the finite volume method. Therefore it is used to calculate the delta coefficients and the weights.
- *polyPatch* is responsible for the geometry handling such as faces areas and points of the underlying boundary patch.
- *pointPatch* is responsible for exporting the field values to the post processing utilities.

If a coupled boundary field is needed the list above must be modified and the *lduInterface* and *lduInterfaceField* class must be implemented otherwise the coupled boundary field is not recognized by the solver and an error will be thrown. Nothing was modified in that class except the type name information, which identifies the interface during runtime.

A.4.1 Class structure of the implemented interfaces

fig. A.4.1 is used to show how the before mentioned classes have been used to derive the implemented mixing-plane interface. It can be seen, that the lowest gray shaded class is the derivation from the *fvPatchField*, which is the major control class and therefore contains relations to all the other classes. The actual averaging as described in *sub. 3.2.2* is performed in the *primitiveMixingPlanePatch* which is also responsible for handling the points and face generation. This class is derived from the *primitiveRotationalPatch*, this class is responsible for all the basic sorting and determination of the machine type. Therefore the *primitiveRotationalPatch* can be considered as a basic turbo-machinery specific patch capable to deal with rotation symmetric surfaces.

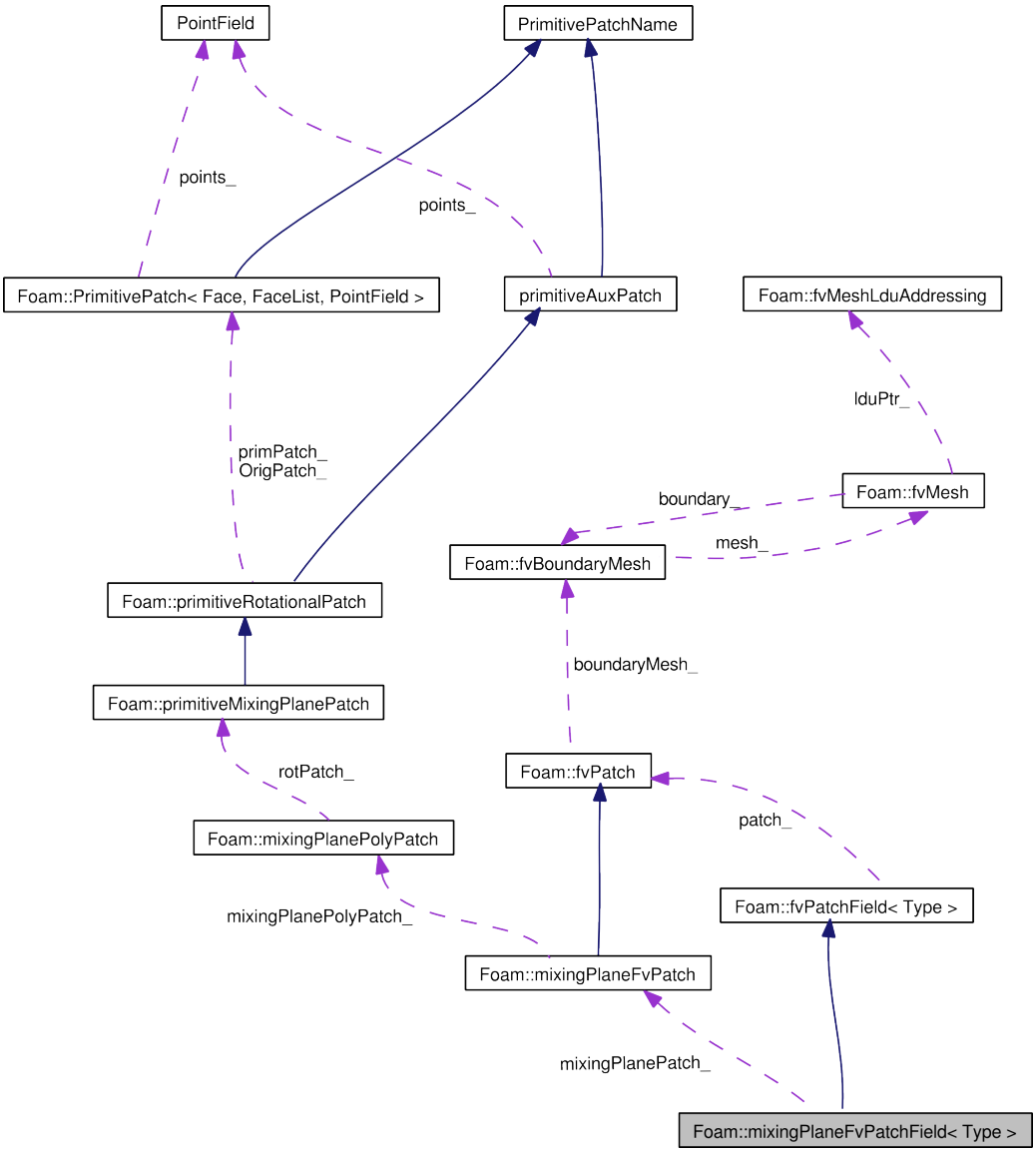


Figure A.3: Collaboration diagram of the *mixingPlaneFvPatchField* showing the relations between the several objects, which are used to assemble the mixing-plane.

For the domain-scaling interface implementation the collaboration diagram is shown in *fig. A.4*. It can be seen again this interface has a *primitiveDomainScalingPatch*, which is derived from the *primitiveRotationalPatch*. In that class the cloning of values and the interpolation to the neighbour patches is performed.

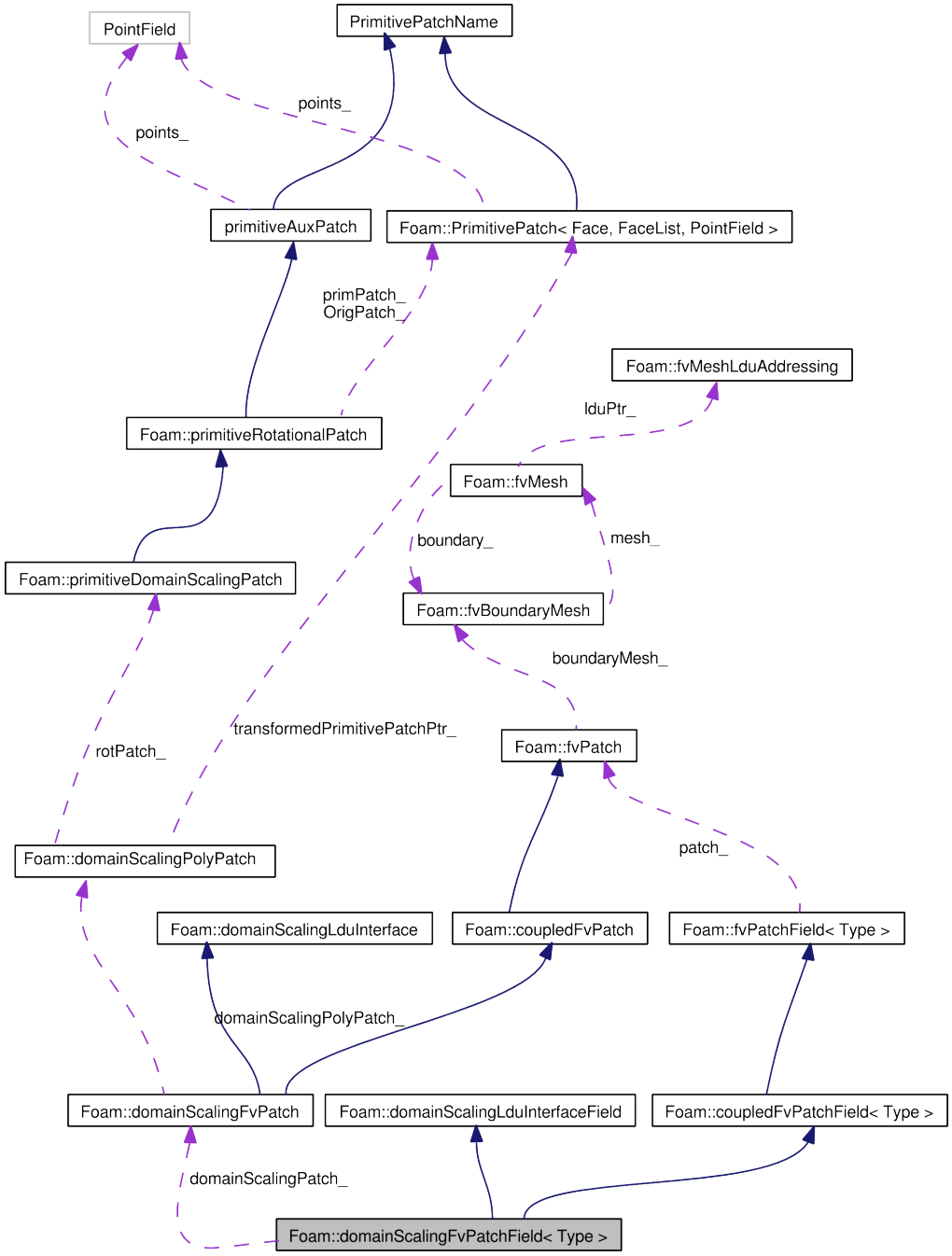


Figure A.4: Collaboration diagram for the domain-scaling *fvPatchField* showing the additional classes, which are necessary to assemble the domain-scaling interface approach.

A.5 Setting up a mixing-plane case

To set up a mixing plane case the first thing which must be done is to write a correct *blockMeshDict* file. The *blockMeshDict* file format is the native OpenFOAM file format for describing a structured mesh. For that it has to be noted that the two domains which are connected by the mixing-plane interface must not be connected in the *blockMeshDict* file.

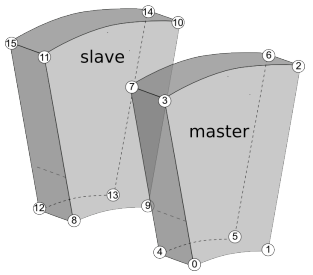


Figure A.5: Example case geometry for setting up a domain-scaling interface, the numbers should be associated with vertex numbers in the *blockMeshDict* file necessary for defining the blocks.

This is shown in *fig. A.5*, one can see that the two regions master and slave do not share the same vertexes, which makes OpenFOAM to recognize them as not connected to each other. To connect those with the implemented mixing-plane approach one just writes into the *blockMeshDict* two new patches one for the master of the mixing-plane and the other for the slave patch as shown in listing *lst. A.1*.

```
mixingPlane rotor-Interface // the first is denoting the type of the
    patch the second the arbitrary chosen name
{
    ( 5 6 7 4)
}
mixingPlane statorInterface
{
    (8 11 10 9)
}
```

Listing A.1: "part of boundary file in the *polyMesh* folder, showing the entries defining the mixing-plane interface"

After those entries have been added to the *blockMeshDict* one can generate the mesh with the blockMesh command line tool. After that has been executed the user must edit the newly generated boundary file residing in the constant/polymesh directory. The mixing-plane patch has written to that file template parameters, which must be replaced by the user. This is shown in listing A.2.

After the geometric parameters such as the origin and direction of the cylindrical coordinate system has been set, the field values must be set. In listing *lst. A.3* the upstream

```
rotorInterface // the name of the patch
{       type mixingPlane; // nothing to change here
        nFaces 800; // nothing to change here
        startFace 393600;  // nothing to change here
        shadowPatch statorInterface; // instead of master the name of
            the shadow
        // must be written in this case it is statorInterface
        hierarchy slave;  // decide which patch should be master and
            which should be slave
        // in this case master
        origin (0 0 0); // defines the origin of the cylindrical
            coordinate system
        axis (0 0 1); // defines the axis of the cylindrical coordinate
             system
        direction (0 1 0);  // defines the direction to which defines
            the zero value of the angle
}
statorInterface
{
        type mixingPlane;
        nFaces 800;
        startFace 394400;
        shadowPatch master;  // in this case the corresponding shadow
        // patch is the rotorInterface
        hierarchy slave; // in this case it can be left slave
        origin (0 0 0);
        axis (0 0 1);
        direction (0 1 0);
}
```

Listing A.2: "boundary file entry for the mixing-plane"

boundary is identified by setting the switch for zeroGradient to true and to make the mixing-plane use mass weighted averaging for that property the extensive switch is set to true.

To tell the downstream boundary that it should use the averaged values of the upstream the zeroGradient switch must be set to false. After all that has been performed for every boundary field file the mixing-plane case can be started.

A.6 Setting up a domain-scaling case

Similar to the mixing-plane case the computational domain is split up into two separate domains shown in *fig. A.5* one domain contains the upstream interface patch and the other the downstream interface patch. The *blockMeshDict* file entry is shown in *lst. A.4*.

After that the blockMesh command can be run on the new case, this will generate a boundary file which contains the additional information for the domain-scaling interfaces

```
rotorInterface
{
    type            mixingPlaneFvPatchField;
    zeroGradient    true;
    extensive       true;
}

statorInterface
{
    type            mixingPlaneFvPatchField;
    zeroGradient    false;
    extensive       true;
}
```

Listing A.3: "boundary conditions file entry for the mixing-plane."

filled with template values, see *lst. A.5*. This additional information is essential for the domain-scaling interface during runtime, therefore the user must specify this data after the boundary file has been generated.

In *lst. A.5* an additional parameter startAngle can be set, to set the interface interpolation disk to a specified starting angle, which makes it possible to use it for a frozen-rotor approach. In comparison to the mixing-plane set up for the boundary field it can be seen in *lst. A.6* that only the type information must be specified for the domain-scaling interface. After those steps have been performed the domain-scaling case is ready to run.

```
domainScaling master
(
   ( 5 6 7 4)
)
domainScaling slave
(
   (8 11 10 9)
)
```

Listing A.4: "blockMesh domainScaling patches"

```
rotorInterface
{       type domainScaling; // nothing to change here
        nFaces 800; // nothing to change here
        startFace 393600;  // nothing to change here
        shadowPatch master; // instead of master the name of the shadow
        // must be written. In this case it is statorInterface
        hierarchy slave;  // decide which patch should be master and
            which should be slave
        // in this case master
        origin (0 0 0); // defines the origin of the cylindrical
            coordinate system
        axis (0 0 1); // defines the axis of the cylindrical coordinate
             system
        direction (0 1 0);  // defines the direction to which the zero
            value of the angle is defined
        omega 0; // omega defines the angular velocity in rpm must be
            adopted appropriately
        startAngle 0; // optional parameter to set the start position
            of
        // virtual interpolation patch
}
statorInterface
{
        type domainScaling;
        nFaces 800;
        startFace 394400;
        shadowPatch master;  // in this case the corresponding shadow
        // patch is the rotorInterface
        hierarchy slave; // in this case it can be left slave
        origin (0 0 0);
        axis (0 0 1);
        direction (0 1 0);
        omega 0;
        startAngle 0;
}
```

Listing A.5: "part of boundary file in the polyMesh folder, showing the entries defining the domain-scaling interface."

```
rotorInterface
{
    type            domainScaling;
}

statorInterface
{
     type            domainScaling;
}
```

Listing A.6: "boundary conditions file entry for the domain-scaling interface"

Bibliography

[1] Yavuz Basar and Dieter Weichert. *Nonlinear Continuum Mechanics*. Springer Verlag, 2000.

[2] Martin Beaudoin and Hrvoje Jasak. Adaptation of the general grid interface (ggi) for turbomachinery simulations with openfoam. page 32, 2008.

[3] Oliver Borm. personal communication.

[4] Willy J.G. Bräunling. *Flugzeugtriebwerke*. Springer Verlag, 2004.

[5] Lars Davidson. A pressure correction method for unstructured meshes with arbitrary control volumes. *International Journal For Numerical Methods In Fluids*, 22:265–281, 1996.

[6] J.D. Denton and U.K Singh. Time marching methods for turbomachinery flow calculation. *Von Karman Institute For Fluid Dynamics*, 7, 1979.

[7] John I. Erdos and Egar Alzner. Computation of unsteady transonic flows through rotating and stationary cascades i - method of analysis. *Nasa Contract Report 2900*, 2900:all, 1977.

[8] Charbel Farhat, Philippe Geuzaine, and Celine Grandmont. The discrete geometric conservation law and the nonlinear stability of ale schemes for the solution of flow problems on moving grids. *Journal of Computational Physics*, 174:669–694, 2001.

[9] G. A. Gerolymos and V. Chapin. Generalized expression of chorochronic periodicity in turbomachinery blade-row interaction. *Rech. Aerosp.*, 5:96–73, 1991.

[10] Jasak H. http://openfoam.cfd-online.com/forum/messages/1/3660.html?1186517028, January 2007.

[11] Jasak H., Weller H.G., and Nordin N. In-cylinder cfd simulation using a c++ object-oriented toolkit. SAE Technical Paper 2004-01-0110, 2004.

[12] Schade Heinz and Klaus Neemann. *Tensoranalysis*. de Gruyter, 2006.

[13] D.G. Holmes and S.S. Tong. A three-dimensional euler solver for turbomachinery blade rows. *Transactions of the ASME*, 107:258–264, 1985.

[14] A. Jameson. Aiaa (91-1596) time dependent calculations using multigrid, with applications to unsteady flows past airfoils and wings. In *AIAA 10th Computational Fluid Dynamics Conference*. AIAA, June 1991.

[15] H. Jasak and H.G. Weller. Finite volume methodology for contact problems of linear elastic solids. In *Proceedings of 3rd International Conference of Croatian Society of Mechanics, Cavtat/Dubrovnik*, pages 253–260, September 2000.

[16] Dongjoo Kim and Haecheon Choi. A second-order time-accurate finite volume method for unsteady incompressible flow on hybrid unstructured grids. *Journal of Computational Physics*, 162:411–428, March 200.

[17] Eberhard Klingbeil. *Tensorrechnung für Ingenieure*. Bibliographisches Institut Mannheim/Wien/Zürich, 1966.

[18] Luca Mangani. *Development and Validation of an Object Oriented CFD Solver for Heat Transfer and Combustion Modeling in Turbomachinery Applications*. PhD thesis, Universita degli Studi di Firenze, 2008.

[19] Professor Dr. Kurt Meyberg and Dr. Peter Vachenauer. *Höhere Mathematik I*, volume 6. Springer Verlag, 2001.

[20] Maryse Page and Martin Beaudoin. Adapting openfoam for turbomachinery applications. In *OpenFOAM Workshop*, June 2007.

[21] Patankar and Suhas V. *Numerical Heat Transfer And Fluid Flow*. Series in Computational Methods in Mechanics and Thermal Sciences. Hemisphere Publishing Corporation McGraw-Hill Book Company, 1980.

[22] Dan Pilone and Neil Pitman. *UML 2.0 In A Nutshell*. O'Reilly, 1 edition, 2006.

[23] Sergey V. Rusanov A.V. Yershow. Numerical method for calculation of 3d viscous turbomachine flow taking into account stator / rotor unsteady interaction. In *The 4th Colloq. Process Simulation, ed. A. Jokilaakso, 11-13 June 1997, Espoo, Finland,*, pages 179–197, 1997.

[24] J. Y. Murthy S. R. Mathur. A pressure-based method for unstructured meshes. *Numerical Heat Transfer, Part B Fundamentals*, 31:195–215, 1997.

[25] E. Lorrain M. Swoboda S. Vilmin, Ch. Hirsch. Unsteady flow modeling across the rotor/stator interface using the nonlinear harmonic method. In *Proceedings of GT2006*, Barcelona, Spain, May 2006. ASME.

[26] Quarteroni Sacco Saleri. *Numerische Mathematik 1*. Springer Verlag, 2002.

[27] Mohamad Sleiman. *Simulation of 3-D Viscous Compressible Flow in Multistage Turbomachinery by Finite Element Methods*. PhD thesis, Concordia University Montreal, Quebec, Canada, April 1999.

[28] Joseph H. Spurk. *Strömungslehre*. Springer Verlag, 4 edition, 1996.

[29] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 2000.

[30] H K Versteeg and W Malalasekera. *An Introduction to Computational Fluid Dynamics The finite Volume Method*. Pearson Prentice Hall, 2007.

[31] David A. Wheeler. Sloccount. http://www.dwheeler.com/sloccount/, 09 2008.

[32] Dipl. Ing. Kai Uwe Markus Ziegler. *Experimentelle Untersuchung der Laufrad-Diffusor-Interaktion in einem Radialverdichter variabler Geometrie.* PhD thesis, Rheinisch Westfälische Technische Hochschule Aachen, July 2003. pages 52ff.

Eidesstaatliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe bzw. unerlaubte Hilfsmittel angefertigt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

München, 30.09.2008
Franz Blaim