SORBONNE UNIVERSITÉ

ONERA
THE FRENCH AEROSPACE LAB

MASTER OF ENGINEERING SCIENCES - 2^{nd} YEAR
SORBONNE UNIVERSITY - 2017/2022

Graduating internship at ONERA

for the validation of

Master degree in Engineering speciality Computational Mechanics / Cursus Master en Ingénierie

Mesh adaptation for CFD (Computational Fluid Dynamics) simulations of turbomachinery applications

Author:
Valentin DUVIVIER *+33 7 67 79 30 80*
valentin.duvivier@etu.sorbonne-universite.fr

Supervisor:
Sâm LANDIER, Research Engineer at ONERA *+33 6 58 19 76 70*
sam.landier@onera.fr

Résumé

Les turbomachines sont des structures complexes nécessitant une grande attention quant à l'adaptation du maillage. Ces dernières années, l'ONERA a développé une stratégie de maillage adaptée aux turbomachines, basée sur l'intersection et l'assemblage de maillages de manière conforme Landier (2017). Cette stratégie amène à un maillage conforme, crucial dans le contexte des turbomachines où la conservativité est requise Cali et al. (2001).

Cette stratégie manque cependant de flexibilité. En fait, le processus de maillage ne permet pas encore de raffiner sur une frontière périodique, pas sans casser cette périodicité. Pourtant, en turbomachinerie, il est courant de restreindre l'étude à un canal autour d'une aube en utilisant la propriété de périodicité.

L'objectif de ce stage est d'apporter cette flexibilité en assurant un bon processus de raffinement lors de chaque étape d'adaptation du maillage, en mettant l'accent sur la propriété de périodicité. Nous proposons le cas d'un canal entourant une aube de turbomachine, pour lequel nous voulons assurer une stricte conservativité, en fixant des conditions aux limites périodiques sur un maillage conforme.

Dans un premier temps, nous avons effectué des modifications, des implémentations et des débogages de codes python et C++ développés pour assurer la périodicité des simulations CFD de turbomachines. Ces codes ont ensuite été implémentés au côté de codes existants dans le logiciel Cassiopee afin de calculer numériquement des structures aérodynamiques et des cas de turbomachines complexes et de grande taille.

Enfin, nous avons évalué les développements implémentés à travers un cas de turbomachine bien référencé: la configuration CM2012, un canal de turbomachine traversé par un choc transversal.

Les résultats ont montré une grande consistence dans le cas de turbomachines industrielles, assurant une périodicité à chaque étape de la stratégie de maillage.

Mots-clés: adaptation de maillage, turbomachinerie, condition de périodicité, conservativité, Cassiopée

Abstract

Turbomachines are complex structures requiring great care when it comes to mesh adaptation. In recent years, ONERA has developed a suitable meshing strategy for turbomachinery, based on intersecting and assembling meshes in a conformal way Landier (2017). This strategy results in a conformal mesh, crucial in the context of turbomachines where conservativity is required Cali et al. (2001).

This strategy however lacks flexibility. In fact, the meshing process doesn't consider yet the possibility of refining over a periodic boundary, not without breaking this periodicity. Yet, in turbomachinery it's common to study a stage's channel only, and to then generalize by use of periodicity property.

My internship has for objective to bring this flexibility by ensuring good refinement process during every steps of mesh adaptation, with a focus on periodicity property. We propose the case of a channel surrounding a blade of turbomachine, for which we want to ensure strict conservativity, by setting periodic boundary conditions over a conformal mesh.

First, we did modifications, implementations and debugs of python and C++ codes developed to ensure periodicity for the CFD simulations of turbomachines. These codes were subsequently implemented along existing ones in Cassiopee software in order to numerically compute large and complex aerodynamic structures and turbomachinery cases.

Eventually, we assessed the developments committed through a well referenced turbomachinery case: the CM2012, a turbomachine channel crossed by a transversal choc.

The results showed great consistency in the industrial turbomachinery case, ensuring periodicity at every stages of the meshing strategy.

Keywords: mesh adaptation, turbomachinery, periodicity condition, conservativity, Cassiopee

Acknowledgments

My name is Valentin Duvivier and this report aims at detailing and exposing the work done during my graduating internship at ONERA. This internship took place at ONERA's DAAA laboratory, among DEFI team, in charge of high performance computing of aerodynamic structures.

This DEFI research team is led by Stéphanie Péron, research scientist in CFD, which I thanks for her warm welcome and for the attention she has granted to me throughout my time at ONERA.

During this internship, I have been under the supervision of M. Sâm Landier, researcher specialized in numerical geometry and meshing.

I hereby want to thank M. Sâm Landier for his constant help as a supervisor for this internship, on multiple points of this report including the report structure and some of the content inside it.

In addition, I would like to give a special thank to the researchers, master and Ph.D students from DEFI team, with which I have shared most of my days during this internship at the ONERA.

I thereby give them a special thank for these few months working together.

This internship is the conclusion of my master degree, and it will award me with the master title, speciality computational mechanics at Sorbonne University.

I as well take these few lines to acknowledge them in their coursework, among the greatest I have witnessed by exchanging with other students. I particularly want to stress the work done by the researchers/lecturers of mechanics, among which I want to mention Mrs. Hélène Dumontet, Mr. Corrado Maurini, Mrs. Aurélie Gensbittel and Mrs. Léa Boittin.

From a more personal aspect, I would like to give special thanks to the people thanks to who I am here today: my parents, Eminem, luck and myself.

Table of content

List of Figures

List of Tables

General introduction

Turbomachinery is at the core of aerodynamic structures, from propulsion to energy production systems Denton (1990). The study of turbomachines is therefore essential but it's made difficult by the inherent coupling fluid / structure that comes with it.

Several researches have been carried out to develop the matter, among which one can mention the study of dynamic flutter of blades in modal vibration Dowell et al. (1982), the fatigue life prediction of blades Fleeter et al. (1999), the topology optimization of axisymmetric turbine and compressor disks Wang et al. (2022) or as well the anisotropic mesh adaptation for high-fidelity CFD turbomachinery applications Alauzet et al. (2022).

ONERA, a french company running aerospace-oriented researches, addresses since 1950 studies of aeroelastic systems ROY (1959), A.Dugeai et al. (2018), as well as turbomachinery systems ONERA researchers (2022). ONERA is a leader in the aerospatial domain, developing France's scientific excellence at the service of both societal and industrial issues.

Nowadays, the aerodynamic researches at ONERA are conducted by the Department of Aerodynamics, Aeroelasticity and Acoustics (DAAA), which prepares technological responses to the aerodynamics challenges of tomorrow. In this sense, DAAA develops aerodynamic solvers for CFD and CAA simulations, to the benefit of the aeronautic field. DAAA as well conducts fundamental researches towards the understanding of physical phenomena (e.g. turbulence modelling, experimental research).

DAAA's axes of research combine fluids simulations as well as meshes generation for complex systems such as aircrafts, helicopters and turbomachines.

As part of their studies on mesh generation, researchers of DAAA have developed Cassiopee, a numerical tool merging pre- and post-processing steps of CFD simulations. These pre- and post-process steps include among else the generation of complex mesh, the overset-assembly of grids as well as post-processing functions.

The CFD solvers of ONERA then outsource the meshing process to Cassiopee tool, which automatizes most of the meshing process. The combination of Cassiopee tool / CFD solvers then take applications on aerodynamic systems, including turbomachinery configurations.

In light of this, DAAA has developed a meshing strategy to ease parametric studies of turbomachinery design, based on adapting, intersecting and assembling meshes in a conformal way Landier (2017). This strategy is used in adaptCells function of Cassiopee, in which polyhedral adaptation process is being handled.

On the one hand, this meshing strategy provides a conformal mesh which is crucial in the context of turbomachines where the conservativity is required. In fact, one major concern when simulating turbomachinery flows is to evaluate accurately the mass flow rates through a channel. Consequently, the numerical methods need to be as conservative as possible to avoid numerical losses. For that purpose, the mesh must be conformal to ensure that the flux balance at cell interfaces is conservative.

On the other hand, this meshing strategy does not include the possibility of refining over a boundary, not without breaking periodicity. Yet, computation of turbomachines is very cost-effective, asking one to study a stage's channel only, and to then generalize to the entire structure by use of periodicity property.

As a result of current assembly process, periodicity is not ensured, therefore it is not possible to reduce the analysis to this periodic channel anymore.

In this context, the ONERA ensued a master internship to improve this meshing strategy. In particular, my mission during this internship was to add a periodic-boundary-preserving capability to adaptCells function of Cassiopee in order to ensure conformity and so conservativity.

I have modified, implemented and done debug processes of python and C++ codes under adaptCells / Cassiopee, to ensure periodicity during the mesh-adaptation phase. I went from a conformal mesh made of a single blade surrounded by a channel and I implemented step by step the periodicity property throughout the meshing strategy. During this stage of the internship the point was to prevent every possible cases of wrong-doing during the assembly process, in order to ensure the robustness of periodicity codes *a priori*.

Eventually, I tested the developments committed in the case of a well referenced turbomachinery case: the CM2012 case from ONERA, a turbomachinery channel crossed by a transversal shock. This practical application was meant to test in real conditions the preservation of the periodic boundaries while adapting the mesh.

The results showed great consistency in the industrial turbomachinery case, ensuring periodicity at every stage of the meshing strategy. As a conclusion, we discuss possible missions for one to enhance robustness of application cases as well as their diversity.

Chapter 1

ONERA

In this chapter, we will take an insight on ONERA and its structure as a company.

Contents

1.1 History ONERA

ONERA, Office National d'Etudes et de Recherches Aérospatiales, is a pioneer in aerospace and defence. It has been created in 1946 with the objective to catch up technologically with France's then-allied forces Dubois (1966).

Among the world of sciences, ONERA has been working on great projects like the ONERA-M6 wing and the Concorde Schmitt and Charpin (1979), J. Carpentier and K. Dang-Tran (1997).

In addition to its own projects, the ONERA has several partnerships with recognized companies regarding: Defence Naval Group, ONERA (2021), Aeronautics Airbus, DLR, ONERA (2017), Airbus Defence & Space, ONERA (2018), High-precision imaging ULIS | Sofradir, ONERA (2016), etc.

The diversity of its application fields reinforce its aura and influence over the scientific world.

Here is a chronological frieze gathering the main events in ONERA history:

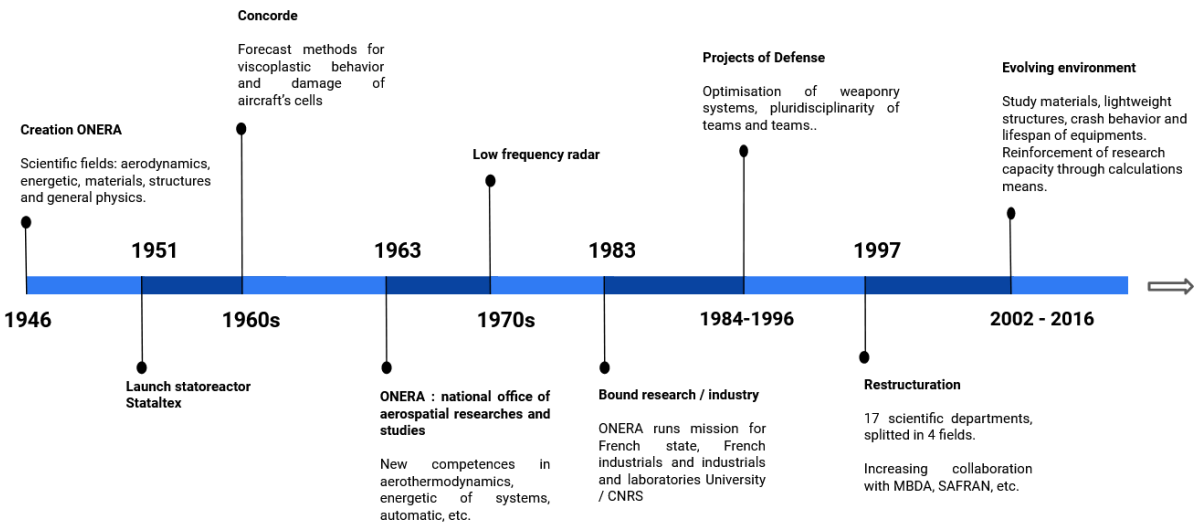


Figure 1.1: Chronological frieze of ONERA's history.

Now that we have some insight into ONERA's history, let's see its structure as a company.

1.2 Company's structure

The ONERA is a key player in the aerospatial field, depending directly from the Ministry of the Armed Forces (France). Thus, it benefits from a budget of around 237 million euros, among which half comes from company contracts.

ONERA employs over 2000 engineers and technicians workers. These people are spread over 8 working sites all over France, from Châtillon to Lille and more recently a cutting-edge research plant at Palaiseau.

These working sites combine a knowledge applied through 7 departments: DAAA, DEMR, DMAS, DMPE, DOTA, DPHY, DTIS. They study aerodynamic structures from materials and structures (DMAS) to electromagnetism and radar (DEMR) and aerodynamic, aeroelasticity, acoustic (DAAA).
The range of scientific fields tackled down is therefore very large.

Considering ONERA's structure, each of the mentioned department is splitted into research team, with each its objective. This paper is part of the researches of DAAA department.

1.3 DAAA Laboratory

The Department of Aerodynamics, Aeroelasticity and Acoustics (DAAA) is in place since 2017 with the joining of several scientific departments of ONERA, as part of a restructuring process. It develops and improves scientific tools and solvers from experimentation to numerical computation of aerodynamic systems.

This department of the ONERA is developing aerodynamics, aeroelastics and acoustics solutions and technologies to enhance the performances of aero-structures such as aircrafts, turbomachines, for societal, environmental and defence applications. The research projects concern a wide range of scientific skills, from thesis to post-doctorate and partnerships with private companies.

DAAA is composed of 14 research units, working on 11 major scientific themes. It represents over 200 people, based in Châtillon, Meudon and Lille.

Among the DAAA laboratory, I have been working in DEFI (Demonstration, Efficiency, Fiability, Interoperability of softwares) research unit, composed of around 12 people, with coming and going master's students as well as Ph.D students.
This branch is specialized in the numerical development of pre- and post- processing of complex geometries, as well as meshes generation and adaptation.

In a more personal experience, I have been included in the teamwork through team meetings which helped put my work in a more general context. It has well helped keeping up with what others did and let them know what I was working on.

Chapter 2

State of the art

In this chapter, we will go through the fields and research papers covering the present report's subject. We will derive a state of the art for the literature and then the counterpart for ONERA.

The point will eventually be to develop

- what is done at ONERA with respect to the literature,
- where our research stands with respect to the literature.

Contents

2.1 Literature

2.1.1 CFD

We refer to as Computation Fluid Dynamics (CFD) the field of mechanics which deals with the numerical solving of fluid's motion. It's a precious numerical tool used from the study of laminar flow in wind turbines to the more complex case of turbulent flows in turbomachinery Pahlke et al. (2001), Hammond et al. (2022).

In practical terms, the point of CFD is to deduce a fluid dynamics by solving Navier-Stokes equations:

$$\overbrace{\underline{\nabla}.\underline{u}}^{\text{CONTINUITY}} = 0 \tag{2.1}$$

$$\overbrace{\underbrace{\rho}_{\text{density}} . \left(\underbrace{\frac{\partial \underline{u}}{\partial t}}_{\text{time derivative}} + \underbrace{\left(\underline{u}.\underline{\nabla}\right).\underline{u}}_{\text{convective term}} \right)}^{\text{MASS} \quad \text{ACCELERATION}} = \overbrace{\underbrace{\rho.\underline{g}}_{\text{body force}} - \underbrace{\frac{\partial p}{\partial \underline{x}}}_{\text{pressure gradient}} + \underbrace{\mu.\underline{\nabla}^2\underline{u}}_{\text{diffusion term}}}^{\text{FORCE}} \tag{2.2}$$

CFD is a field in great devenir given application fields are at center of attention (e.g. space conquest Boustani et al. (2020), Koch et al. (2021)) ; and with a great progression step with machine learning technologies. This last point translates through Ph.D theses propositions Merle et al. (2021), Angelini (2019) , as well as research papers Zhao et al. (2020), Hammond et al. (2022).

These recent developments are especially useful in the yet too complex fields of aerodynamics such as turbomachinery. One could then improve generalization aspect of CFD simulations of complex structures together with consistency of simulation.

2.1.2 Turbomachines

Turbomachines are mechanical systems generating an interaction between a fluid component and a solid component, both in movement. This interaction between solid stages and a fluid aims at transferring or extracting energy from the fluid Logan and Roy (2003), Tyacke et al. (2019).

This inherent coupling fluid / structure is a point of interest in multiples axes of research. For instance, if one looks at a single stage of turbomachinery, blades interacting with each other present a great matter for the study of a turbulent flow and the repercussion this flow has on the structure Persico et al. (2019), Rubino et al. (2020).

Similarly, at an even lower scale, the blade of a stage is an asymmetrical structure whose analysis asks for great care regarding meshing process for analysis of structure's damage Du Toit et al. (2019), or as well design of the blade in the first hand Demeulenaere and den Braembussche (1998), Rehman et al. (2018).

These papers highlight that turbomachines are complex structures, eventually difficult to compute as they require specifications attached to each considered cases, both from a fluid and a solid point of view.

As mentioned in CFD section, multiple researches go towards a generalisation of the study of turbomachines. Similarly, such a treatment can be done from a solid point of view, where the meshing process has a great influence over computed results.

2.1.3 Meshing strategies

In turbomachinery, the meshing process aims at recovering accurately properties of the flow and mechanical stress on structures, while keeping calculation cost low.

This asks one to shrink turbomachinery system to a single periodic flow channel and to apply an efficient meshing process to it. This process is a two-sided approach:

- choose adequate cell topology to describe the structure,
- associate a consistent meshing strategy with regard to flow properties.

From tetrahedron and hexahedron to polyhedron, numerous types of cells are available for one to better describe complex structures Wikipedia contributors (2022). These cells extend CFD simulations to more realistic cases, and they are the object of several papers among which we note the anisotropic adaptation for RANS simulations Alauzet et al. (2022) and the automatic polyhedral mesh generation for ship resistance Jeong and Seo (2020).

The second aspect of the meshing process is how one combines and uses these elements to run high performance computing simulations. The meshing strategy must then remain accurate with respect to the CFD context of the ongoing simulation.

Several strategies are employed for turbomachine analysis, from use of unstructured mesh Sbardella et al. (2000), overset-grid process Chang et al. (2020), structured mesh Tucker and Ali

(2014) and polyhedral meshing for adaptation/intersection Diazzi and Attene (2021). As with CFD, the point eventually is to improve these strategies to make it more consistent and less case-dependent to deal with turbomachinery configurations for instance.

Now that we have an insight in what is done in the literature, let's go through the ONERA counterpart.

2.2 Researches at ONERA

We now enter a section where we'll discuss researches and application cases directly in link with the ONERA and more specifically the mission of this internship.

2.2.1 CFD

The research unit DEFI in the Department of Aerodynamics, Aeroelasticity, Acoustics (DAAA) of ONERA develops aerodynamic-systems-aimed numerical tools for research units of DAAA and other departments of ONERA. The use of CFD is thereby at the core of their researches.

Application fields cover laboratory researches as well as industrial systems, through company-partnerships Cambier et al. (2011), Reneaux et al. (2011).

Therefore, in addition to usual study of turbulent flows through CFD Pahlke et al. (2001), the ONERA develops industry-aimed research papers, from computation of ONERA-M6 Wing Mayeur et al. (2016) through aerodynamic simulation and optimization of helicopter rotors Wilke et al. (2021) to the CFD assessment of turbomachines' flows Tailliez and Arntz (2018).

For the ONERA to work with some companies includes the development of their own codes and solvers, eventually put at the use of some of their company-partners:

> *"With elsA, ONERA offers industries, academics, research institutes and service companies a unique aerodynamic calculation code."*
>
> elsA ONERA

In this paper, as we put light on turbomachines, let's see the researches DAAA/ONERA undertakes in this domain.

2.2.2 Turbomachine

The ONERA is the french leader in the aerospace domain, and so a part of its work is dedicated to aerodynamic structures, including turbomachinery systems.

As part of its studies, one can mention the study of wind turbine Lienard et al. (2019), the simulation of the fluid-structure interaction of thrusters through elsA solver Dugeai et al. (2018) and the prediction of flutter for 3D airfoils Liang et al. (2018).

In fact, in cases like turbomachines, importance is as much on choice of CFD solver than on meshing tool.

In this context, ONERA develops multiple solvers, as well as meshing tools like Salome (Salome) and gmsh Gueuzaine and Remacle (2009). Following these meshing softwares, ONERA has developed Cassiopee, a pre-/post-processing tool generalizing mesh generation, and that comes in combination to ONERA's different solvers.

2.2.3 Meshing strategies

Cassiopee is a numerical tool developed since 2008 by the DAAA, and dedicated to the pre-/post-processing of CFD simulations Benoit et al. (2015).

Cassiopee has been developed for this purpose, through several meshing strategies: Coverset grids or Chimera Péron and Benoit (2013), Péron (2016), immersed boundary conditions (IBC) Mincu et al. (2017), conformal assembly Landier (2017), ensuring good grid connectivity at the interface.

Part of the strategy is based on refining mesh locally (especially in overset regions) during assembly process, limiting calculation cost, without neglecting accuracy needed for CFD applications.

This current refinement process then allows a better mesh gradation in the assembly, limiting generation of superfluous points in the far field. It eventually provides a conformal mesh, necessary when using an unstructured solver that does not handle "hanging nodes" like elsA and CEDRE.

In this report, we present the work done in adaptCells functions in order to preserve periodicity during the adaptation process.

The following figure provides some details on the meshing strategies entitled to Cassiopee tool at the moment Landier (2017), Renaud et al. (2018):

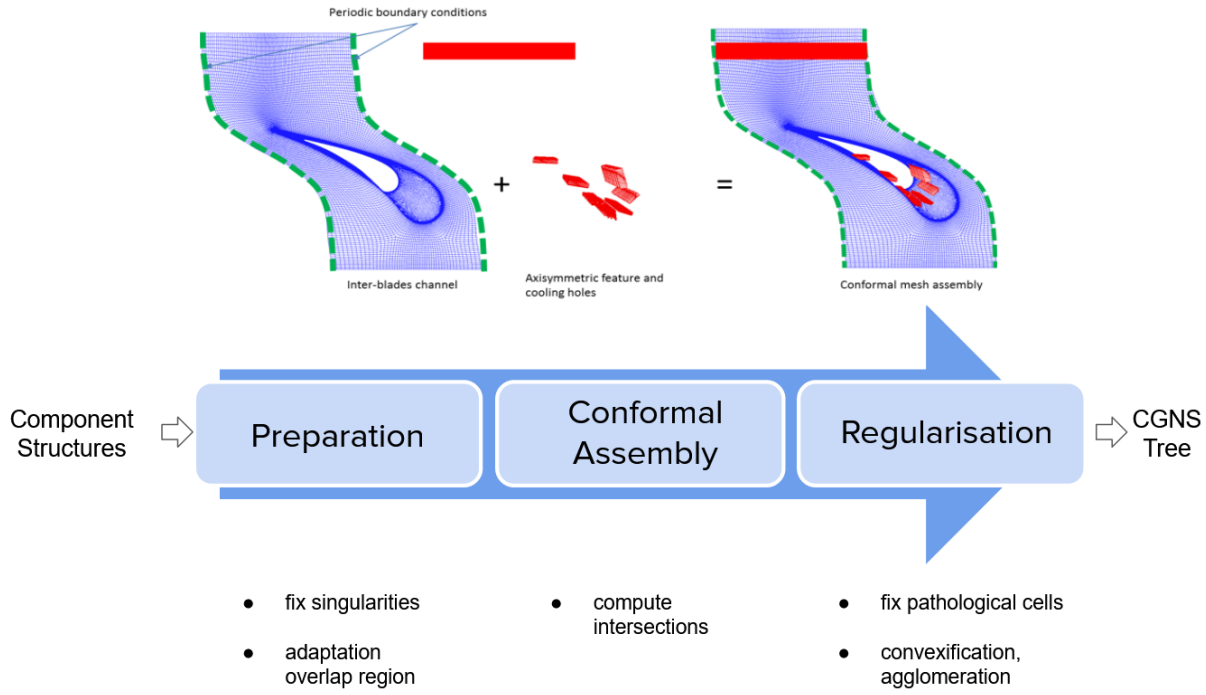


Figure 2.1: Geometric workflow - Meshing strategy under Cassiopee tool. This schematic frieze gathers main features and steps of the meshing strategy. Figures on top provide an example of mesh concerned by the strategy.

The frieze lists the main tasks of each step of the strategy. To detail the strategy briefly, it consists into combining input components/domains by assembling them, in a way that ensures a conformal mesh, necessary for conservativity for instance. The figures on top provide an example of a) external features one would add to a channel, b) the channel we study.

We especially note the periodic boundary conditions in green, which define the limit of the channel.

Let's now ensure periodicity property through this meshing strategy 2.1.

Chapter 3

Development periodicity codes

This chapter will expose and detail the codes and debug processes undergone in the implementation of periodicity property in *Cassiopee/adaptCells-mpi.cpp* and *Cassiopee/Mpi.py*.

Contents

3.1 Insight Cassiopee

Cassiopee is a pre- and post-processing tool for elsA CFD solver, other ONERA-in-house solvers (e.g. CEDRE) and CODA, a solver developed in partnership with the DLR.

Cassiopee does the pre-processing (e.g. meshing, setting the boundary conditions, the physical properties of the flow, preparing the parallelism), and the post-processing by providing data extraction features (e.g. slices, streamlines).

Its major use is done through python scripting but it also provides a graphical user interface, through which one can display the mesh.

It is for these reasons that most Cassiopee functions can be applied transparently to the end-user, for several mesh topologies (e.g. sstructured, unstructured with basic element types or polyhedral meshes).

Given the wide range of application of these solvers (e.g. aircrafts, helicopters, turbomachinery, missiles, launchers), Cassiopee remains a general tool used over multi-disciplinary systems. Thereby, it must be consistent while applied on diverse cases (in terms of geometry and flows).

It's for these reasons that the meshing strategy included in it generalizes mesh generation process, even in complex structures cases.

To better understand the composition of Cassiopee, here is a mind map detailing the main characteristics of this software, together with highlights of the parts we modified (right-hand-side bubbles):

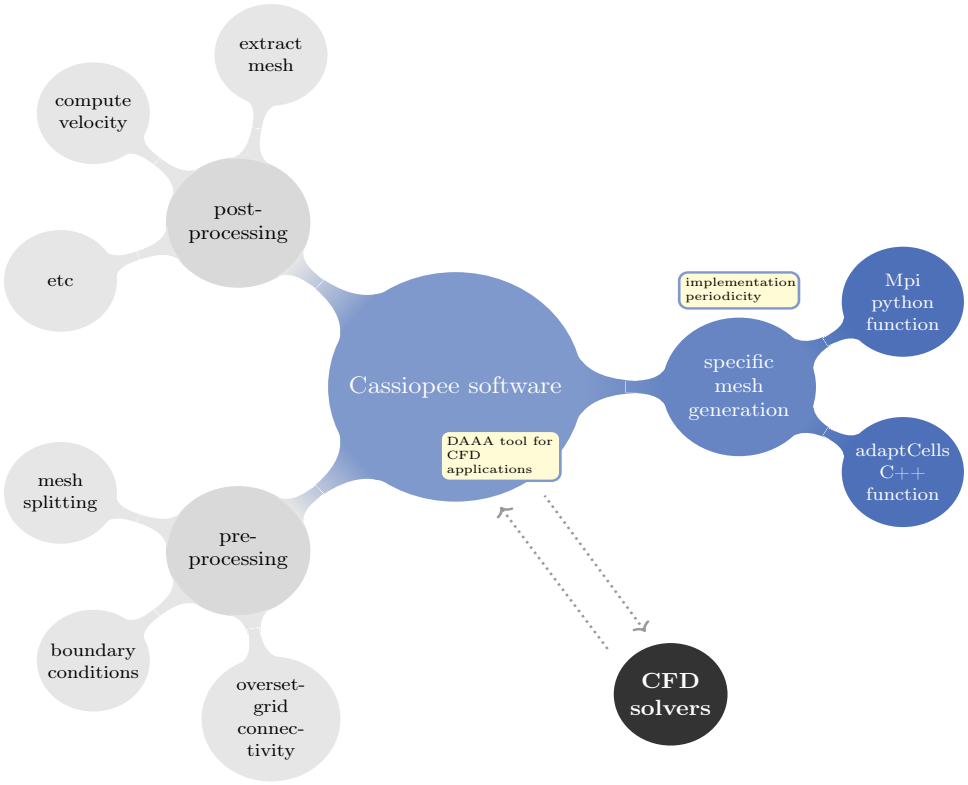


Figure 3.1: Cassiopee mind map

This map summarizes the diversity of applications of Cassiopee, at several stages of the CFD simulation. In this report, we work on the mesh generation process, through python and C++ codes.

Indeed, Cassiopee's structure, in agreement with solvers already in use, combines the programming languages python, C++ and Fortran for the development of its codes.

In fact, Cassiopee presents a python interface (more user-friendly), behind which is hidden C++ code in which most calculation like adaptation process is done.

Our periodicity-implementations will take place in the C++ code ***adaptCells_mpi.cpp*** and its python interface code ***Mpi.py***.

It's mainly these functions we will modify to implement periodicity throughout the meshing strategy.

As it can been seen on figure 2.1, once the meshing process has been applied, one is left with a CGNS tree, which is the format under which mesh is saved.

In fact, this CFD General Notation System (CGNS) allows one to store CFD analysis data, to modify and access it. The CGNS tree then contains among else CFD information such as flow solution, boundary conditions (e.g. inflow/outflow, walls) as well as meshing information like mesh coordinates and grid connectivity. Here below is an example of a CGNS tree:

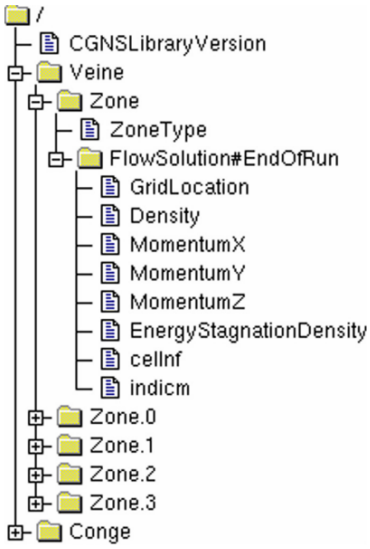


Figure 3.2: Example of CGNS/Python tree.

This system presents the good point of being general enough for one to work on diverse types of structures, elements, flows, which is good in the multi-disciplinary application of Cassiopee (CGNS).

One can then access, add and visualize aimed information in the tree, from pre-processing to post-processing. More importantly, the CGNS file, built throughout the meshing strategy, contains the necessary information for one to set good adaptation process, and periodicity property with it.

In the tree 3.2, each component we want to assemble is called a **zone**. The combination of every zones then defines the **mesh**. Besides, as a zone represents a component like a cooling feature or a fluid, it's as well made of a finite number of **cells**, which in our case are unstructured cells.

The table below provides a visual feedback of the mentioned vocabulary:

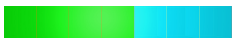
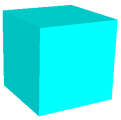
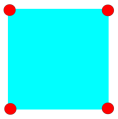
name	visual	name	visual
• mesh		• zone	
• cell		• face	
• nodes			

Table 3.1: Vocabulary mesh.

In the sections to come we will detail how refinement process is done numerically, and how we can implement periodicity through simple structures like the ones in table 3.1.

Eventually, we have an idea of what Cassiopee does, how it works and how we can modify it. Let's now detail coding process undergone in the implementation of periodicity, in Cassiopee's functions.

3.2 Periodicity implementation

As an objective for this section, we propose the following structure, which is a simplified model of a channel to which we add 5 features. We will have to implement periodic-boundary-preserving capability for this curved structure:

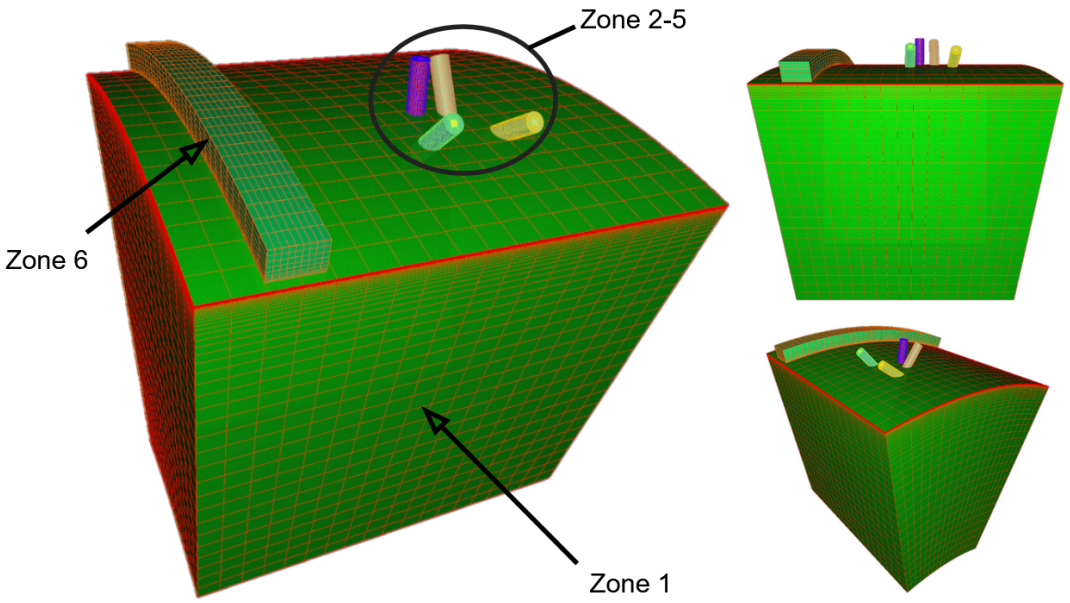


Figure 3.3: Channel structure (zone 1) to which one adds 4 cylindrical features (zones 2-5) and a slot (zone 6). Depending on the position of these different features, periodicity on frontiers may be impacted by the adaptation process.

This section will expose the different steps undergone to develop, test and implement periodicity for the case 3.3. Refinement process at interface will be studied as well.

We will then identify the yet not working cases (with respect to periodicity property) and develop solutions to include them in the adaptation process. These solutions will include among else to ensure periodicity property at every stages of the meshing process. The point is for one to have the necessary tools to study case 3.3.

First, we will briefly go through adaptCells features, to have insights on current refinement process and highlight intrinsic issues within it. These issues on periodicity property will concern self-joined configurations as well as configurations of multiple joins between the same zones.

Then, we will implement translation periodicity and later rotation periodicity in the highlighted periodicity-non-preserving cases.

Eventually, these implementations will be assessed and we will equivalently check that: we now handle new cases and that currently-working-cases still work after modifications have been implemented.

3.2.1 adaptCells

adaptCells is a function handling hierarchical adaptation process withing Cassiopee tool. It takes a conformal polyhedral mesh as an entry and then adapts it depending on asked refinement.

Once each given refinement schemes has been applied, adaptCells function returns a conformal polyhedral mesh as output.

Let's go through some generalities of the refinement process within adaptCells before we furthermore detail its content.

3.2.1 Adaptation generalities

Our final objective is to work on turbomachine structures. Yet, before we can do so we rather work on simpler regular hexahedron cases: cubes. We will derive these elementary tests to expose clearly the refinement process. The work done here will then assess current refinement behavior and it will provide leads on how to implement periodicity.

We will then see how continuity of refinement is made possible, and how we can include periodicity in the process. Note that the discussed elementary tests are cases obtained from Cassiopee, but that we display through LateX TikZ tool.

Periodicity implementation will later be tested and furthermore developed over larger configurations, for applications to more realistic structures. Here is the type of mesh we consider in a first time:

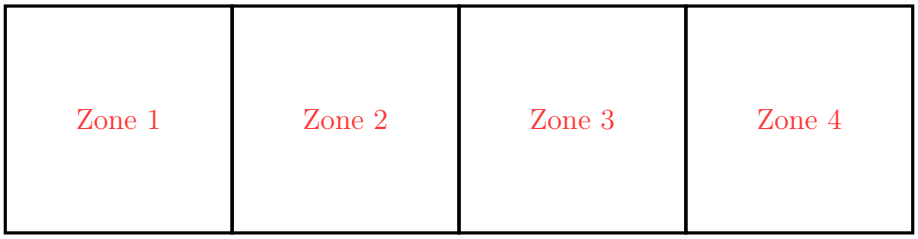


Figure 3.4: 2D view of a mesh made of 4 zones, where each zone is made of a unique cell.

This mesh is a model for a system for which we want to implement periodicity. We will test a refinement scheme over this model to see reaction from the mesh.

In fact, refinement can be seen as a scheme. This scheme contains information of refinement occurring in a part of the mesh. Then, each zone of the mesh will have a refinement scheme: some where this scheme contains refinement information, some others where the scheme is empty (i.e. 0 as refinement scheme) as we do not refine there.

The point will eventually be to assess refinement process by checking that zones read, apply their own scheme and exchange them with joined zones. In a process of numerical exploration and for the sake of simplicity, the scheme will remain simple throughout this chapter.

Here is a 2D view of the refinement scheme considered for mesh 3.4:

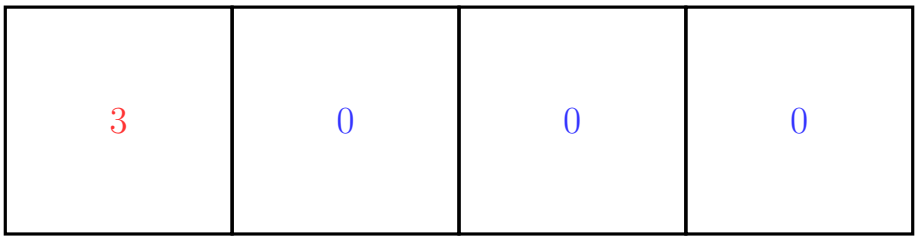


Figure 3.5: Refinement scheme for model 3.4. We apply an order 3 refinement on zone 1. By default, we apply 0 as refinement scheme for the other zones.

This refinement scheme recalls refining locally the mesh. This will create a discontinuity in refinement level.

We then expect, by application of adaptation process, the neighboring zones to refine. The principle of the meshing strategy in fact includes that when mesh presents a discontinuity in refinement, neighboring zones will refine with a 2:1 smoothing rule, towards continuity of refine-

ment. This rule will work this way: if a zone is refined **n** times, its neighboring zones will be refined **n-1** times, and so through each zone until the rule is respected by every zones.

This rule will preserve continuity of refinement at interface (good for conservativity and overall consistence of computation).

Here is how the mesh eventually looks like once refinement occurred:

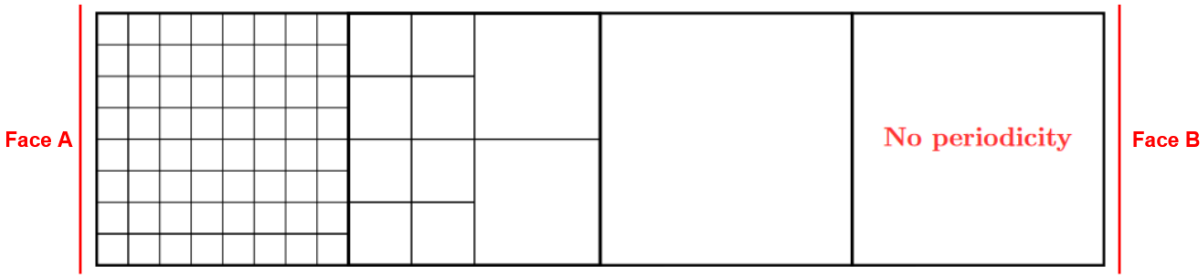


Figure 3.6: Refined mesh, in the case where each cell is a zone. We here refined left-end cell and adaptation process did the rest of refinement. **No periodicity** observed has face A and face B don't coincide.

We obverse a refinement which needs to be furthermore detailed for one to understand the process:

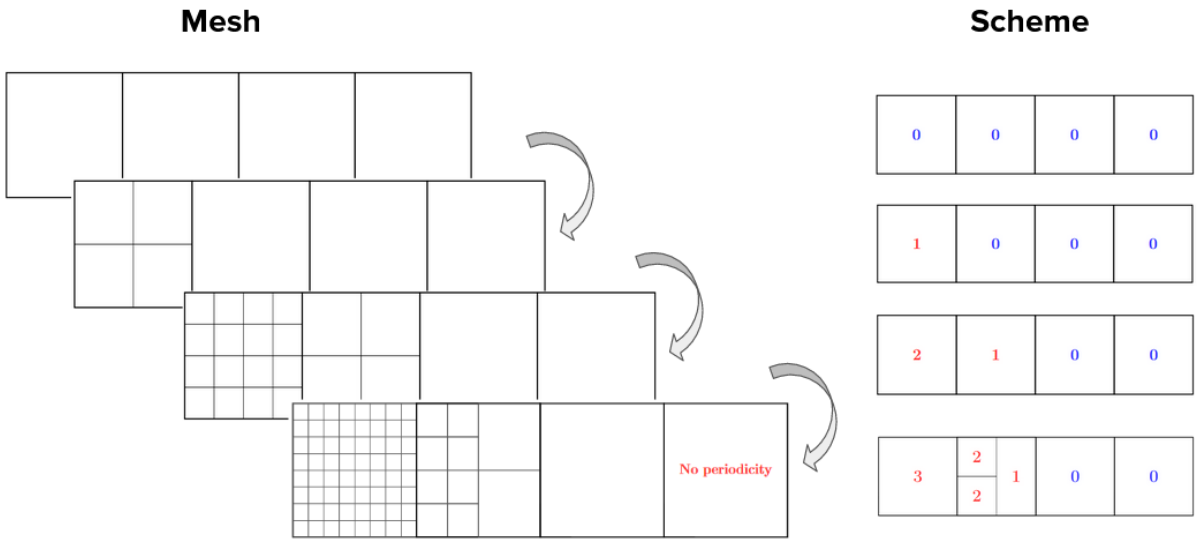


Figure 3.7: Refinement process for the four zones case. We compare each step of refinement scheme with associated mesh. Eventually, no periodicity occurs.

In case 3.4, each cube is a zone. As a zone gets refined, it splits into several cells. In fact, it's over these cells on edge that refinement is applied, under the rule of 2:1 smoothing.

Eventually, as one refines in the four zones case, each zone gets split and refinement then applies over halved zones. We note that 2:1 smoothing rule is respected as one can see in scheme section of figure 3.7.

Let's now adapt code to consider periodicity in addition to existing refinement process.

For the moment, zones are internally connected to exchange their refinement schemes with each other. In fact, we connect each neighbor zones, thus exchanging refinement schemes.

If one considers two zones: zone A will send its refinement scheme to zone B, and zone B will do the same with A. The point is to create a link between zone A and zone B, to ensure continuity of refinement.

What we do then is to tell the code which zones are connected to each other, and then make

sure refinement process proceeds as expected.

Applying this process to the four zones case 3.4, we get above behavior, namely refinement 'propagates' between each connected zones. From a coding point of view, it comes to:

```
1   #include "adaptor_mpi.hxx"

3   void adapt_MPI()
    {
5
    // ----------- Link between the 4 zones ----------
7
    std::vector<E_Int> Bound_Zone1_to_Zone2{ListFaceA}, Bound_Zone2_to_Zone1{
    ListFaceB};
9   std::vector<E_Int> Bound_Zone2_to_Zone3{ListFaceA}, Bound_Cell3_to_Cell2{
    ListFaceB};
    std::vector<E_Int> Bound_Zone3_to_Zone4{ListFaceA}, Bound_Zone4_to_Zone3{
    ListFaceB};
11
    //set link
13  zone_to_list[Zone1][Zone2] = Bound_Zone1_to_Zone2;
    zone_to_list[Zone2][Zone1] = Bound_Zone2_to_Zone1;
15  zone_to_list[Zone2][Zone3] = Bound_Zone2_to_Zone3;
    ...
17
    // ----------- Refinement schemes ----------
19
    // set refinement schemes (default is 0)
21  E_Int refinement_scheme[Zone1] = 3;

23  // apply refinement schemes to every zones
    for (z=0; z < NbZones; ++z)
25  {
    adapt[z]->assign_scheme(refinement_scheme[z]);
27  }

29  // ----------- Adaptation function ----------

31  // adapt zones with refinement_scheme
    adaptor_mpi::run(adapt, zone_to_list);
33  }
```

Listing 3.1: Code of refinement process for the four zones case. This code details the bounds between each zone, as well as transfer of refinement schemes and call of adaptation function.

These few code-lines translate a connection between neighboring zones, which visually translate as:

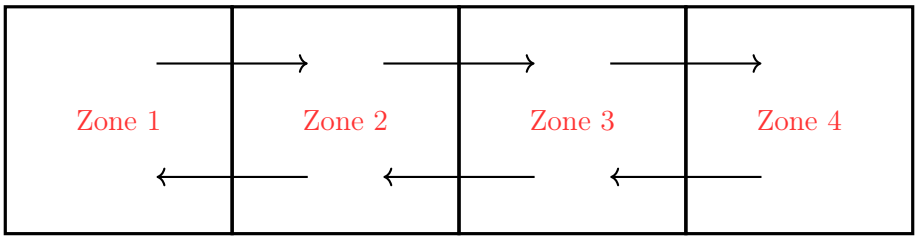


Figure 3.8: Intern connections between zones of fig 3.4. Each bound ensures continuity of refinement between associated zones, namely each bound transmits information of refinement.

We note that to link two zones, we in fact link two faces with each other. Indeed, the connection between two zones is done by saying that neighbor zones have a common face (i.e. *lines 13-16* in code 3.1).

We then tell the code which zones are connected, and more specifically the pair of faces connected. Doing so, we create a continuity in refinement process.

To implement periodicity, one simply has to add a similar link to *lines 8-10* of code 3.1, between the two end faces. Eventually, we will get a complete cycle bounding every zones.

Once this link has been added to code 3.1, we get the following refined mesh, obeying the 2:1 smoothing principle as well as refinement process from fig 3.7:

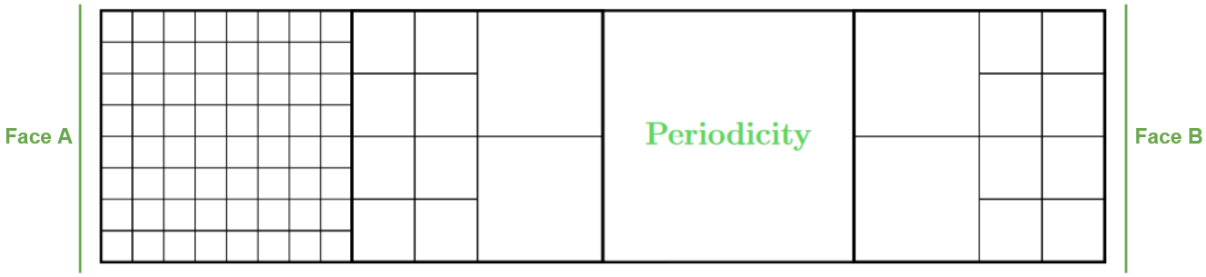


Figure 3.9: Refined mesh, in the case where each cell is a zone. We here refined left-end cell and {adaptation process + periodicity} did the rest of refinement. **Periodicity** is observed as face A and face B coincide.

To sum up, refinement process is here done over an elementary mesh. We bounded one by one the zones with each other, thus building match-connection (i.e. intern link between zones) and then periodicity-connection.
Doing so, each zone exchanged its refinement scheme with its neighbors, allowing good refinement process.

Let's detail adaptation process within adaptCells function.

3.2.2 MPI parallel mecanism

First, to proceed to a join, adaptCells takes several inputs, which gather information of:

- components we work on, known as zones;
- location of zones with respect to processors;
- link between zones and faces, zones and connection, etc.

As discussed previously, refinement in the mesh asks for continuity between each component. We thus bound zones together to create a link through which refinement's information are transmitted.
In the context of HPC, calculations within zones can be split over several processors. Thus, one can work on several processors, each containing packs of zones (from 1 to n), towards parallel calculation.
Adaptation process then requires to treat both local information (same processor) and parallel information (different processor). Let's present adaptCells's hierarchical adaptation process:

1. First, adaptCells function charges a processor and the zones that come with it. To a proc, we can then have from 1 to n zones.

   There, we apply refinement schemes to each zone that has one. Then, each zone proceeds to its local refinement, without caring about what other zones do. Note that no connections are applied at this stage.

2. adaptCells function splits connections between zones in two groups: local connections (between zones in the same proc) and remote connections (between zones in different procs).

   The point is to know if the zones we try to connect are from the same proc or from different ones. These two groups are saved and they will structure exchange process.

3. We proceed to exchange of schemes between remote zones. It's at this step that we set connections between zones and that we exchange refinement schemes.

   In order to better understand where information is localized, here is a figure gathering main information:

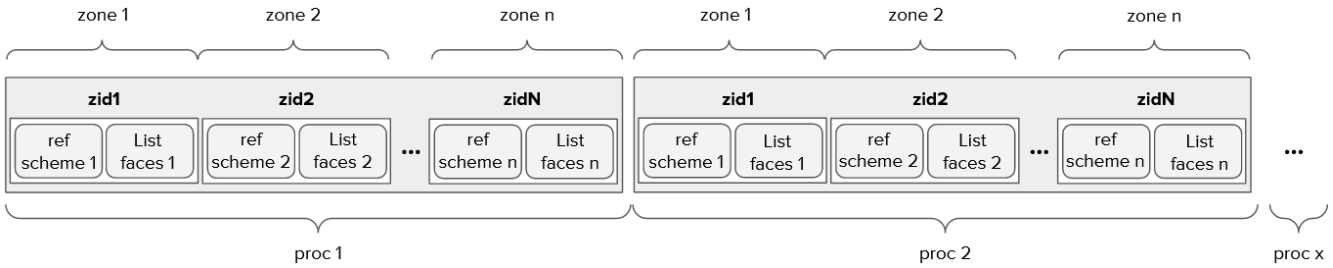


Figure 3.10: Figure telling how the pack of data exchanged between MPI ranks looks like. This structure provides one with the necessary information of links between procs and zones, between zones and lists of faces, etc.

Information associated to a zone are thereby its refinement scheme (do we want to refine here) and list of faces concerned by a connection.

With current process, we have one list of faces per zone, and so it's one connection per zone.

In details, each connection associates two zones, and each connection is made of two half-bounds, one per joined zone. Each half-bound contains the list of faces concerned by a connection, as well as information of joined zone.

Visually, a half-bound looks like this:

$$half-bound \;=\; (zone1,\; zone2\;(=joined\_zone),\; list\_faces\_1)$$

Thereby, for one to bound zones together leads to the following joining process:

$$Join \;=\; (zone1,\; zone2,\; list\_faces\_1) \;\longleftrightarrow\; (zone2,\; zone1,\; list\_faces\_2) \quad (3.1)$$

It comes to tell the code that zone1 is bounded to zone2 through the list of faces 1 and that zone2 is bounded to zone1 through the list of faces 2. Physically, the lists of faces 1 and 2 are at the same location, they are the common faces between zones 1 and 2.

In order to ease manipulations and reduce processing time, formalism 3.1 has been extended to the following one in C++ layer:

$$Join \;=\; (zid1,\; zid2,\; list\_faces\_1) \;\longleftrightarrow\; (zid2,\; zid1,\; list\_faces\_2) \quad (3.2)$$

where **zid** stands for **z**one's **id**ex.

Thereby, one doesn't deal with names of zones but rather indices of zones (zid). It's this formalism that is currently used in adaptCells.

To sum-up, to each zone we associate an index zid, and we connect two zones by applying two half-bounds, resulting in a connection.

Numerically it implies to exchange ptLists, namely lists of faces. Once we have exchanged ptLists, and associated refinement schemes, we let zones refine.

4. Once every zones have refined, we move onto exchanging schemes again. The point is to ask connected zone if they have refined during previous step.

   If they have, zones will exchange information again, before adapting again.

Finally, when no refinement is needed anymore, it's that adaptation process is over.

A paradox with current adaptation process is that the same index zid which allows to connect zones, is the one that limits application cases. As discussed, adaptCells associates a unique index zid to each zone. Then each zid has a unique list of faces and so a unique connection is possible per pair of zones.

This poses two problems:

- what if we are in a self-joined case, namely a case where a zone is bounded to itself ?
- how to have several connections per pair of zones ?

For this first point, we know that to have a connection we need information from the two parts of a common face. Yet, as we have one zone only there, we have access to one list of faces only. We will see in the following how to recover the two lists of faces in self-joined case.

The second problem will be handled by an extension of zid formalism to a more general formalism. We will then derive a new index rid to consider more connections.

To these 2 problems we will add one in rotation periodicity case, that we will introduce and discuss then.

Now that we better understand the inherent issues of adaptation process, let's solve them one by one. First, we tackled down the two first points through translation periodicity.

3.2.2 Translation periodicity

3.2.1 Self-joined case

general process Let's have a look again at previous case 3.4. If one considers a unique zone:

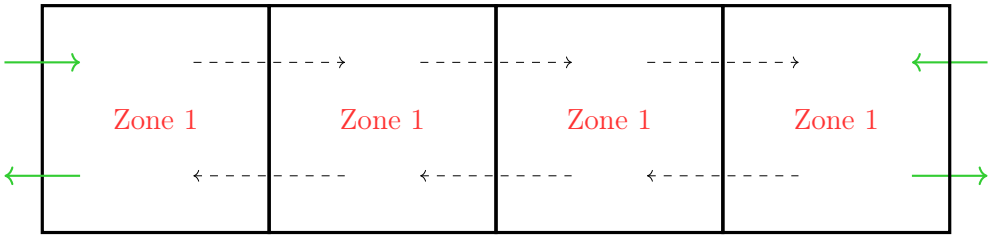


Figure 3.11: 2D view of a mesh with a single zone made of 4 cells. We here model a unique domain over which we apply refinement scheme from fig 3.5. Dashed black line implies that we don't deal with match connection in one zone case. In plain green line we have the periodicity connection under half-bounds-shape for each side.

This case 3.11 differs from 3.4 in that

- we have a single zone with 4 cells (instead of 4 zones with one cell each). Thus, we don't need to specify joins for inner faces in a unique zone;

- periodicity connection is here done between one zone and itself. It's what we refer to as self-joined case.

From this observation it comes that *lines 13-16* in code 3.1 will become:

```
zone_to_list[Zone1][Zone1] = Bound_Zone1_to_Zone1;
zone_to_list[Zone1][Zone1] = Bound_Zone1_to_Zone1_bis;
```

This formulation presents a problem as we define several times a same link. Yet, the code will only keep the second one, overwriting the first one.

The issue is that we need a back and forth communication, for both sides to exchange their schemes. Otherwise, we will miss information, making the code not consistent for turbomachinery applications.

Besides, we must fix this issue without modifying the current process, in order to keep what is working and to just extend it to self-joined case (i.e. when we link a zone to itself). A solution is to transmit all info at once rather than in several packages, namely:

```
zone_to_list[Zone1][Zone1] =
[Bound_Zone1_to_Zone1, Bound_Zone1_to_Zone1_bis];
```

With line 3.2.1, we propose to send the lists of faces in the same list, by concatenating donor (*PointList*) and receiver (*PointListDonor*) lists of faces. It offers a solution to fix self-joined case, while keeping current structure untouched. One will afterwards have to decompose this list.

Thus, one concatenates the lists *PointList* and *PointListDonor* of faces when we send information for adaptation process 3.2.2, and one later decomposes the list when refinement process has been applied (i.e. to update *PointList* and *PointListDonor* with new shapes and content).

In the Appendix A, you will be presented the code we implemented to handle decomposition of concatenated lists. The concatenation part is very similar to the one presented in code 3.1.

The functions modified here are source functions deep in Cassiopee tool. What we do here is to set the possibility for adaptCells to work with concatenated lists during adaptation process 3.2.2.

However, it's still left to build the concatenated list in the first hand. Sâm Landier did the implementations in the C++ layer (*adaptCells_mpi.cpp*) while I proceeded to equivalent implementations in python interface (*Mpi.py*).

These implementations take place where we deal with ptLists (i.e. lists of faces), namely in *getJoinsPtLists* and *updateJoinsPointLists* functions.

Mpi.py - getJoinsPtLists The former function, *getJoinsPtLists*, provides a list of zones and their list of faces, i.e. *getJoinsPtLists* provides the structure 3.10.

First, a condition is needed to know when we connect a zone to itself. We then compare current zid associated to *PointList* to the index of zone associated to *PointListDonor*. If the two zid are the same, it's that we have a self-join. Otherwise, we are in the usual case, namely a case already working.

Once we know we deal with self-joining, we proceed to lists' concatenation. We thus merge the two ptLists into one ptList. This information now has the good format for adaptation process 3.2.2. Output from this function is then sent to C++ layer where one does equivalent modifications of lists.

Now that we have the good format for the list of faces, it's left to build the counterpart in decomposition.

Mpi.py - updateJoinsPointLists The latter function which needs special care in self-joined case is *updateJoinsPointLists* function where we deal with ptLists once refinement has occurred. This function takes concatenated ptLists as an input and asks for decomposition.

The point of this function is to update both *PointList* and *PointListDonor*, with new shapes and content.

Indeed, as we refined, we have zones made of more cells and so lists made of more indices, and this must be made known to Cassiopee tree.

Once we have implemented this decomposition step in function *updateJoinsPointLists*, post-processing is made consistent and implementation of one zone case is finished.

Code can then handle periodicity for one zone case:

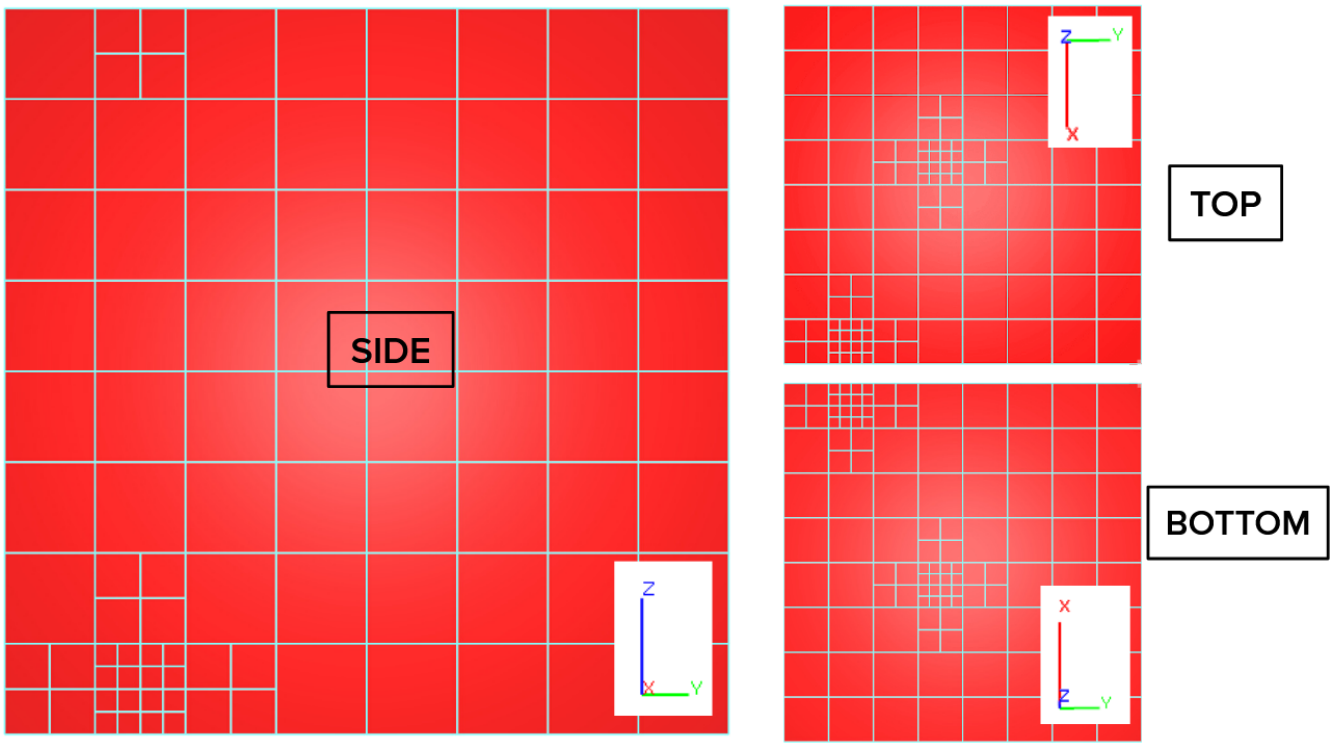


Figure 3.12: 2D views of the one zone case, where one refined locally two cells from bottom face, and refinement has propagated to top face through translation periodicity. The translation periodicity has there been set to have periodicity between the two end faces along e_z axis.

In figure 3.12, we refined bottom face, and we observe that translation periodicity did its job as expected given top and bottom faces are the same within a translation.

This case offers a visual feedback to what was meant by coincident faces in figure 3.9.

We furthermore observe that, once periodicity has been applied, there has been the expected adaptation step given left face presents refinement on its top (see 3.12).

Let's now tackle down the second problem of adaptation process by extending zid process to account for more than one connection per pair of zones.

3.2.2 Multi-bounds - zid to rid

From adaptation procedure 3.2.2, we know refinement process lacks consistency. In fact, the periodicity is handled as long as we don't have more than one join defined per pair of zones.

Once this condition isn't respected, refinement process isn't consistent, and so periodicity property isn't respected anymore.

To fix this, we propose to extend current zid process to a new one. Thus, we can't only think in term of number of zones, but we must include more nuances.

Let's consider a two zones case, for which we want to create a match connection as well as a translation periodicity connection:

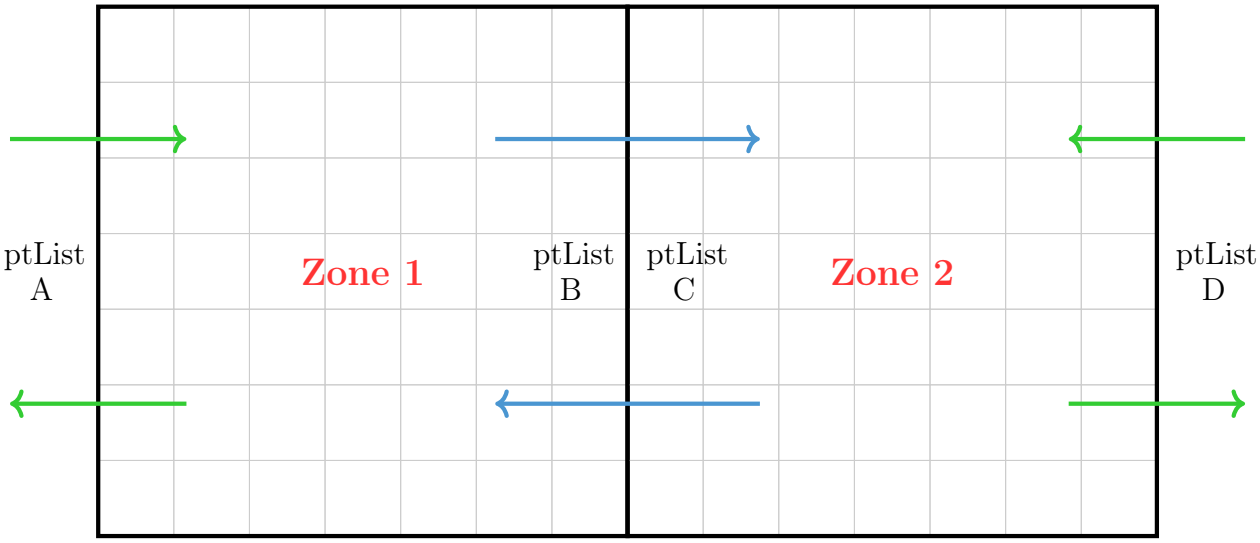


Figure 3.13: Model for a two zones mesh, where each zone is made of several cells ($\approx$ 100). We have match connection in blue, and periodic connection in green. For periodic connection, we can see it as two-halves of a same connection, one at the left and the other at the right.

Looking at figure 3.13, we highlight that two zones case presents more than one connection between the two same zones. This issue is the one highlighted in elementary structures: some of the bounds will be overwritten to the benefit of others.

zid flag Let's have a look at how code currently creates zid indices:

```
import Converter.Internal as Internal

def set_zidDict(CGNS_tree):

    # ---------------- Initialization ------------------

    zones = Internal.getZones(CGNS_tree)
    idx_connec = -1

    zidDict = {}

    # ------------- Cration zid + zidDict ---------------

    for zone in zones:
        idx_connec += 1

        # zidDict
        zidDict[zone] = idx_connec

         # save zid in CGNS tree:
        connec=Internal.newIntegralData(name='zid', parent=zone, value=
idx_connec)

```

```
    return zidDict
```

Listing 3.2: Code creating zid flags, together with zidDict, associating a unique connection index zid to each zone of the mesh.

From code 3.2, we observe two different processes:

- we create a flag each time we read a new zone $\rightarrow$ *zid* ;
- we build a dictionary bounding each zid flag to associated zone.

Visually, code 3.2 translates by:

$$zidDict = \{zone\_1 : zid\_1,\ zone\_2 : zid\_2,\ zone\_3 : zid\_3\ ...\}$$

This zidDict dictionary comes in complement of zid flag, in order to have necessary information when proceeding to a join. The issue then is on how zid flag is created in the first hand.

It is currently bounded to the number of zones, and we would like to adapt code 3.2 to account for more specifications regarding idx_connec attribution.

rid flag A way to do this is to say the two connections we have in fig 3.13 don't concern the same lists of faces (ptLists). In fact, we have 4 ptLists and we say that:

- ptList A and ptList D are a common ptList in periodic connection;
- ptList B and ptList C are a common ptList in match connection.

We then have two distinct ptLists across which code transmits refinement. Given each ptList has its own local indexing, one can compare them to differentiate the 2 bounds of fig 3.13.

Generally speaking, if we have a link bounding zone 1 and zone 2 through a certain ptList, we have a connection.

If we encounter a new connection between zone 1 and zone 2 through another ptList, then we have another connection.

For implementation's sake, rather than looking to the whole ptLists, we will rather look at minimum of ptLists. This way, to each connection will be associated a min ptList, namely the minimum index in ptList. We then have two min_ptList that we will compare to differentiate the two connections in fig 3.13.

Thus, if the min_ptList of 2 connections between the 2 same zones are different, it's that we have 2 different connections to deal with.

In the end, every time we encounter a new connection, we raise a flag, which comes with an index rid, unique for each connection. This creates the rid flag in CGNS tree, saving current state of the mesh.

The code in Appendix B details creation of the new flag rid and how we add this info to the tree. In this appendix, we develop the criteria used to set rid flag and we show flag's creation.

Moreover, as a tool for us to work with, we create an equivalent to zidDict: *ridDict*. It contains each rid indices paired with associated zones:

$$ridDict = \{rid\_1 : (zone\_1,\ zone\_2),\ rid\_2 : (zone\_1,\ zone\_2),\ rid\_3\ ...$$

This *ridDict* dictionary contains every indices of connection, between given couple of zones. As in *zidDict* case, one then goes through it when applying adaptation process 3.2.2 to get information of connection (i.e. bounded zones).

The code Appendix B is implemented in adaptCells, such that we now changed the way we flag a connection. Let's now detail where we called this rid tag and associated ridDict in adaptCells codes.

Implementation rid in adaptCells First and foremost, we must update the structure 3.10 with the newly implemented rid flag. To do so, one goes through *getJoinsPtLists* function. There, we go from current structure to a list of zones of lists of rid and their list of faces, i.e. *getJoinsPtLists* provides this new structure:

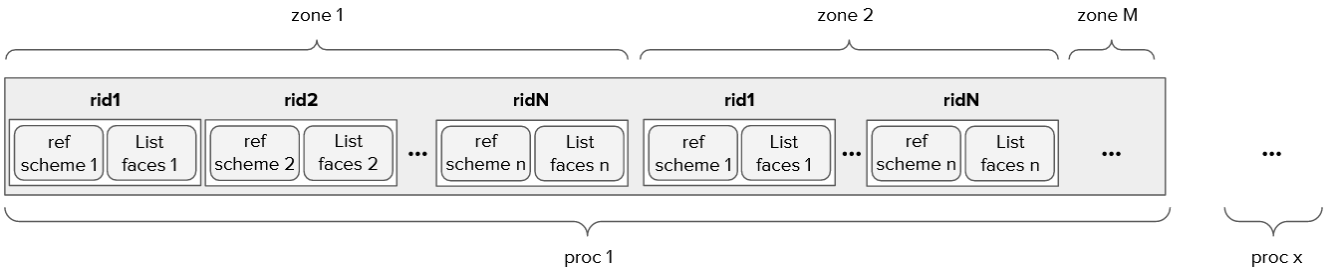


Figure 3.14: This figure is an extension of fig 3.10 where one considers a new flag (rid) for each connection rather than one for each zone. It's the structure we work with now in exchange process between MPI ranks. It provides one with the necessary information of links between procs and zones, between zones and rids, etc.

With new process 3.14, we now have **n** possible connections per zone. Using ridDict, we know which zones are bounded and code can exchange ptLists. This dictionary then is at the core of connection phase.

It is used in 3 functions of *Mpi.py*: *_exchangePointLists*, *_adaptCells* and *_conformizeHMesh*. Let's see why ridDict is used in these function:

_exchangePointLists	In this function, we do 2 things: • exchange ptLists through exchangePointLists function in C++ code; • update ptLists *PointList* and *PointListDonor* once refinement has been applied. For this last point, *_exchangePointLists* function updates ptLists and synchronizes with others zones. ridDict is thereby necessary to know where to localize ptLists in these two steps.
_adaptCells	This function acts similarly to the source code 3.1. Then, it's where we do equivalent of *lines 19-28*. In particular, we point zones where refinement will occur (thanks to ridDict), and we apply refinement process by calling *adaptCells_mpi* C++ function.
_conformizeHMesh	This function converts basic element of mesh to a polyhedral mesh. Polyhedral mesh is then replaced in the tree while BCs/Joins/Fields are transferred ; requiring ridDict to access ptLists for conversion.

Table 3.3: Table gathering functions we modified in *Mpi.py* code. These functions call ridDict and they set inputs for C++ functions.

In these functions, we mainly call ridDict 3.2.2 to replace zidDict 3.2.2. Now that we have briefly detailed the python codes we modified, let's go through the C++ part through *adapt-*

Cells_mpi.cpp function, where refinement process and periodicity are applied.

Once we get into C++ counterpart, it's necessary to retranscribe input dictionaries, tables, tuples, etc, from python codes into variables readable in C++. To this aim we create the function *convert_dico_to_map__int_pair_int* where we decompose ridDict into variables for rid and the pair of zones. These variables will then be accessible at any moments, which will find itself useful to know which zones to access, and to know where to search the ptLists in CGNS tree.

Here below is an insight into this *convert_dico_to_map__int_pair_int* code that we implemented:

```
void convert_dico_to_map__int_pair_int
(
    PyObject *py_rid_to_zones, //ridDict
    std::map<int, std::pair<int, int>>& rid_to_zones)
{
    PyObject *py_rid /* rid */, *py_pair /* pair of zones */

    while (PyDict_Next(py_rid_to_zones, &py_rid, &py_pair))
    {
        // ------------- rid flag ---------------

        int rid = (int) PyInt_AsLong(py_rid);
        std::pair<int,int> pair_rid;

        // ----------- pair of zones ------------

        PyObject * z1 PyTuple_GET_ITEM(py_pair,0);
        PyObject * z2 PyTuple_GET_ITEM(py_pair,1);

        pair_rid.first = (double) PyFloat_AsDouble(zone1);  // zone1
        pair_rid.second = (double) PyFloat_AsDouble(zone2); // zone2

        // -------------- ridDict ---------------

        rid_to_zones[rid] = pair_rid;
    }
}
```

Listing 3.3: Decomposition of ridDict in C++ code *adaptCells_mpi.cpp*. This allows one to read the dictionary and use information in it for later manipulations.

Once this code has been created to read ridDict dictionary, we call it in adequate functions. In fact, we must decompose input ridDict each time we use ridDict, and so it's each time we used ridDict in python *Mpi.py* code.

From the functions of table 3.3, the functions sending ridDict as inputs for C++ codes are *_exchangePointLists* and *_adaptCells_mpi*. As discussed before, one uses ridDict where connections are created (exchange of ptLists) and it's needed for refinement purpose (adaptation of cells).

These python functions send ridDict in C++ counterpart respectively in *exchangePointLists* and *adaptCells_mpi*. There, we call the newly-implemented function *convert_dico_to__map__int_pair_int* to decompose ridDict. Then codes have access to decomposed information to proceed to hierarchical adaptation.

At this stage, a few modifications are needed to adapt hierarchical refinement process to the presence of rid label. These few steps have been implemented by Sâm Landier.

To sum-up, initially the code was adding a connection for each zone we worked with (zid), while now we have a connection for each pair of zones connected through a different face (rid).

We then make sure to create an index of connection for every bounds made through a different face.

We then didn't specifically implemented periodicity but rather extended existing codes to account for more connections, which was needed to account for periodicity in addition to match connection.

Let's check that our code works as expected in two zones case 3.13:

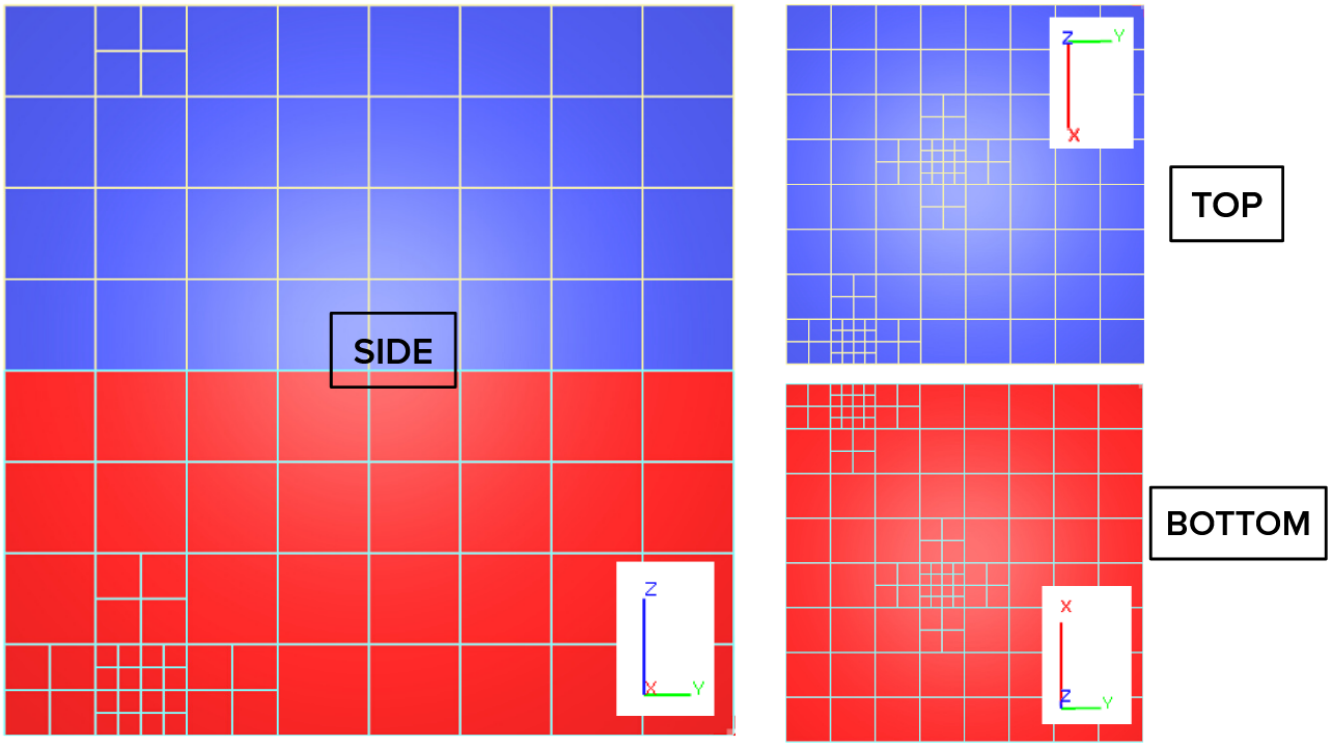


Figure 3.15: 2D views of the two zones case 3.13, where one refined locally two cells from red zone, and refinement has propagated to top face through periodicity link. The translation periodicity has there been set to have periodicity between the two end faces along e_z axis.

From fig 3.15, we observe good refinement process, with expected periodicity applying between the two end faces along e_z axis. We here refine two cells and observe good refinement through inner match and periodicity.

The three views represent the same case after refinement. They put in light the periodic faces as well as overall refined mesh.

Let's do a final test over a three zones case, which is concerned by needs covered by rid process.

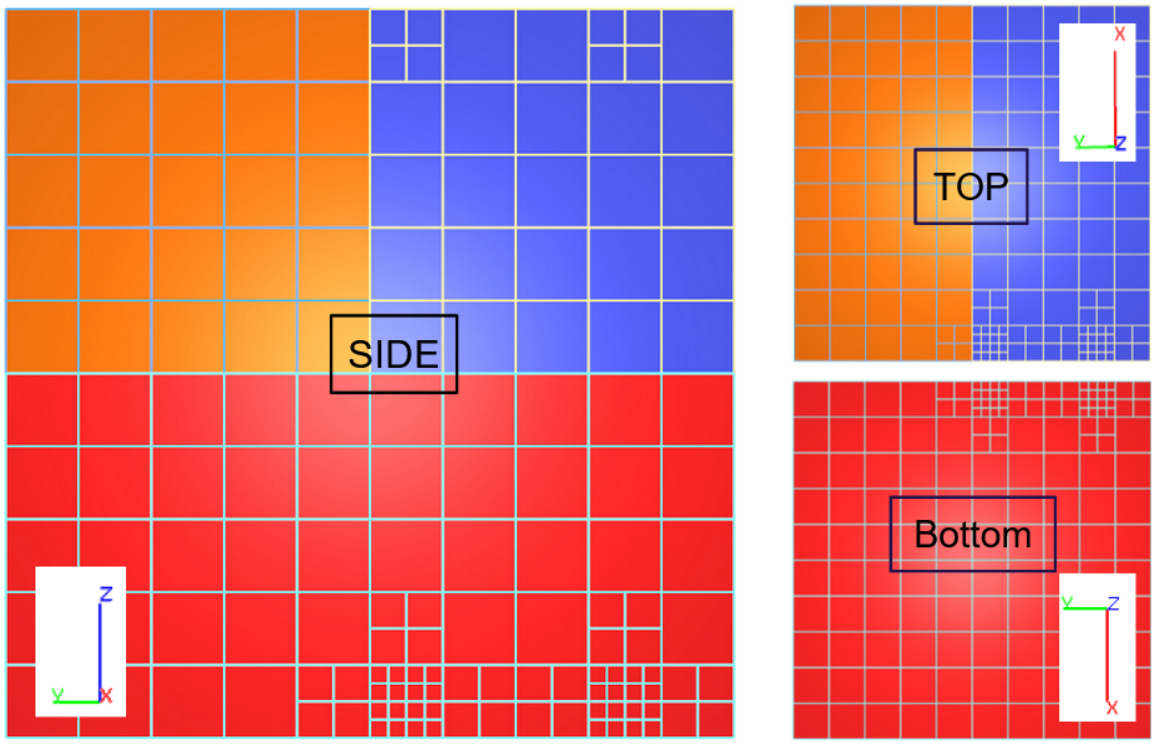
3.2.3 Validation translation periodicity

In this subsection, we propose the case of a three zones mesh for which we want to test good refinement process. The point once again will be to refine mesh locally, and to proceed to refinement through adaptCells code.

In this three zones case, one zone is concerned by refinement ; in this zone we there refine two cells.

What we do then is to send the mesh (i.e. CGNS tree) to adaptCells code, and we ask for a translation periodicity between the two end faces (along e_z axis).

The zones are localized such that it will include issues of multi-bounds case. Here below are figures showing the mesh after refinement took place:



(a) $(0,y,z)$ view of the refined three zones mesh.

(b) $(0,x,y)$ views of the periodic faces.

Figure 3.16: 2D views of the three zones case, where one refined locally two cells from bottom face, and refinement propagated to top face through periodicity link. The translation periodicity has there been set to have periodicity between the two end faces along e_z axis.

Let's now proceed to post-processing analysis of the ongoing refinement process:

- we refined bottom face, and there refinement with neighboring cell took place;
- we then sent refinement data through periodicity bound, from bottom face to top face;
- mesh has been furthermore refined such that you observe the 2:1 smoothing rule occurring in the top of fig 3.16a.

Eventually, in the configuration 3.16, we get that the red zone is connected 4 times: 2 times by periodicity, and 2 times by match join. pointLists are then created with respect to the configuration 3.16, and one is therefore left with a red zone connected twice to orange zone, and twice to blue zone. We have been able to study this case thanks to rid implementations.

This example then shows that our modifications on rid process will have repercussions over more cases than tested, confirming its general aspect and its consistency over complex layouts of domains.

In the rotation case that comes, we will see how this three zones configuration presents features of both rid process and self-join process.

3.2.4 Conclusion self-join / multi-joins

As a conclusion, we now have **rid** to handle case of several bounds per zone ; as well as **concatenation** of **ptLists** to handle the self-joined case.

These rid tags allow the connection to be clear and unique, for more cases than what was previously possible. We then transmit ptLists without issue, repetition, or else. In the one-zone case,

we modify slightly the way we transmit lists of faces, without modifying the data nor the process.

In order to study mesh 3.3, let's now extend adaptCells codes to the case of rotation periodicity.

3.2.3 Rotation periodicity

In this section, we will try to understand:

In what does rotation periodicity differs from translation periodicity in application of refinement process ?

So far, to ensure good refinement process, we made sure that we accounted for every existing bounds between zones. We then exchanged ptLists, without caring about **faces synchronization**. In fact, here is how refinement process is done:

- we want zones to exchange ptLists with each other.

  What is done by the code when given the ptLists is to exchange information between faces. The discrete information of refinement is then contained in faces, that we bound to each other when exchanging ptLists.

- to ensure that the ptLists are sorted the same way on both sides of a join, we use a mechanism that needs to synchronize faces, namely to ensure that they have the same starting point.

  This step comes as a pre-processing step of adaptation phase. We here make sure faces of common ptLists are sorted the same way, which will ensure good sorting of ptLists after adaptation as well.

  Synchronization is made through *shift_geom* function, which does the following:

  – *shift_geom* takes as input a NGON structure, which is a structure containing an explicit definition of the mesh. This input has the following shape:

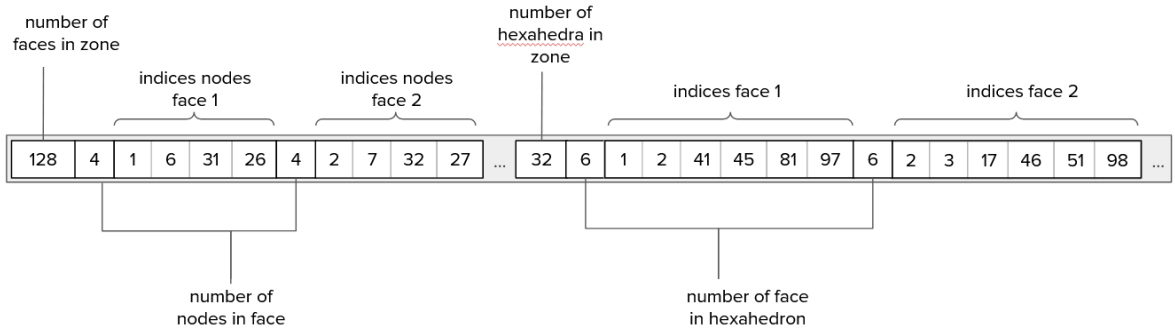


Figure 3.17: Structure gathering mesh information from number of faces per zone to the number of hexahedra per zone as well as indices of nodes for each face.

From this structure 3.17, we have access to a combination $[number\ of\ nodes\ -\ list\ of\ nodes]$ for every faces of a zone.

Given we are interested on some faces only (the one of ptLists), we can go through this structure, searching for the indices in ptLists, and we then get associated lists of nodes.

The objective will be to modify these nodes in a generic way, to eventually ensure good synchronization of faces.

Let's consider two faces A and B which are connected by a translation periodic connection. The following figure shows current state of the faces:

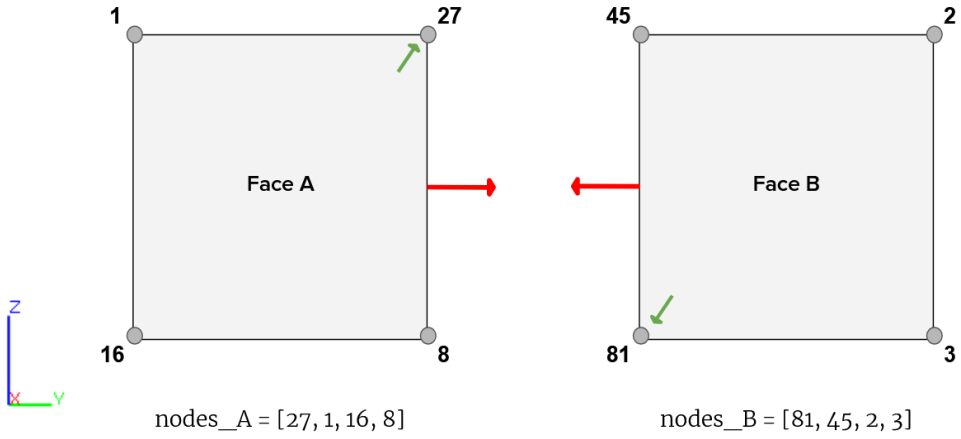


Figure 3.18: Example of connection between two faces A and B. The red arrows stand for inwards normals while the green arrow points the first node in list of nodes. This sorting is arbitrary and is the one obtained from 3.17.

Let's now proceed to nodes synchronization of faces, namely good sorting of nodes indices.

– by convention, a mesh is correctly oriented if the normal of exterior faces points outwards.

A first step then is to swap nodes to make sure faces are oriented outwards. This criterion ensures coherence of the mesh in global framework.
This is done by calling *reorient_skin* with object 3.17 and ptLists as inputs. Here are the resulting faces A and B:

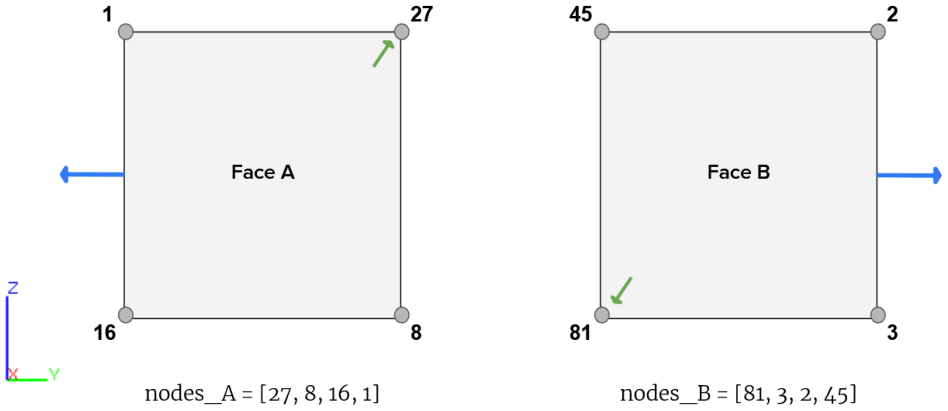


Figure 3.19: Faces from fig 3.18 went through *reorient_skin* function. In blue, we now have outwards normals as opposed to fig 3.18. The green arrow represents the first point in list of nodes for each face. Some nodes indices are swapped to reach coherent orientation.

– what is left to do is to align one node in both lists, and good sorting will be ensured for the both faces.

In particular, we align first point of each face, referring to the following generic-sorting plan:

* the first node is picked as the one with the lowest x coord ;
* if several nodes are at same x coordinate, we look at the node with the lowest y coordinate ;

* again, if several nodes have same x and y coordinates, we look at the node with the lowest z coordinate.

Eventually, this process leads to a unique node which will play the role of first node. We then go through lists of nodes respecting outward-normal-rule, plus first points are now aligned:

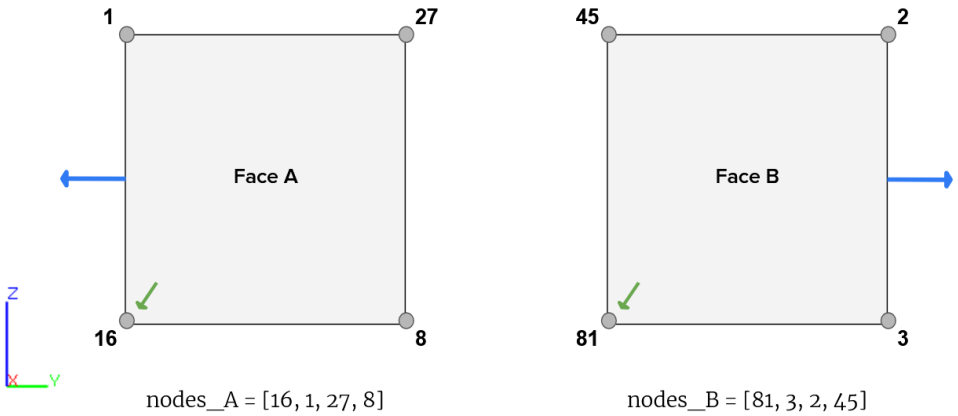


Figure 3.20: Faces from fig 3.19 went through *shift_geom*, then now both faces have the same initial point. The blue arrows still represent the outwards arrows while the green arrow represents the first point in list of nodes for each face.

This formalism is meant to know how we localize nodes. Thus, depending on the configuration we are on, we know if the nodes will be correctly synchronized or not.

For instance, this formalism tells us that in the case of translation periodicity, the use of *shift_geom* function is sufficient to ensure good synchronization.

On the other hand, when we are in a rotational case, this function may not be sufficient. Indeed, faces then aren't in the same plane. As a result, it may be that sorting plan 3.2.3 leads to two different first indices for the two lists A and B.

Thereby, it may be that order of nodes won't match between the two faces, which would lead to a ill association of refinement information during adaptation step.

To prevent this, we will have to make sure that the first nodes for the two concerned faces is the same. A way to do this is to virtually put both faces in the same location, using a ghost face_B (i.e. face_B_ghost). The function *shif_geom*, applied to two faces at same location, will then return the same first nodes. Synchronization will thereby be ensured.

The point will be to rotate one of the two faces only such that they will then be at the same location. The objective then recalls telling one face to apply *shif_geom* function in the frame of the other face.

Then, when we have a rotation periodicity connection, we fix face_A and rotate face_B_ghost around $\underline{x_0}$, of same angle $\underline{\theta}$ than the one of rotation periodicity. Doing so, both faces are now at same location.

This process is done by *axial_rotate* function where we simply apply the following formula, to every nodes of ptList_B_ghost:

$$\begin{pmatrix} x_{1_{new}} \\ x_{2_{new}} \\ x_{3_{new}} \end{pmatrix} = \begin{pmatrix} +cos(\theta) & -sin(\theta) & 0 \\ +sin(\theta) & +cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x_1 - x_0 \\ x_2 - y_0 \\ x_3 - z_0 \end{pmatrix} + \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}$$

Figure 3.21: Rotation formula. Process under *axial_rotate* is automatized to put structure in a framework where rotation is around e_z axis.

This function replaces the previous one which was ill defined and not as general (i.e. it didn't work for a non-zero center point). This function is called in *axial_rotate*, and so we now have both ptList A and ptList_B_ghost at same location.

At this stage if one applies *shift_geom* on the two ptLists, as they are at the same location, they will provide the same first nodes. Thus, we don't change orientation of faces through rotation process, nor the data ; we simply set the same first nodes for every faces of both ptLists.

For ptList B, we simply say to the code to apply *shift_geom* over faces of ptList B in the case where it is in another frame. Then nodes are correctly aligned with respect to ptList A.

Eventually, whenever we have something else than rotation periodicity, we simply apply *shift_geom* function, which is already working. On the other hand, when we deal with rotation periodicity, we include the *axial_rotation* step which allows to set the same first nodes for both ptLists.

The function associated to the discussed process can be found in Appendix C. At this moment, periodicity is applied well, and rotation periodicity has been handled.

3.2.1 Validation rotation periodicity.

Let's consider briefly the three zones case again. This time, we will have both self-join and multi-joins features in the same case.

We want to make sure we can apply a rotation periodicity of 90° and that refinement information is transmitted from a ptList to another.

We then refine two cells of one zone, and we send the mesh (i.e. CGNS tree) to adaptCells code, and we ask for a rotation periodicity of 90° around e_y axis.

Here are the figures showing resulting adapted mesh:

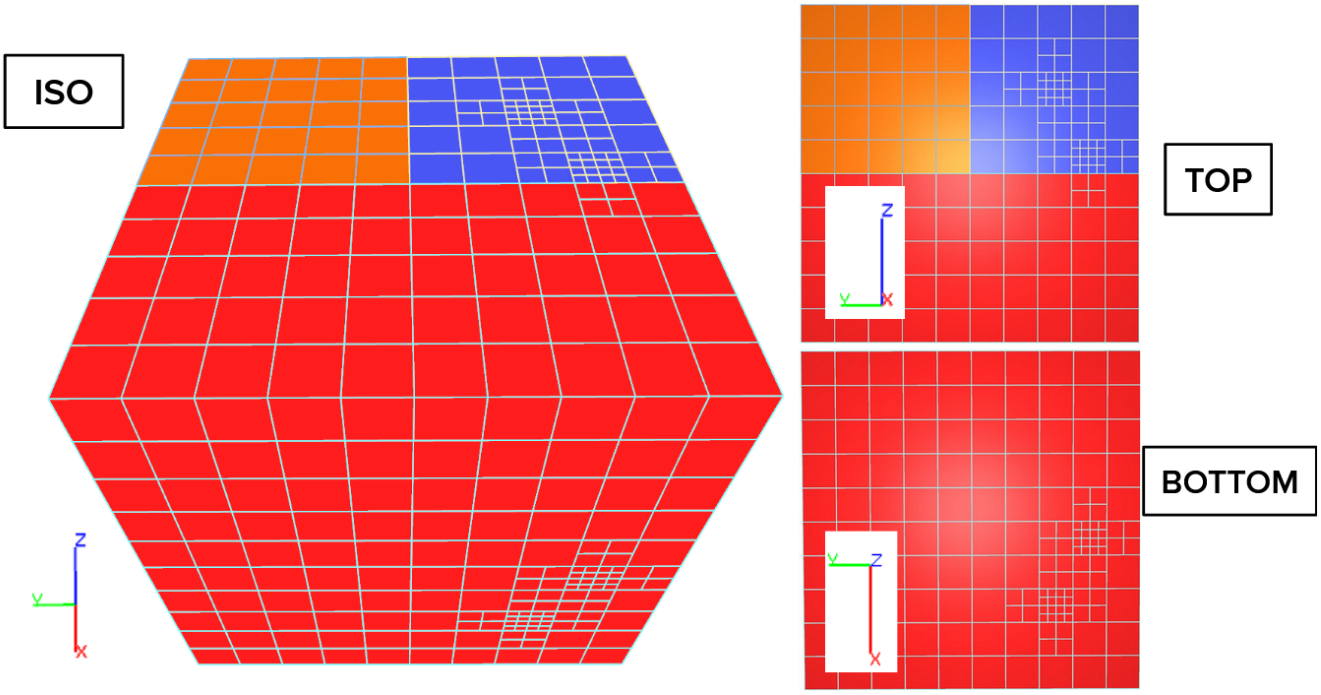


Figure 3.22: 2D views of the three zones case, where one refined locally two cells from bottom face, and refinement propagated to top face through rotation periodicity connection. The rotation periodicity has there been set to have a periodicity of 90° around e_y axis.

Note that what matters in this three zones case is that it gathers properties of self-join and multi-joins cases.

We then have for periodicity connection:

- half-bottom-red face which is connected to top-red face (**self-join**);
- the other half-bottom-red face which is connected to once orange zone and once blue zone.

As a result, we have three periodicity connections for the same zone (red). Internally, we have two connections, between red zone and respectively blue and orange zones.

It implies a zone (red) connected 5 times: 2 times by periodicity and 2 times by match join (with blue and orange zones), plus once by self-join. Red zone thus presents multi-joins between the same pair of zones, as well as self-join.

Once again, this case confirms that our modifications on rid process and self-join case have repercussions over more cases than the ones we tested, confirming its general aspect and its consistency over complex layouts of domains.

3.2.2 Conclusion rotation periodicity

As a conclusion, in rotation periodicity we focused our attention on the good sorting of nodes for adaptation process. After refinement process has been applied, the updated lists will then directly be in good order, thanks to this synchronization step.

The modifications brought here make rotation process more consistent, finalizing our implementation of periodicity property throughout the meshing strategy.

We furthermore automated some processes in *initForAdaptCells* function (not shown here), allowing easier modifications for one to work on rotation periodicity and extend current codes.

Let's now validate periodicity implementation through an intermediate case, the channel model 3.3.

3.3 Validation channel model

The study of channel model 3.3 is a first step to test implemented codes. We developed step by step the periodicity property, which is now ensured throughout the meshing strategy.

In the following, we show the mesh 3.3 after refinement:

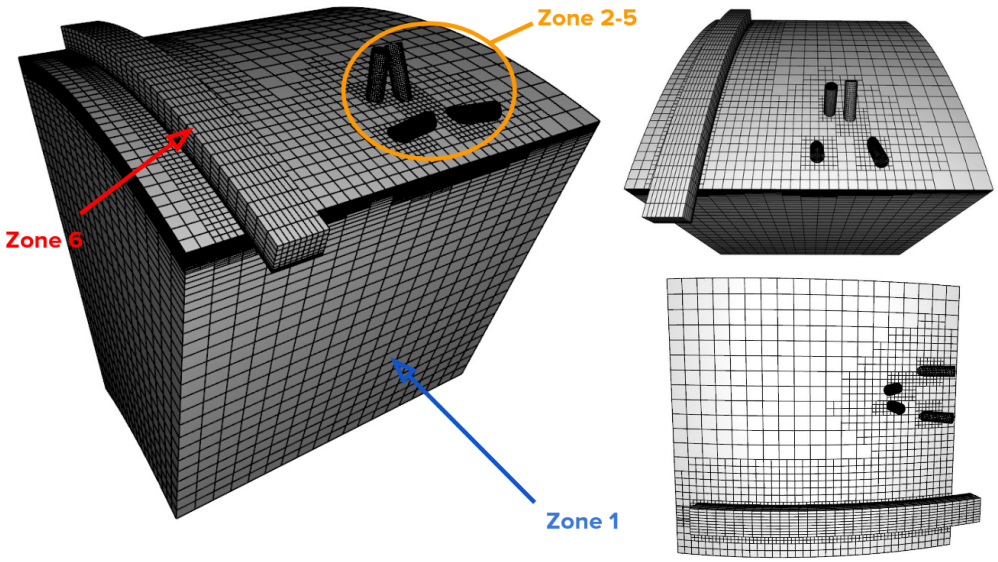


Figure 3.23: Refined mesh associated to model 3.3. 3D views of the channel structure (zone 1) to which one adds 4 cylindrical features (zones 2-5) and a slot (zone 6). Refinement is localized around features, and on some edge as well.

What we observe on fig 3.23 is the adaptation process for channel model 3.3, including the 2:1 smoothing rule. By importing the cylindrical features and the slot, we in fact create a discontinuity in refinement. Then, zone 1 doesn't have the same degree of refinement than zones 2-6 and so adaptation in needed.

Here, adaptation process creates continuity in refinement by refining neighbor cells of features under the 2:1 smoothing rule 3.2.2. Eventually, the mesh stops refining once every zones obey the rule.

In the case of fig 3.23, this refinement occurs over some cells of a periodic face, and rotation periodicity transmits information:

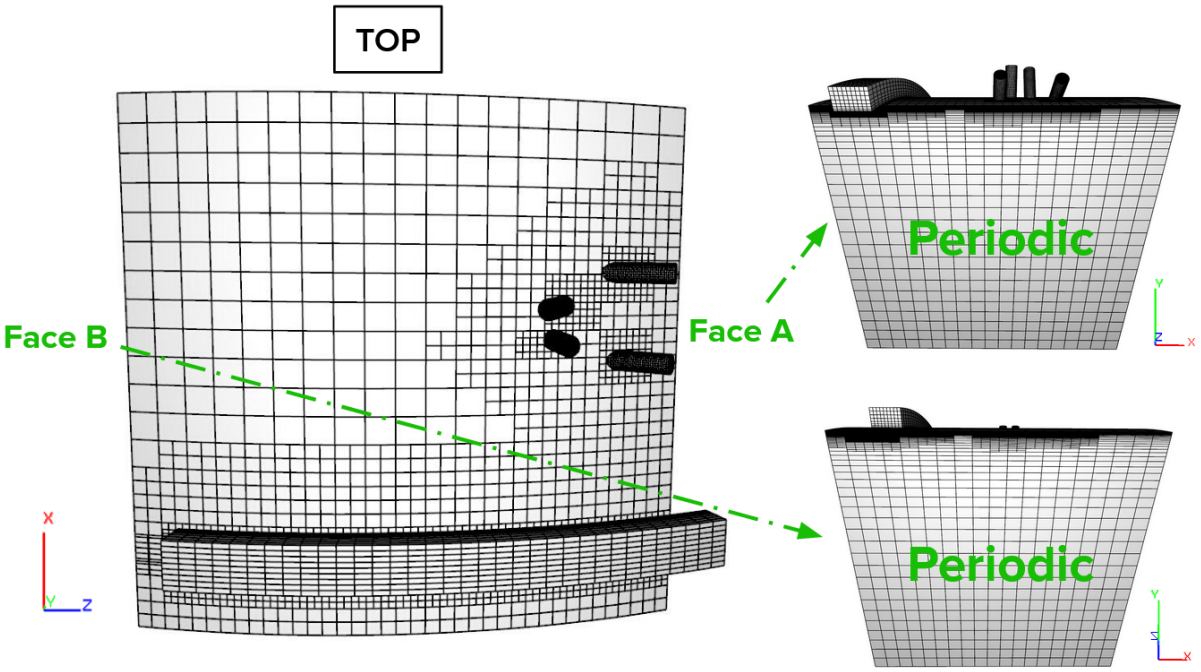


Figure 3.24: 3D views of the refined channel model. We observe refinement on ptList A and rotation periodic connection transfers information to ptList B. Both ptList A and ptList B then are the same within a rotation $\underline{\theta}$.

Figure 3.24 shows that both ptLists A and B are refined the same way. Then, we confirm that now one can refine on edges, while imposing a periodic connection.

Before implementation of our codes, the same study resulted in a **loss of information**. Here is an insight into how Cassiopee deals with periodic faces:

- we initially set let's say a rotation periodicity between ptList A and ptList B of a channel. In CGNS tree, these faces are then labelled as periodic boundaries ;
- at this stage, no refinement occurred and if one generalized from the channel to an entire stage of turbomachinery, it would work well.

  We now consider external features added to the channel, as in fig 3.3;
- as refinement discontinuity is handle by the codes, some cells of a periodic face will refine (let's say on faces of ptList A).

  If it was to happen before we brought our modifications, only the concerned faces of ptList A would be refined, namely the other ones would remain unchanged as no rotation periodicity was possible before;

- in fact, only the common faces between ptList A and ptList B are considered as periodic faces by Cassiopee. Yet, where ptList A has refined faces, ptList B didn't.

  Therefore, the lists of faces accounted for in periodic link are ptLists A and B minus refined faces that are not common faces.

  This implies that we lost the information of some faces in refinement process;

- as a result, if one generalized the channel to an entire stage of turbomachinery at this stage, it would concern only a reduced part of the channel.

  Therefore, if the periodic faces lack information, the resulting turbomachinery stage will be incomplete, and so it will be inconsistent.

By application of the codes we implemented, we solved this issue, and so both faces now refine correctly (i.e. no loss of info).

Let's display a final set of figures highlighting this, showing periodic ptLists saved in Cassiopee tree. These figures display only periodic faces with respect to model 3.23:

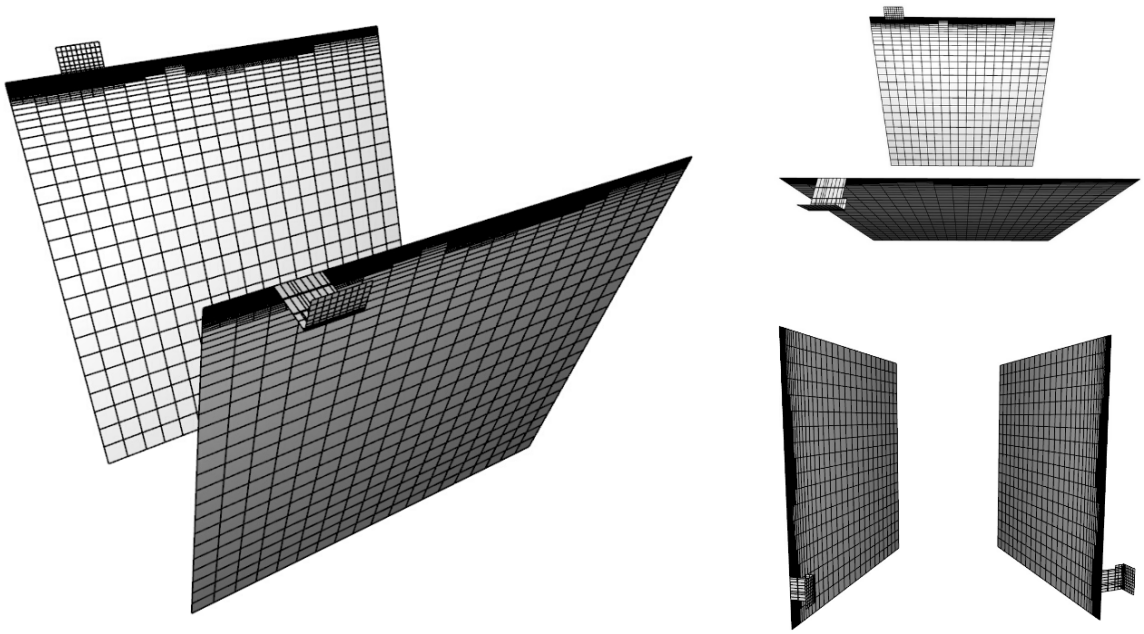


Figure 3.25: 3D views of the two periodic ptLists in the case of model 3.23. Each of these figure represents the same two faces in rotation periodic connection, in the configurations of figure 3.23.

Figures 3.24 show the periodic ptLists for model 3.23. These figures tell that our implementation gives consistent pointLists upon exit, periodic boundaries being well refined (no more loss of information).

Periodic boundaries are therefore well defined and adaptation information is transmitted correctly.

We confirm that now one can refine on one edge, while imposing a periodic connection. More particularly, in fig 3.25 we are in a self-joined case. We then validate our code work in more complex cases than previously tested, assessing their general aspect and efficiency in turbomachinery cases.

Before we move on to a more realistic turbomachinery case, let's make sure our implementations didn't break what was already working before our implementations.

3.4 Conclusion periodicity - Non-regression basis

The validation of our implementations necessitates to go through a non-regression basis.

The point is to run a given list of tests functions, which already exist under Cassiopee. These functions have stored in files the results obtained in previous configuration (i.e. before our implementations).

Thereby, by running these functions, if we get same results, it implies that our implementations are still consistent with other cases like translation periodicity case with one connection per zone (case 3.4), etc.

The point simply is to make sure that our improvements didn't break a feature which was already working.

Once we run these test functions, we observe good result, namely our implementations are consistent with what was previously running under Cassiopee.

To conclude on this non-regression basis, given our codes are consistent, **we created a new test function** which includes rotation periodicity. This code is a simple test but it must be fulfilled by anyone implementing new modifications in adaptation process.

Our codes have thereby been tested over new and former configurations, and they work well in both cases. Our implementations now have been thoroughly validated, so let's consider a real turbomachinery case.

Chapter 4

Numerical simulation of a turbomachine channel - CM2012

The validation of this report will be done through CM2012 turbomachinery case, for which we will implement a more physical source of refinement.

We will test a rotation periodicity in this industrial case to demonstrate the well-doing of our codes in real conditions.

The well doing of this case will conclude on our report and the implementation of periodic-boundary-preserving capability under adaptCells function of Cassiopee.

Contents

4.1 Introduction CM2012 case

The turbomachinery CM2012 case is a mesh of a rotor blade, surrounded by a mesh representing a flow:

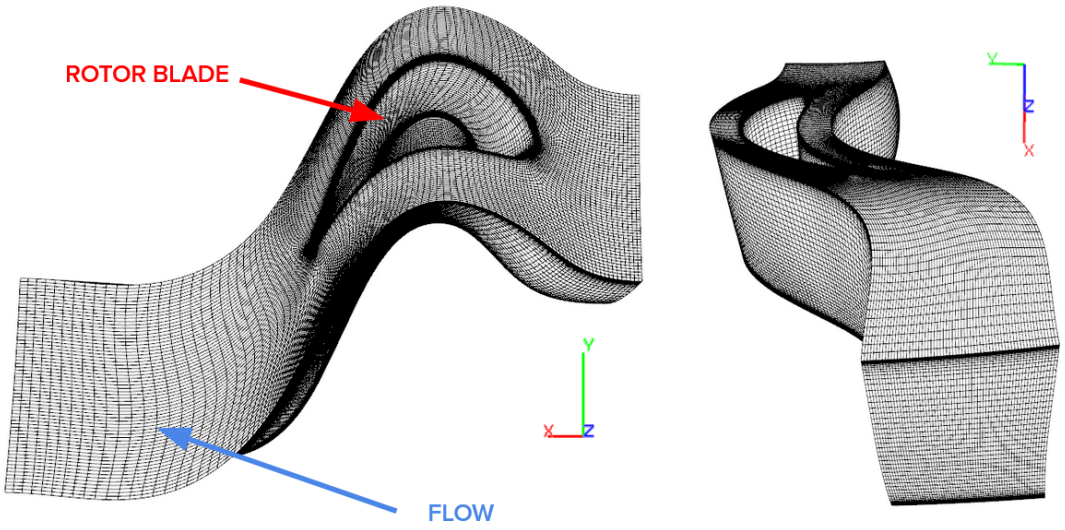


Figure 4.1: Mesh of the turbomachinery case CM2012. This mesh is made of a unique zone and so it's concerned by implementation of self-joining.

This self-join case being used in practical cases, it's necessary to make a few physical information known to CGNS tree.

Here is the CGNS tree of the CM2012 together with some highlight of the features in it:

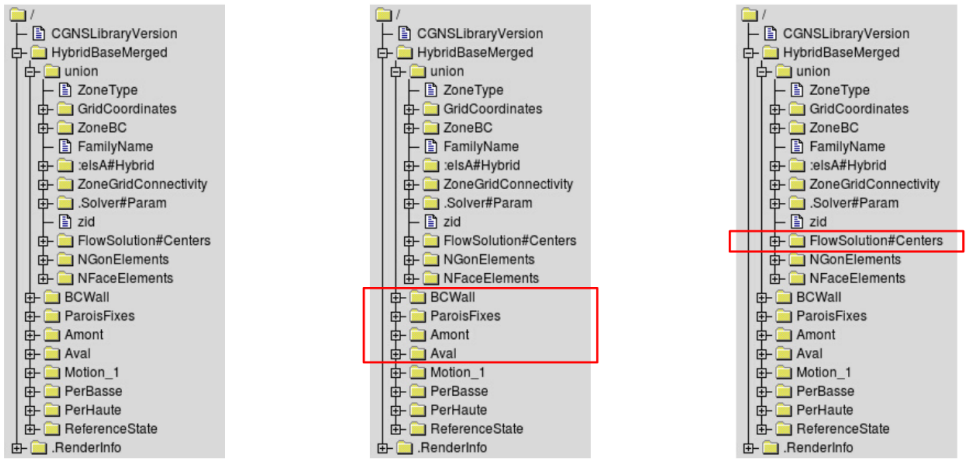


Figure 4.2: CGNS tree of the CM2012 case, where we highlight the labels in it, which helps building a more physical structure.

Here, one associates to each boundary a physical label such as inflow, outflow, periodic, wall, etc (see center figure 4.2). These labels in fact translate what is done numerically.

We then build the mesh with some physical properties, which includes the flow counterpart. As part of the post-processing step, information concerning the flow are saved in the CGNS tree (e.g. density, pressure field, mach field, etc).

This information is labeled under `FlowSolution#Centers` in right figure 4.2. As the label tells, this file will contain flow solutions, calculated at faces' center. Mesh properties are then saved in tree from code and one can access them anytime he/she wants.

In our case, we would like to consider a physical source for the local refinements. To this aim, we will consider a refinement sensor based on a shock-flow through the CM2012 channel.

4.2 Implementation shock

To begin with, we must define an analytic field to describe a step function that would model a shock that goes through the channel.

The function we considered to this aim is a 3D Gaussian function. Besides, we included a rotation property in the function, in order to orientate the shock as we wanted.

Before we furthermore describe the shock function, here is how one can add a flow variable to the tree in the first hand:

- this process is done using the *initVars* function of adaptCells;
- it takes as input: the function of the flow variable (e.g. *shock_function*(x_1, x_2, x_3)), the input of the function (i.e. x_1, x_2, x_3 coordinates) and the location of information.

  Concerning this last point, one can define a flow variable either on nodes or in face's barycenter;
- in our case we define the shock variable at faces' barycenter, which numerically translates by:

```
CGNS_tree = Converter.initVars(CGNS_tree, 'centers:shock', shock_function,
  ['centers:CoordinateX', 'centers:CoordinateY', 'centers:CoordinateZ'])
```

This line is part of the pre-processing step, and it recalls initializing a flow variable, known as shock, and which will take values at faces' center.

The shock function we used in 4.2 is the following one:

```
def shock_function(x1, x2, x3):

    """ Function returning 3D gaussian
    function for shock simulation. """

    # ---------------- Initialization ------------------

    A = 1. # amplitude
    x0, y0, z0 = 35., -19., 220. #center rotation

    sig_1, sig_2, sig_3 = 3., 200., 3. #variance

    # rotation gaussian --> transversal shock
    angle = +90
    angle*= numpy.pi/180 #degree --> radian

    # ---------------- Rotation ------------------

    a = numpy.cos(  angle)**2/(2*sig_1**2) + numpy.sin(  angle)**2/(2*sig_2**2)
    b = numpy.sin(2*angle)**2/(2*sig_1**2) - numpy.sin(2*angle)**2/(2*sig_2**2)
    c = numpy.sin(  angle)**2/(2*sig_1**2) + numpy.cos(  angle)**2/(2*sig_2**2)

    # ---------------- Gaussian function ------------------

    shock = A . exp(-(a*(x1 - x0)**2 + b*(x1 - x0)*(x2 - y0) + c*(x2 - y0)**2 +
    (x3 - z0)**2 / (2*sig_3**2)))

    return shock
```

Listing 4.1: Code defining the shock function, based on the formula of 3D Gaussian function. This code models a Gaussian function for the simulation of a shock. The coordinates and rotation properties appearing here are based on data of the CM2012 mesh 4.1.

Once shock has been applied, we get flow solution as an output. One can then superpose the shock field 4.1 to the mesh 4.1 to visualize the shock in terms of amplitude and localisation with respect to CM2012's mesh:

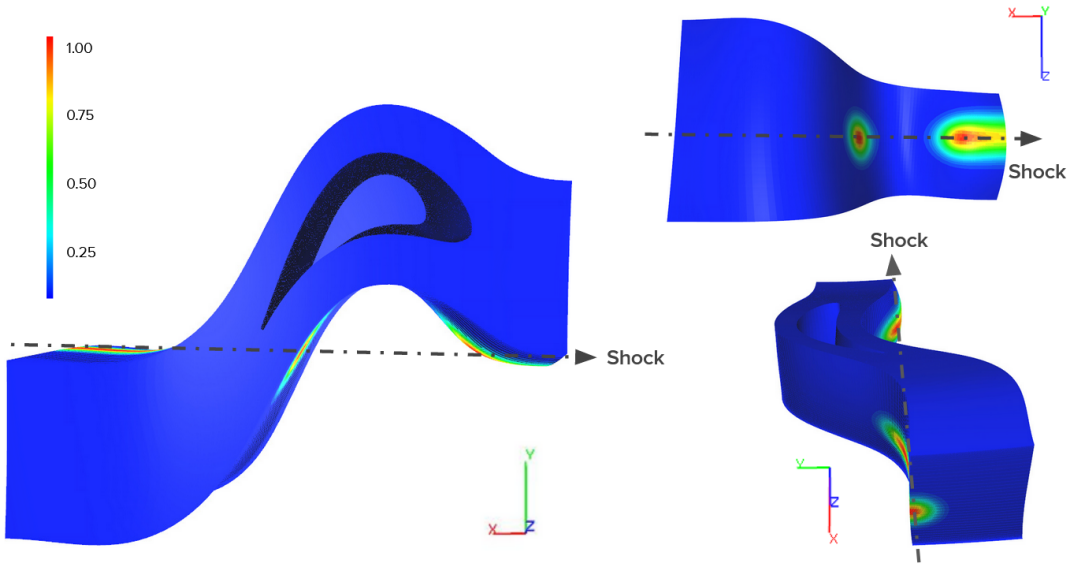


Figure 4.3: 3D figures of the CM2012 case, where one superposed a colored field to the mesh, telling one about the amplitude of the shock along the mesh. The color code is such that the redder, the higher the shock there, and the bluer the lower the shock.

From fig 4.3 we see that one deals with a shock going through the entire mesh. There are locations where the shock-flow is steady (blue-colored parts of the mesh), and others where shock-flow is unsteady (red-colored parts of the mesh).

As mentioned before, we will now base a refinement sensor on this shock function 4.1. This implies to tell the code to refine locally, as a function of amplitude of shock-flow.

In fact, to refine a mesh makes sense only if one wants to improve quality of recovered data there. Therefore, one won't refine where the flow field is steady, but rather where it's unsteady.
In this sense, we want refinement to occur where shock culminates while no much refinement will be needed elsewhere. This condition will constitute our refinement scheme.

Here below is the condition we imposed to retranscribe values of shock into sensor conditions:

```
import Intersector.Mpi as XORMPI
import Converter.Distributed as D
import Converter.Internal as I

# ------------- initialize degree refinement ---------------

REF_1 = REF_LEV_1
REF_2 = REF_LEV_2
REF_3 = REF_LEV_3

M = max(shock)

# ---

zones = I.getZones(CGNS_tree)
sensor_data = []
k=-1
for zone in zones:
  k+=1
  n = C.getNCells(I.getZones(zone))
  sensor = numpy.empty((n,), dtype=numpy.int32)
  sensor[:]=0

  # ---------------- apply refinement scheme -----------------

  if k == 0 and Cmpi.rank == 0:
    for i in range(n):

      if shock[i] > 0.25*M and 0.75*M >= shock[i]:
        sensor[i] = REF_1
      elif shock[i] > 0.75*M and 0.95*M >= shock[i]:
        sensor[i] = REF_2
      elif shock[i] > 0.95*M:
        sensor[i] = REF_3
  sensor_data.append(sensor)

  # --------------- adaptation process -----------------

adapted_CGNS_tree = XORMPI.adaptCells(CGNS_tree, sensor_data, procDict=
procDico, zidDict=zidDico)
```

Listing 4.2: Code translating amplitude of transversal shock into a refinement scheme for adaptCells function. We here set a refinement scheme based on three degrees of refinement, from finer mesh where shock amplitude is bigger to coarser mesh where amplitude is lower.

Note that we here set the degrees of refinement as: $REF\_1 = 1$, $REF\_2 = 2$ and $REF\_3 = 3$. We then have a refinement scheme made of three steps, which we assume is enough to accurately recover information around the shock, without creating too much points:

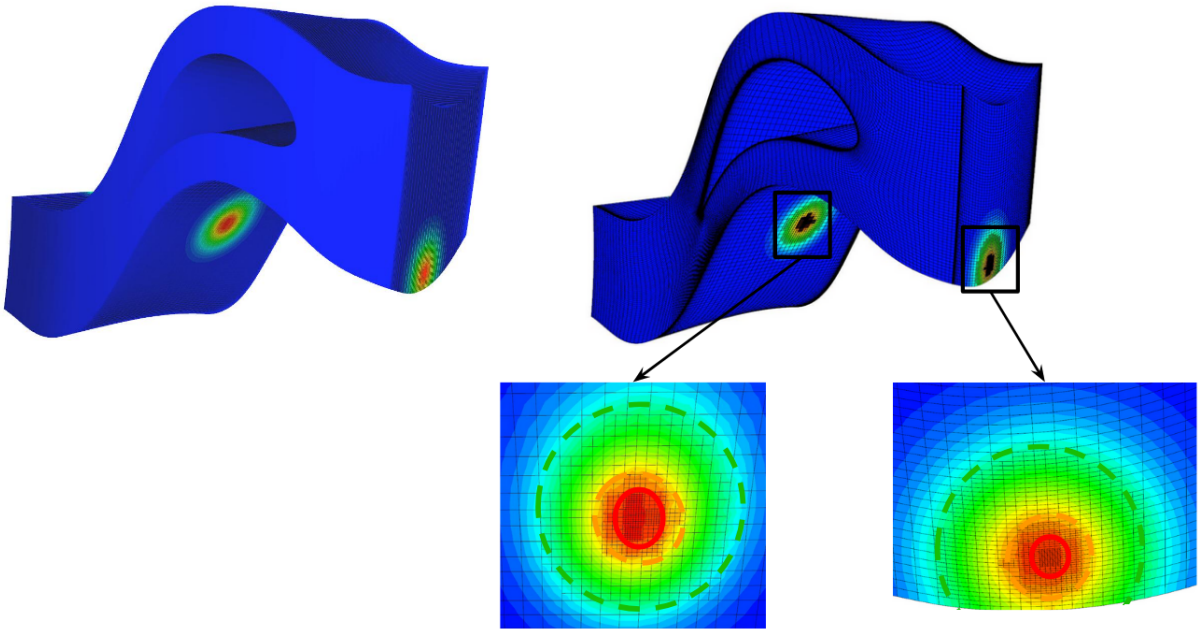


Figure 4.4: 3D graph of the CM2012 case once shock-flow solution has been translated into refinement levels. Left figure represents CM2012 case with flow-shock only, while right figures tell about how mesh is refined as a function of shock's localisation.

In figure 4.4 we observe how we translate input flow-shock into several levels of refinement. We thus note three level of refinement used to refine around the shock. We furthermore note that we respect the 2:1 smoothing rule by application of 4.2. Finally, no refinement occurs on static field (blue).

At this stage, we went from the turbomachinery mesh 4.1, and we applied a shock function to it, which directly influences the refinement scheme.

We then send mesh with flow variable and refinement scheme to adaptCells and wait for adaptation to be handled (with rotation periodicity).

The following section shows results of the simulation.

4.3 Validation periodicity

In the following figures, we gather the 3D views of the turbomachinery case 4.1 after refinement and periodicity occurred. We especially highlight refinement due to shock flow, and refinement from periodicity property:

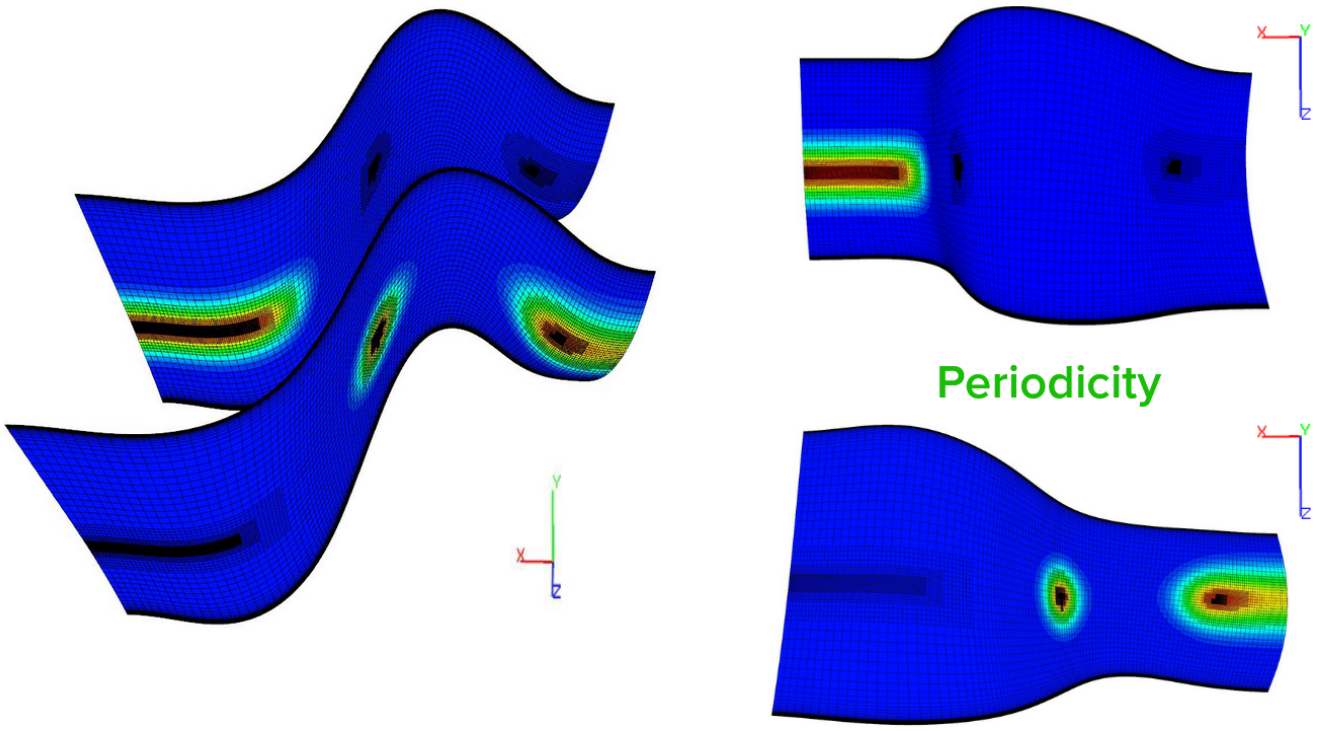


Figure 4.5: Figures of the CM2012 after periodicity has been applied. In this case, shock function led to local refinement and periodicity transmitted information between the two faces.

These figures result from the refinement scheme 4.2 applied to mesh 4.1, where rotation periodicity has been applied.

The refinement we observe then comes from:

- direct application of refinement scheme over shock function's peaks (warm-colored unsteady field);
- rotation periodicity transmitting information of refinement from one end to the other (blue-colored steady field).

Ultimately, these two steps concern the same process where periodic-conserving-boundary capability is ensured, in the case of a turbomachinery case.

This case concludes once again on the consistency of the code we implemented, as we here consider rotation periodicity in a self-joined case.

With the improvements described in this report, the adaptation process can now handle cases that were not working at first.

4.4 Conclusion

To conclude on this report, the periodicity property of the meshing strategy wasn't consistent enough for turbomachinery applications. Mesh adaptation process wasn't preserving the periodicity capability when one refined on the periodic boundaries of a channel.

Through a step-by-step process, we have handled this issue at every stages of the meshing strategy, extending and fixing adaptCells code and its python interface.

Our implementations eventually showed great consistency with both newly-implemented-cases and already-existing-cases, in several turbomachinery cases. We therefore expanded the non-regression basis with a test of our own.

Finally, the codes developed during this internship will soon be tested in CFD simulations of turbomachinery and aerodynamic systems, by researchers of DEFI team.

There exists as well several paths of improvements for one to further enhance the current adaptation process, among which one can mention the parallelization of match connection and the extent of conforming step to the multi-zones case.

4.5 Personal balance sheet

In a more personal point of view, the five months I have been within DEFI research team have been of great interest to witness and be part of constantly evolving projects such as Cassiopee.

More specifically, the processes of updates and commits within a software shared by hundreds of research engineers has been an opportunity for me to understand in real conditions the steps undergone to validate modifications and share it with others.

The diversity of contact and means of exchange put at my disposal have really helped me to have a global idea of the work done by others, where our codes stand with respect to others and how one can be part of a general system at its level.

I hereby thanks again the entire DEFI team for offering me the opportunity to work over such crucial numerical implementations, and be part of the great project Cassiopee.

Appendices

Appendix A

Self-joined case - Cassiopee source code

In complement of the concatenation code 3.1, there is a decomposition part which has been done in *adaptor_ mpi.hxx* function, a header of adaptCells function. In the following code, we implemented from line 30 to the end to deal with the particular self-joined case:

```
1     for (auto& r : rid_to_Face_to_plan)
      {
3       // ---------------- Set flags of zones / connections ------------------

5       int rid = r.first;
        int jzid = get_opposed_zone(rid_to_zones, rid, zid);
7
        //get the opposed list of faces
9       const auto itopp = zone_to_rid_to_list.find(jzid);
        assert(itopp != zone_to_rid_to_list.end());
11
        const auto itptl = itopp->second.find(rid);
13      assert(itptl != itopp->second.end());
        const auto& ptlist = itptl->second;
15
        auto & Face_to_plan = r.second;
17
        // ---------------- If usual case ------------------
19
        if (jzid != hmeshes[i]->zid)
21      {
          for (auto & k : Face_to_plan)
23        {
            E_Int Face_i = ptlist[k.first] -1;    //list of faces
25          sensor_data[jzid][Face_i] = k.second; //info refinement
          }
27      }

29      // ---------------- If self-joined case ------------------

31      else
        {
33        int size_ptList = ptlist.size() / 2;

35        for (auto& k : Face_to_plan)
          {
37          // refinement schemes
            int key1 = k.first; // 1st Plan
39          int key2 = k.first + size_ptList // 2nd Plan

41          // Plan 1
            E_Int Face_i = ptlist[key1] -1; // Face on which we test plan 1
43
            // Plan 2
45          E_Int Face_j = ptlist[key2] -1; // Face on which we test plan 2
```

```
47          // ---------- //

49          sensor_data[jzid][Face_j] = k.second;       // zone to refine - Plan 1

51          const auto Plan2 = Face_to_plan.find(key2);
            if (Plan2 == Face_to_plan.end()) continue;
53          sensor_data[jzid][Face_i] = Plan2->second;  // zone to refine - Plan 2
          }
55      }
    }
```

Listing A.1: Code decomposing input concatenated list of faces. We read each connection and when we have a self-join case, we deal with else condition, the one we implemented.

This code A.1 is a reduced part of the decomposition code that we implemented to tackle the self-joined case. More specifically, this code deals with decomposition of input lists of faces to apply correctly refinement schemes to each list of faces.

Note that part of this function has been added afterwards (once we did every implementations) for consistency of the code.

Appendix B

Multi-bounds - rid process

The point of rid flag is to flag each connection that we have for each pair of zones. To the criteria already mentioned for the creation of rid flags (index of zone, index of min ptList, etc) one must add the criterion *idx_perio*. In fact, the way mesh is sometimes generated can be an issue in multi-joins case.

Let's consider a regular domain of size $(4 \times 4 \times 4)$. What may in fact take place during the creation of this mesh (pre-processing step) is that Cassiopee will create a first zone of size $(2 \times 4 \times 4)$ and then duplicate it. Thereby, the resulting mesh will have expected size, expected number of zones, except it will be created from same zone.

This implies that, even though face numbering is local, the two zones will have exact same face numbering (i.e. $PointList = PointListDonor$). The issue then is that we will be left with two connections, between the same zones and for which list of faces (and so min_idx) are the same. This leaves us with no condition to differentiate the two connections.

What we must do then is to add another criterion to know if we have a new connection. In fact, the issue exposed here can happen only if one has a match connection and a periodic connection. Thus, the condition will be to know if the connection concerns a periodic connection or not.

This last test over the connection ensures to account for every bounds in the multi-joins case. We then have a unique rid for every bound in every back and forth connections.

Note that the use of min_index remains necessary. Indeed, there can theoretically be an arbitrary number of connections between 2 zones. It's then possible to have several inner match bounds at different locations (i.e. different indices of faces) together with periodic connections.

The following code gathers main steps in the creation of rid tag. The point for one going through this code is especially to go through the *if ... else ...* conditions and measure the number of criteria considered in order to account for every possible connections (as opposed to no condition for previous zid formalism):

```
import Converter.Internal as Internal
import numpy

def set_rid(CGNS_tree):

  # ------------------------- Initialization ----------------------------

  zones = Internal.getZones(CGNS_tree)
  idx_connec = -1

  dict_save = {} #dict type variable

```

```
    # ---------------------------- rid process -------------------------------

    for zone in zones:
    bounds = Internal.getNodes(zone, 'GridConnectivity_t')

      for bound in bounds:
        zone1 = getProperty(zone, 'zid')
        list_faces_1 = Internal.getNode(bound, zone1, 'List_Faces')

        zone2 = zidDict[bound]
        list_faces_2 = Internal.getNode(bound, zone2, 'List_Faces')

        # ---

        idx_min = numpy.minimum(min(list_faces_1), min(list_faces_2))

        # -------- Construction of rid flags -------

        if zone1 in dict_save:
          if zone2 in dict_save[z1]:
            if idx_perio in dict_VD[z1][z2]:
              if idx_min in dict_save[z1][z2]:
                # save rid to tree
                connec = Internal.newIntegralData(name='rid', parent=bound)
                connec[1] = dict_save[z1][z2][idx_perio][idx_min]
              else:
                idx_connec += 1 dict_VD[z1][z2][idx_perio][0].append(idx_min)
                dict_VD[z1][z2][idx_perio][1].append(idx_connec)

                # save rid to tree
                connec = Internal.newIntegralData(name='rid', parent=bound)
                connec[1] = idx_connec
            else:
              idx_connec += 1
              dict_VD[z1][z2][idx_perio] = [[idx_min], [idx_connec]]

              # save rid to tree
              connec    = Internal.newIntegralData(name='rid', parent=rac)
              connec[1] = idx_connec
          else:
            idx_connec += 1

            dict_save[z1][z2] = {}
            dict_save[z1][z2][idx_perio] = [[idx_min], [idx_connec]]

            # save rid to tree
            connec = Internal.newIntegralData(name='rid', parent=bound)
            connec[1] = idx_connec
        else:
          idx_connec += 1

          dict_save[z1] = {}
          dict_save[z1][z2] = {}
          dict_save[z1][z2][idx_perio] = [[idx_min], [idx_connec]]

          # save rid to tree
          connec = Internal.newIntegralData(name='rid', parent=bound)
          connec[1] = idx_connec
}
```

Listing B.1: Definition of rid flags, which associates a unique connection index to each connection made through a different face. This code is only simplified a bit, the point being to let it as it has been implemented to retranscribe accurately the degree of condition for one to consider every possible connections for the creation of rid.

This code creates a temporary dictionary dict_save which helps creating the unique rid tag for each connection, which is then available directly from CGNS tree. All these manoeuvres have

for objective to prevent code from neglecting and overwriting some connections. In code B.1, we associates a rid to every back and forth link in process of exchange of refinement schemes.

Appendix C

Rotation periodicity - *InitForAdaptCells* function

The function that follows has been built during this internship in order to externalize *shift_ geom* process, and extend it to the case of rotation periodicity:

```
PyObject* K_INTERSECTOR::initForAdaptCells(PyObject* self, PyObject* args)
{

  // --------------------------- Initialization ------------------------------

  // 1. Get coords mesh
  FloatArray & crd = *f;

  // NGON structure 3.17
  ngon_type NGON;

  // reorient_skin function
  NGON.flag_externals(1);
  reorient_skins(crd, NGON); //normal outwards

  // ------------- Loop over connections (match, transla, rota) -----------

  for (auto& key : key_list)
  {
    // -------- Data rotation periodicity -------

    auto & rota_data = key.first;
    auto & ptList  = key.second;

    int size_ptList = ptList.size();

    for (E_Int i = 0; i < 3; ++i)
    {
      E_Float center_rota[i] = rota_data[i];
      E_Float axis_rota[i] = rota_data[i+3];
    }

    E_Float angle = normalize(axis_rota);
    angle *= PI / 180; //degree --> radian

    // -------- Apply shift_geom & axial_rotate -------

    if (angle != 0.) //rotation periodicity
    {
      auto crd_ghost = crd; //==ptList_B_ghost

      axial_rotate(crd_ghost, center_rota, axis_rota, angle);

```

```
            for (E_Int i=0; i < size_ptList; i++)
45          {
              shift_geom(crd_ghost, nodes, NB_nodes, 1);
47          }
          }
49
          else //translation, match connections
51        {
            for (E_Int i=0; i < size_ptList; i++)
53          {
              shift_geom(crd, nodes, NB_nodes, 1);
55          }
          }
57      }

59      PyObject* m = buildArray(crd, NGON, 8, "NGON", false);

61      return m;
    }
```

Listing C.1: Function *initForAdaptCells* built as part of rotation periodicity implementation. We here display a simplified version of the code, mainly to improve its readability and to be in agreement with previously discussed codes.

References

A.Dugeai, A.Placzek, Y. Mauffrey and S. Verley. Overview of the Aeroelastic Capabilities of the elsA Solver within the Context of Aeronautical Engines. *AerospaceLab Journal*, 14, Sep. 2018. ISSN 2107-6596. doi:10.12762/2018.AL14-03.

Airbus Defence & Space, ONERA. L'ONERA et Airbus Defence & Space signent un accord de partenariat. downloadable link, Jun. 2018.

Airbus, DLR, ONERA. L'ONERA signe un partenariat stratégique avec Airbus et le DLR. downloadable link, Jun. 2017.

F. Alauzet, L. Frazza and D. Papadogiannis. Periodic adjoints and anisotropic mesh adaptation in rotating frame for high-fidelity RANS turbomachinery applications. *Journal of Computational Physic*, 450:p. 110814, 2022. ISSN 0021-9991. doi:10.1016/j.jcp.2021.110814.

G. Angelini. *Machine learning in Industrial Turbomachinery: development of new framework for design, analysis and optimization.* Ph.D. thesis, Sapienza, Università di Roma, downloadable link, 2019.

C. Benoit, S. Péron and S. Landier. Cassiopee: A CFD pre- and post-processing tool. *Aerospace Science and Technology*, 45:pp. 272–283, Sep. 2015. doi:10.1016/j.ast.2015.05.023.

J. Boustani, G. Anugrah, M. Barad, C. Kiris and C. Brehm. A Numerical Investigation of Parachute Deployment in Supersonic Flow. AIAA Scitech 2020 Forum, Jan. 2020. doi:10.2514/6.2020-1050.

P. M. Cali, V. Couaillier and A. Jameson. Conservative Interfacing for Turbomachinery Applications. *American Society of Mechanical Engineers*, 1, Jun. 2001. doi:10.1115/2001-gt-0357.

L. Cambier, M. Gazaix, S. Heib, S. Plot, M. Poinot, J.-P. Veuillot, J.-F. Boussuge and M. Montagnac. An Overview of the Multi-Purpose elsA Flow Solver. *Aerospace Lab*, 2:p. 2, Mar. 2011.

CGNS. https://github.com/CGNS/.

X. H. Chang, R. Ma, N. H. Wang and L. P. Zhang. Parallel Implicit Hole-cutting Method for Unstructured Chimera Grid. *Tenth International Conference on Computational Fluid Dynamics*, 198:p. 104403, Feb. 2020. doi:10.1016/j.compfluid.2019.104403.

A. Demeulenaere and R. V. den Braembussche. Three-Dimensional Inverse Method for Turbomachinery Blading Design. *Journal of Turbomachinery*, 120(2):pp. 247–255, Apr. 1998. doi:10.1115/1.2841399.

J. D. Denton. The aerodynamics of turbomachinery. *Science progress, Oxford*, 74(4):pp. 443–463, 1990.

L. Diazzi and M. Attene. Convex polyhedral meshing for robust solid modeling. *ACM Transactions on Graphics*, 40(6):pp. 1–16, Dec. 2021. doi:10.1145/3478513.3480564.

E. H. Dowell, R. M. Bennett, H. C. Curtiss, R. H. Scanlan and F. Sisto. A Modern Course in Aeroelasticity. *Journal of Applied Mechanics*, 49(2):pp. 465–466, 1982. doi:10.1115/1.3162177.

R. G. Du Toit, D. H. Diamond and P. S. Heyns. A stochastic hybrid blade tip timing approach for the identification and classification of turbomachine blade damage. *Mechanical Systems and Signal Processing*, 121:pp. 389–411, Apr. 2019. doi:10.1016/j.ymssp.2018.11.032.

J. Dubois. Article fondation de l'onera. download link, Jun. 1966.

A. Dugeai, Y. Mauffrey, A. Placzek and S. Verley. Overview of the Aeroelastic Capabilities of the elsA Solver within the Context of Aeronautical Engines. *AerospaceLab Journal*, 14, Sep. 2018. doi:10.12762/2018.AL14-03.

S. Fleeter, C. Zhou, E. N. Houstis and J. R. Rice. Fatigue Life Prediction of Turbomachine Blading. *Purdue e-pubs*, Paper 1460, Sep. 1999.

C. Gueuzaine and J.-F. Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):p. 1309–1331, May 2009. doi:10.1002/nme.2579.

J. Hammond, N. Pepper, F. Montomoli and V. Michelassi. Machine Learning Methods in CFD for Turbomachinery: A Review. *International Journal of Turbomachinery, Propulsion and Power*, 7(2):p. 16, May 2022. doi:10.3390/ijtpp7020016.

J. Carpentier and K. Dang-Tran. 50 ans de recherches aéronautiques et spatiales. downloadable link, 1997.

K.-L. Jeong and D.-W. Seo. Automatic polyhedral mesh generation for ship resistance based on the locally refined cartesian cut-cell method. *Journal of marine science and technology*, 28(4), 2020. doi:10.6119/JMST.202008_28(4).0003.

J. R. Koch, E. Dumlupinar, J. A. Housman, C. C. Kiris, M. M. Patel, B. Kleb, G. J. Brauckmann and S. J. Alter. CFD Analysis of Space Launch System Solid Rocket Booster Separation within the Langley Unitary Plan Wind Tunnel. AIAA Aviation 2021 Forum, Jul. 2021. doi: 10.2514/6.2021-2966.

S. Landier. Boolean operations on arbitrary polygonal and polyhedral meshes. *Computer-Aided Design*, 85:pp. 138–153, Apr. 2017. doi:10.1016/j.cad.2016.07.013.

J. Liang, J. Bai and G. Li. Investigation of Stall Flutter Based on Peters-ONERA Aerodynamic Model. *Journal of Northwestern Polytechnical University*, 36(5):p. 875–883, Oct. 2018. doi: 10.1051/jnwpu/20183650875.

C. Lienard, R. Boisard and C. Daudin. Aerodynamic behavior of a floating offshore wind turbine. *AIAA Scitech 2019 Forum*, Jan. 2019. doi:10.2514/6.2019-1575.

E. Logan and R. Roy. Handbook of Turbomachinery. *CRC Press*, 158, 2003. doi:10.1201/9780203911990.

J. Mayeur, A. Dumont, D. Destarac and V. Gleize. Reynolds-averaged navier-stokes simulations on NACA0012 and ONERA-M6 wing with the ONERA elsA solver. *AIAA Journal*, 54(9):pp. 2671–2687, Sep. 2016. doi:10.2514/1.J054512.

X. Merle, P. Cinnella and G. Dergham. Modélisation par Machine Learning d'écoulements turbulents dans les turbomachines. downloadable link, 2021.

D. C. Mincu, T. L. Garrec, S. Peron and M. Terracol. Immersed boundary conditions for high order CAA solvers - Aeroacoustics installation effects assessment. *23rd AIAA/CEAS Aeroacoustics Conference*, Jun. 2017. doi:10.2514/6.2017-3504.

Naval Group, ONERA. Intelligence artificielle embarquée : Naval Group signe un nouveau partenariat avec l'ONERA. downloadable link, Feb. 2021.

ONERA researchers. 1984 à 1996 - les nouvelles orientations. web link, 2022. Last accessed on 11-July-2022.

K. Pahlke, J. Sidès and M. Costes. Numerical simulation of flows around helicopters at DLR and ONERA. *Aerospace Science and Technology*, 5(1):pp. 35–53, Jan. 2001. ISSN 1270-9638. doi:10.1016/S1270-9638(00)01078-6.

G. Persico, P. Rodriguez-Fernandez and A. Romei. High-Fidelity Shape Optimization of Non-Conventional Turbomachinery by Surrogate Evolutionary Strategies. *Journal of Turbomachinery*, 141(8), Apr. 2019. doi:10.1115/1.4043252.

S. Péron. A Review of Overset Grid Technology at ONERA. *13th Overset Grid Symposium*, Oct. 2016.

S. Péron and C. Benoit. Automatic off-body overset adaptive Cartesian mesh method based on an octree approach. *Journal of Computational Physics*, 232(1):pp. 153–173, Jan. 2013. doi:10.1016/j.jcp.2012.07.029.

S. Rehman, M. M. Alam, L. M. Alhems and M. M. Rafique. Horizontal Axis Wind Turbine Blade Design Methodologies for Efficiency Enhancement—A Review. *Energies*, 11(3):p. 506, Feb. 2018. doi:10.3390/en11030506.

T. Renaud, S. Landier and S. Péron. New CFD capabilities based on intersecting arbitrary polyhedral meshes. *AIAA Aerospace Sciences Meeting*, Jan. 2018. doi:10.2514/6.2018-1498.

J. Reneaux, P. Beaumier and P. Girodroux-Lavigne. Advanced Aerodynamic : applications with the elsA Software. *Aerospace Lab*, 2:pp. 1–21, Mar. 2011.

M. ROY. Means and Examples of Aeronautical Research in France at ONERA. *Les nouvelles de l'ONERA*, 26(4), Apr. 1959. doi:10.2514/8.8013.

A. Rubino, S. Vitale, P. Colonna and M. Pini. Fully-turbulent adjoint method for the unsteady shape optimization of multi-row turbomachinery. *Aerospace Science and Technology*, 106:p. 106132, Nov. 2020. ISSN 1270-9638. doi:10.1016/j.ast.2020.106132.

Salome. http:http://www.salome-platform.org.

L. Sbardella, A. I. Sayma and M. Imregun. Semi-structured meshes for axial turbomachinery blades. *International Journal for Numerical Methods in Fluids*, 32(5):pp. 569–584, Mar. 2000. doi:10.1002/(sici)1097-0363(20000315)32:5<569::aid-fld975>3.0.co;2-v.

V. Schmitt and F. Charpin. Pressure Distributions on the ONERA-M6-Wing at Transonic Mach Numbers. *Report of the Fluid Dynamics Panel Working Group 04*, May 1979.

C. Tailliez and A. Arntz. CFD Assessment of the Use of Exergy Analysis for Losses Identification in Turbomachine Flows, Mar. 2018. 53rd International Conference of Applied Aerodynamics, French Air Force Academy at Salon-de-Provence, France.

P. G. Tucker and Z. Ali. Multiblock Structured Mesh Generation for Turbomachinery Flows. *Proceedings of the 22nd International Meshing Roundtable*, pp. 165–182, 2014. doi:10.1007/978-3-319-02335-9_10.

J. Tyacke, N. R. Vadlamani, W. Trojak, R.Watson, Y.Ma and P. G. Tucker. Turbomachinery simulation challenges and the future. *Progress in Aerospace Sciences*, 110:p. 100554, Oct. 2019. doi:10.1016/j.paerosci.2019.100554.

ULIS | Sofradir, ONERA. Accord de partenariat entre l'ONERA et la société ULIS. downloadable link, 2016. Accessed on 2022-06-27.

B. Wang, G. Wang, K. Tian, Y. Shi, C. Zhou, H. Liu and S. Xu. A preliminary design method for axisymmetric turbomachinery disks based on topology optimization. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 236(7):p. 3313–3322, Apr. 2022. doi:10.1177/09544062211039529.

Wikipedia contributors. Types of mesh — Wikipedia, the free encyclopedia. web link, 2022. Last accessed on 6-July-2022.

G. Wilke, J. Bailly, K. Kimura and Y. Tanabe. JAXA-ONERA-DLR Cooperation : Results from Rotor Optimization in Hover, Sep. 2021. 47th European Rotorcraft Forum, GLASGOW, United Kingdom.

Y. Zhao, H. D. Akolekar, J. Weatheritt, V. Michelassi and R. D. Sandberg. RANS turbulence model development using CFD-driven machine learning. *Journal of Computational Physics*, 411:p. 109413, Jun. 2020. doi:10.1016/j.jcp.2020.109413.