General Framework for Intelligent Unstructured Mesh Generation

by

Eric Daoust

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science (MSc) in Computational Sciences

School of Graduates Studies
Laurentian University
Sudbury, Ontario

GENERAL FRAMEWORK FOR INTELLIGENT UNSTRUCTURED MESH GENERATION

by **Eric Daoust**

a thesis submitted to the School of Graduate Studies of Laurentian University in partial fulfilment of the requirements for the degree of

Master of Science (MSc) in Computational Sciences

GENERAL FRAMEWORK FOR INTELLIGENT UNSTRUCTURED MESH GENERATION

by **Eric Daoust**

Examination Committee Members:

1. Dr. Sabine Montaut, President
2. Dr. Julien Dompierre, Supervisor
3. Dr. Nicolas Robidoux, Co-supervisor
4. Dr. Vincent François, External Examiner
5. Dr. Abdellatif Serghini, Examiner

To my parents who support me in all my projects, and for this thesis, in particular.

> Everything that can be counted does not necessarily count; everything that counts cannot necessarily be counted.
>
> Albert Einstein

Abstract

Delaunay mesh generators are fully automatic, fast and can handle complex geometries. However, users have very little control on the final mesh quality. The mesh size is only given manually on boundary edges and faces and then graded inside. Delaunay meshes are nearly uniform far from boundaries. A better mesh would take into account the distance to the walls, the curvature of the walls, the local thickness of the domain, the proximity of some discontinuities such as corners, the trailing edge, etc. These geometric characteristics of the computational domain should drive the mesh generation. The mesh should be finer where local thickness is small. It should be stretched and aligned with the walls, the tangential size being a function of the wall curvatures and the normal size being a function of the distance to the wall. What is called a good mesh also depends on the domain of application such as computational fluid dynamics, heat transfer, stress analysis, electromagnetism, etc. Good meshes for an application always depend on the geometric characteristics of the domain, but in a different manner.

The "mesh sizing problem" is the problem of automatic unstructured mesh generation according to geometric characteristics of the computational domain and depending on the domain of application. The objective of this research proposal is to reuse and extend ongoing research activities in the field on anisotropic mesh adaptation to create a general framework for the mesh sizing problem.

The now typical way to build a mesh with local size, stretching and orientation is by defining a Riemannian metric at any point of the domain and by adapting a mesh according to this Riemannian metric. The Riemannian metric defines a space

transformation that specifies the local size, stretching and orientation of the mesh to be build. To address the mesh sizing problem, one has to 1) compute geometric characteristics of the computational domain, 2) transform these geometric characteristics into a Riemannian metric, 3) store these Riemannian metric on a discrete support for use in an adaptive mesh generator.

These three steps are usually solved using a regular grid or a binary tree that overlaps the computational domain. However, such a grid is not linked to the boundaries of the domain and may not be fine enough where needed. The proposed approach uses an unstructured mesh of the domain. As the mesh is a discretization of the domain, it includes boundary points. Geometric characteristics will be computed on the mesh, starting from the boundaries and sweeping the whole mesh with neighbourhood algorithms. Problem 2) is solved by introducing parametrized templates that encapsulates the know how of the user. These parametrized templates contain rules that transform geometric characteristics into a Riemannian metric according to the domain of application. Problem 3) is solved by using anisotropic mesh adaptation. The Riemannian metric computed by steps 1) and 2) may be complex, with strong variations and directionality. A good mesh to represent a complex solution field, accurate enough but not too fine, is the mesh adapted to this field. So, the steps 1), 2) and 3) are done iteratively with a fourth step of anisotropic mesh adaptation. This process ends up with two outputs for the price of one: the accurate Riemannian metric of the geometric characteristics of the domain, and the mesh itself adapted to this Riemannian metric.

Possible applications of this research include two-dimensional shape optimization. The optimizer tests different geometries that must be automatically and intelligently meshed such that a numerical simulation can be accurately performed. An example in three-dimensional is the automatic meshing of complex geometries with thin parts as in the molding process simulation. The mesh must be fine in the thickness direction and coarse in the tangential directions. The overall benefit expected from this

research proposal is improved automation and more reliable numerical simulations by producing automatic intelligent unstructured mesh and by reducing the engineering cost associated with the meshing process.

Acknowledgements

First, I would like to express my deep gratitude towards my thesis supervisor Dr. Julien Dompierre. He is the one that defined the subject of this thesis, obtained a grant from NSERC to finance the research and supervised five summer students who completed preliminary work for this thesis. I would like to thank him for having confidence in me to undertake the primary portion of this research which finally constitutes my masters thesis. During the past two years he has been very helpful in terms of financial support, in-person meetings, programming efforts, collaborative work with the support of SVN and by arranging two summer work terms for me at École Polytechnique de Montréal. With all of this support I was able to complete my masters degree. More so than a diploma, I was able to acquire knowledge in a particular domain and I had a great experience working both collaboratively and in an industrial setting. Very few students are lucky enough to have been so well surrounded and supported in order to progress so much.

I would next like to thank the members of the Department of Mathematics and Computer Science at Laurentian University where I am registered. The knowledge I obtained during my Bachelor of Computer Science gave me the support necessary to overcome the obstacles of a masters degree. In particular, I would like to thank Dr. Nicolas Robidoux who has always supported me. He was my supervisor for my undergraduate thesis and was also my mentor for my Google Summer of Code project in 2009, both of which helped inspire me to continue doing research. I would also like to thank Dr. Kalpdrum Passi, who has been the Department chair during the last five years, for allowing me the opportunity to pursue higher education at Laurentian

University and also for the co-op program. Through four summer co-op placements, I was implicated in the work force as well as in research, which helped motivate me to progress farther than an ordinary student.

I would also like to thank the members of my thesis jury for accepting to review my thesis. This includes Dr. To Dr. Sabine Montaut, president of the jury, Dr. Nicolas Robidoux and Dr. Abdellatif Serghini from my MSc supervising committee, and especially the external examiner, Dr. Vincent François from the Département de génie mécanique de l'Université du Québec à Trois-Rivières, who agreed diligently to review my thesis. Their precious comments permitted me to greatly improve the quality of my work.

I spent the months of June 2010 and May 2011 working with the research team of Dr. François Guibault of the Deépartement de génie informatique et génie logiciel at École Polytechnique de Montréal. I would like to express my gratitude for hosting me and accepting me among his team. I would also like to thank the members of his team, notably Dr. Ying Zhang and M.Sc. Jean-François Dubé of École Polytecnique as well as Dr. Christophe Devals of Andritz in Pointe-Claire, a suburb of Montréal. This experience added a real-life context to the research that would not have been available otherwise.

I would also like to thank Dr. Paul Labbé of Institut de recherche en Électricité du Québec, in Varennes on the south shore of Montréal, for his invitation to present my work to his colleagues. Their receptivity, their industrial experience and their many helpful questions were very fruitful for this research.

Table of Contents

Abstract **v**

Acknowledgements **viii**

Table of Contents **x**

List of Figures **xv**

List of Algorithms **xx**

Preface **xxi**

Abbreviations **xxiv**

1 Introduction **1**

1.1 The big picture 1

1.2 Mesh generation problem 3

1.3 Problem statement 8

2 General framework for intelligent unstructured mesh generation **10**

2.1 Review of the literature 10

2.1.1 What adding more user control in mesh generation means? . . 10

2.1.2 Where to compute? 12

2.1.3 What would be the output 15

2.1.4 Computation of geometric characteristics 17

2.1.5 Methods to compute geometric characteristics 19
2.1.6 Execution control given to user 20
2.1.7 Direct versus iterative execution 21
2.1.8 Scope 22
2.2 SmartMesh, a new approach of the mesh sizing problem 23
2.2.1 Data structure 23
2.2.2 Use of metrics 24
2.2.3 Multi-domain capability 24
2.2.4 Perform calculations on mesh 24
2.2.5 Use of neighbour propagation 25
2.2.6 Iterative approach 25
2.2.7 Specifications on the geometric model 26
2.2.8 Separate execution parameters from mesh structure and algorithms 26
2.2.9 Full control to user 27
2.2.10 Generalized algorithms to be used 27

3 Riemannian metric **31**
3.1 Metrics 31
3.1.1 Distance 31
3.1.2 Metric, a generalization of distance 32
3.1.3 Metric tensors 33
3.1.4 Size tensors 34
3.1.5 Conversion from size tensor to metric tensor 34
3.1.6 Conversion from metric tensor to size tensor 35
3.1.7 Construction of size tensor 35
3.2 Tensor intersection 36
3.2.1 Alauzet metric tensor intersection 36
3.2.2 Robust size metric intersection 37

3.3 Functions 39
3.3.1 Distance function 40
3.3.2 Curvature function 40
3.4 Metric orientation between two boundaries 41
3.5 Object Oriented Remeshing Toolkit ($\mathsf{OO}\mathcal{RT}$) 41

4 SmartMesh library **43**
4.1 Philosophy 43
4.2 Working environment 43
4.2.1 Programming 44
4.2.2 Visualization 44
4.2.3 Documentation 44
4.2.4 Logging 45
4.3 Object-oriented structure 45
4.3.1 Mesh representation 45
4.3.2 Topological model representation 47
4.3.3 Mutual knowledge between mesh and topological model 48
4.3.4 Size tensor representation 49
4.3.5 Functions 50
4.3.6 Algorithm representation 51
4.3.7 Boundary algorithm 65

5 SmartMesh program **76**
5.1 Technical description 76
5.2 Philosophy 77
5.3 Configuration 77
5.3.1 XML 78
5.3.2 `libconfig` 78
5.3.3 Advantages of configuration file 80

5.3.4 Preprocessor 80
5.4 Execution of SmartMesh library 80
5.4.1 SmartMesh input data 80
5.4.2 `Layer` 83
5.5 SmartMesh script 84
5.5.1 Shell script usage 85
5.5.2 Command-line output 86
5.5.3 SmartMesh and $\mathsf{OO}\mathcal{RT}$ working together 88
5.5.4 Logic of SmartMesh script execution 88
5.5.5 Evolution of the input mesh 89

6 Testing and results **93**
6.1 U-duct 93
6.2 Airplane wings 95
6.2.1 AGARD test cases on a NACA0012 airfoil 96
6.2.2 AGARD01 97
6.2.3 AGARD02 100
6.2.4 AGARD03 102
6.2.5 AGARD04 105
6.2.6 AGARD05 105
6.2.7 The GAMM A7 test case on a NACA0012 airfoil 108
6.3 Hydraulic turbine 111
6.3.1 Simple distributor test cases 111
6.3.2 Industrial testing 117

7 Conclusions **130**
7.1 General conclusion 130
7.2 Analysis of the results 130
7.2.1 Testing results 130

7.2.2 Observations 132
7.3 Contributions 134
7.3.1 General framework for intelligent unstructured mesh generation 134
7.3.2 Iterative approach 134
7.3.3 Use the current mesh for all purposes 135
7.3.4 Independence of a CAD system 136
7.3.5 Efficient geometric calculations done by neighbour propagation 137
7.4 Future work 137
7.4.1 Smoothing algorithm 137
7.4.2 Function types 138
7.4.3 Minimum and maximum size 138
7.4.4 Curvature at topological vertex 139
7.4.5 Explore additional methods to compute normals and curvature 140
7.4.6 Implementation in three dimensions 140

A SmartMesh user's manual **142**
A.1 SmartMesh `libconfig` syntax 142
A.1.1 Regions 142
A.1.2 Distance functions 142
A.1.3 Curvature-to-size functions 143
A.1.4 Projection functions 143
A.1.5 Algorithms 144
A.1.6 Configuration file example 145

B Metric intersection **150**

C Javadoc documentation of SmartMesh **154**

Bibliography **155**

List of Figures

1.1 Accurate solution and corresponding mesh for the numerical simulation of airflow around a NACA0012 airfoil. 4

1.2 Inappropriate triangular mesh automatically generated by the software Tri△ngle and corresponding inaccurate solution. 6

1.3 Complex hand-made block decomposition of the computational domain. Each block is then filled with a structured grid. Pictures are from the website of Gridgen (`www.pointwise.com/gridgen`). 7

2.1 Various circles representing isotropic size of triangles. 16

2.2 Various ellipses representing anisotropic size of triangles. 17

2.3 Example of breadth-first search on a triangular mesh, which is a planar graph. 30

3.1 $\mathcal{E}_{\mathcal{M}_1}$ and $\mathcal{E}_{\mathcal{M}_2}$ in regular space. 37

3.2 $\mathcal{E}_{\mathcal{M}'_1}$ (a unit circle) and $\mathcal{E}_{\mathcal{M}'_2}$ in T space. 38

3.3 $\mathcal{E}_{\mathcal{M}'_{1\cap 2}}$ in T space. 39

3.4 $\mathcal{E}_{\mathcal{M}_{1\cap 2}}$ in regular space. 39

3.5 Curvature of a polyline at vertex P. 41

4.1 A curve at the point P is approximated by the osculating circle of radius r. The length h of the segment that has a maximal error δ with the circle. 52

4.2 Example of uniform mesh of size s. 53

4.3 Example output for distance to vertex algorithm. 54
4.4 Distance between a vertex V and an edge AB. V' is the perpendicular projection of V on the line spanned by the edge. If the projection V' is outside the edge, the closest point is the corresponding end of the edge. 56
4.5 Example of a dense mesh stretched and aligned between two nearby boundaries. 57
4.6 Orientation of a size tensor based on its proximity to two boundaries. 61
4.7 Example of a mesh produced with the distance to imaginary polyline algorithm. 63
4.8 Example of a mesh created using the distance to boundary algorithm. 66
4.9 Approximation of the curvature of a discretized curve by computing the circle that passes through three points. 69
4.10 Example of a mesh produced by the curvature of boundary algorithm. 71
4.11 Two curves having a different orientation meeting at a common topological vertex. 72

5.1 Example of configuration file using the `libconfig` format. 79
5.2 Example of mesh with and without an added structured layer of quadrilaterals. 84
5.3 Initial mesh and corresponding adapted mesh following SmartMesh program execution. 85
5.4 Initial mesh. 89
5.5 Step 1 in the SmartMesh script execution. 90
5.6 Step 2 in the SmartMesh script execution. 90
5.7 Step 3 in the SmartMesh script execution. 91
5.8 Step 4 in the SmartMesh script execution. 91
5.9 Step 5 in the SmartMesh script execution (final mesh). 92

6.1 Geometry of U-duct problem. 94
6.2 Different meshes for the same U-duct problem. 94
6.3 Additional precision where curvature is strong (SmartMesh output). 95
6.4 Test case AGARD01. Generic Delaunay mesh generated by Tri△ngle and Mach contours of the solution computed with NSC2kε. 98
6.5 Test case AGARD01. Adapted mesh generated by OO$\mathcal{RT}$ and Mach contours of the solution computed with NSC2kε. 98
6.6 Test case AGARD01. Mesh generated by SmartMesh and Mach contours of the solution computed with NSC2kε. 99
6.7 Test case AGARD01. Pressure coefficient C_p on the wall of the airfoil for the three different meshes. 100
6.8 Test case AGARD02. At the top, mesh generated by OO$\mathcal{RT}$ and Mach contours of the solution computed with NSC2kε. Same thing at the bottom, but mesh generated with SmartMesh. 101
6.9 Test case AGARD02. Pressure coefficient C_p on the wall of the airfoil with meshes produced by OO$\mathcal{RT}$ and SmartMesh. 102
6.10 Test case AGARD03. From left to right. The mesh obtained with mesh adaptation with OO$\mathcal{RT}$ and the mesh handcrafted with SmartMesh. Then the solution computed with NSC2kε on the corresponding meshes. 103
6.11 Test case AGARD03. At the top, mesh generated by OO$\mathcal{RT}$ and Mach contours of the solution computed with NSC2kε. Same thing at the bottom, but mesh generated with SmartMesh. 104
6.12 Test case AGARD04. From left to right. The mesh obtained with mesh adaptation with OO$\mathcal{RT}$ and the mesh handcrafted with SmartMesh. Then the solution computed with NSC2kε on the corresponding meshes. 106
6.13 Test case AGARD05. From left to right. The mesh obtained with mesh adaptation with OO$\mathcal{RT}$ and the mesh handcrafted with SmartMesh. Then the solution computed with NSC2kε on the corresponding meshes. 107

6.14 Test case GAMM A7. At the top, general view of the handcrafted mesh obtained with SmartMesh and the corresponding unsteady solution. Close view around the wing at the bottom. 109
6.15 Geometric domain and mesh for the first hydraulic turbine distributor. 113
6.16 Effect of curvature of border algorithm on the leading edge of the guide vane. 113
6.17 Geometric domain and mesh for the second hydraulic turbine distributor. 114
6.18 Effect of distance between two borders algorithm on the second distributor. 115
6.19 Geometric domain and mesh for the third hydraulic turbine distributor. 116
6.20 Effect of distance between two borders algorithm on third distributor. 117
6.21 Mesh generated by H2OMESH with a structured layer and isocontours of the velocity field computed by CFX. 118
6.22 Generic Delaunay mesh generated by Tri△ngle and isocontours of the velocity field computed by CFX. 120
6.23 Mesh generated by SmartMesh with curved boundaries and added skin around blades (left) and isocontours of the velocity field computed by CFX (right). 121
6.24 Mesh generated by SmartMesh with curved boundaries and no added skin around blades (left) and isocontours of the velocity field computed by CFX (right). 123
6.25 Mesh generated by SmartMesh with piecewise linear periodic boundaries and added skin around blades (left) and isocontours of the velocity field computed by CFX (right). 125
6.26 Mesh generated by SmartMesh with piecewise linear periodic boundaries and no added skin around blades (left) and isocontours of the velocity field computed by CFX (right). 126
6.27 Simple Delaunay mesh and isocontours between distributor blades. . 128

6.28 Adapted mesh with no structured skin and isocontours between distributor blades. 128
6.29 Adapted mesh with structured skin and isocontours between distributor blades. 129

List of Algorithms

4.1 **Uniform Algorithm**. Input: A topological entity E and the isotropic size value s. 53

4.2 **Distance to vertex algorithm**. Input: A topological vertex E and a distance function with its parameters, amongst them d_{max}. 55

4.3 **Distance between two boundaries algorithm**. Input: Two topological edges E_1 and E_2 and two size functions with their parameters, amongst them d_{max}. 57

4.4 **Distance to imaginary polyline algorithm**. Inputs: the polyline P, a topological face E and three size functions with their parameters, amongst them d_{max}. 62

4.5 **Distance to boundary algorithm**. Input: The topological edge E and two distance functions with their parameters, amongst them $d_{\max}$. 67

4.6 **Curvature of boundary algorithm**. Input: A topological edge E and three size functions with their parameters, amongst them $d_{\max}$. . 73

5.1 SmartMesh program. general layout of the program. 81

5.2 **BuildEdges Algorithm**. The input is the list of mesh triangles, each of them with their three pointers to vertices. 82

5.3 **SmartMesh.sh**. Input: a geometric model (with topological and geometrical parts) and an initial mesh that discretizes the geometric model. Output: an unstructured triangular intelligent mesh. 88

Preface

The tip of the iceberg is the only part that we can see. However, there is a much larger portion which cannot be seen. Likewise, this document that is in your hands or displayed on a screen is not my masters thesis, but only the visible portion of the work completed. Underneath this document there is something much greater hidden.

This thesis took place at Laurentian University in Sudbury, Ontario, Canada. The duration of the program from start to finish was approximately two years. The following four courses were completed during this program:

- CPSC 5006: Matrix Computations
- CPSC 5106: Algorithm Design and Analysis
- CPSC 5216: High-Performance Scientific Computing
- CPSC 5307: Search and Discrete Optimization

A total of ten seminars related to mathematics and computer science were attended at Laurentian University as part of a mandatory condition to complete the program. As well, the CRM-GIREF conference was attended in Québec on May 25 to 27, 2010. This was a series of presentations covering all topics in the ongoing research in mesh generation and adaptation.

In September 2010, I contributed to preparing a seminar for the Department of Mathematics and Computer Science at Laurentian University. In May 2011, I contributed to preparing and presenting a seminar at L'Institut de Recherche en Électricité du Québec (IREQ).

During the thesis there were two work placements at École Polytechnique in Montréal. The first placement took place in June 2010 and lasted approximately four weeks. The goal of this placement was to get educated by experts in the field of mesh adaptation.

The second placement took place in May 2011 with the goal of integrating SMARTMESH into an industrial optimization algorithm in order to obtain results for comparison.

Some bursaries were obtained to help support the research during the two years in this program:

- My adviser provides funds form his Natural Sciences and Engineering Research Council of Canada (NSERC) Discory Grant for salaries and travel expenses.
- The Ontario Graduate Scholarship (OGS) was obtained following application for the 2010–11 school year.
- The Graduate Teaching Assistant (GTA) bursary was obtained for both years of enrollment in the program. In exchange for the bursary, the student must complete 5–10 hours of teaching assistant tasks (marking assignments, preparing assignments, substitute teaching, etc.) per week.
- The Laurentian Summer Fellowship was obtained for the summer of 2010 for continued research during the summer months.

Some additional works were completed during this thesis:

- The document [15] summarizing a summer project by Karèle Fontaine in 2010 was proofread.
- A test case was documented where a suspected error in the "Alauzet" intersection method ([3, 4, 6]) coded in Maple. As part of the first work placement at École Polytechnique, the same test case was set up in the Riemann library, which has access to the same method for metric intersection. The test gave

the same result in both cases, which validates that the intersection method in Maple is in fact correct.

- As follow-up to the first work placement, it was requested that a new intersection method from [28] be coded for Riemann. This was done with several test cases to validate that the code is correct.

Abbreviations

The abbreviations used in this thesis are explained in greater detail in this section.

Concepts

1. CAD (Computer-Aided Design) also known as CADD (computer-aided design and drafting) is the use of computer technology for the process of design. The design consists of a geometric model of an object. The geometric model can be broken into two parts: the geometric part and the topological part. The geometric part defines the curves and their position while the topological part defines the relations between the various entities. SmartMesh uses `pirate` as CAD library.

2. CFD (computational fluid dynamics) is a branch of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. Computers are used to perform the calculations required to simulate the interaction of liquids and gases with surfaces defined by boundary conditions.[1]

Environment of development

1. SmartMesh is the name of the software that implements the ideas developed in this thesis. SmartMesh is a library (`libSmartMesh.a`), a program (`SmartMesh.exe`)

[1] From `en.wikipedia.org/wiki/CFD` retrieved on August 26, 2011.

and a bash script (`SmartMesh.sh`).

2. OO$\mathcal{RT}$ (Object-Oriented Remeshing Toolkit) is a program that modifies an input mesh based on a supplied field of metric tensors. It was developed mainly by Julien Dompierre and Paul Labbé from 1997 to 2003 at CERCA (Centre de recherche en calcul appliquée) in Montréal.

3. `XtremeMetrics` is a project developed in `Maple` that stores and manipulates (intersection and unions) various types of metric tensors. It was mainly developed by Shawn McKenzie, under the supervision of Julien Dompierre, in 2007 at Laurentian University.

4. `Layer` is a program that adds a structured skin around curves in two dimensions or surfaces in three dimensions. In two dimensions, the skin is composed of quadrilaterals that can be divided into triangles. In three dimensions, the skin is composed of prisms or hexahedra that can be divided into prisms. It was mainly developed by Ovidiu Manole from 2000 to 2002 at CERCA (Centre de recherche en calcul appliquée) in Montréal.

5. Tri$\triangle$ngle is an open source program that generates Delaunay meshes in two dimensions. This software was done by Jonathan Shewchuk, associate professor in computer science at University of California at Berkeley.

6. NSC2$k\varepsilon$ (Navier-Stokes Compressible en 2D avec modèle de turbulence $k - \varepsilon$) is a finite-volume Galerkin program computing two-dimensional and axisymmetric flows on unstructured triangular meshes. It was developed by Bijan Mohammadi at INRIA (Institut National de Recherche en Informatique et Automatique), France, from 1992 to 1994.

7. CFX is a commercial Computational Fluid Dynamics (CFD) program, used to simulate fluid flow in a variety of applications. The ANSYS CFX product allows engineers to test systems in a virtual environment. The scalable program

has been applied to the simulation of water flowing past ship hulls, gas turbine engines (including the compressors, combustion chamber, turbines and afterburners), aircraft aerodynamics, pumps, fans, HVAC systems, mixing vessels, hydrocyclones, vacuum cleaners, and more.[2]

8. `Pirate` is an in-house CAD system. It is a C++ library that reads, stores, manipulates and writes all the data used in numerical simulations. This data includes a computer-aided design with the topological model and the geometric model, two- and three-dimensional meshes, and solutions composed of scalars, vectors and tensors. It was developed mainly by François Guibault and Paul Labbé at École Polytechnique de Montréal from 1993 to until now.

Support software

1. The C preprocessor (`cpp`) is the preprocessor for the C programming language. In many C implementations, it is a separate program invoked by the compiler as the first part of translation. The preprocessor handles directives for source file inclusion (`#include`), macro definitions (`#define`), and conditional inclusion (`#if`). The language of preprocessor directives is agnostic to the grammar of C, so the C preprocessor can also be used independently to process other types of files.[3]

2. HTML, which stands for HyperText Markup Language, is the predominant markup language for web pages. HTML elements are the basic building-blocks of web pages.[4]

3. The Standard Template Library (STL) is a C++ software library which later evolved into the C++ Standard Library. It provides four components called

[2]From `en.wikipedia.org/wiki/CFX` retrieved on August 23, 2011.

[3]From `en.wikipedia.org/wiki/C_preprocessor` retrieved on August 23, 2011.

[4]From `en.wikipedia.org/wiki/Html` retrieved on August 22, 2011.

algorithms, containers, functors, and iterators.[5]

4. Apache Subversion (often abbreviated SVN, after the command name `svn`) is a software versioning and a revision control system founded and sponsored in 2000 by CollabNet Inc. Developers use Subversion to maintain current and historical versions of files such as source code, web pages, and documentation. Its goal is to be a mostly-compatible successor to the widely used Concurrent Versions System (CVS).[6]

5. Extensible Markup Language (XML) is a set of rules for encoding documents in machine-readable form. It is defined in the XML 1.0 Specification produced by the W3C, and several other related specifications, all gratis open standards.[7]

6. `Vu` is a visualization software for two- and three-dimensional meshes and solutions. It reads `pirate` files. It is developed by Benoît Ozell at École Polytechnique de Montréal.[8]

7. `ParaView` is an open source, freely available program for parallel, interactive, scientific visualization. It has a clientserver architecture to facilitate remote visualization of datasets, and generates level of detail (LOD) models to maintain interactive frame rates for large datasets. It is an application built on top of the Visualization Tool Kit (VTK) libraries.[9]

8. LaTeX is a document markup language and document preparation system for the TeX typesetting program. The term LaTeX refers only to the language in which documents are written, not to the editor used to write those documents.[10]

---

[5]From `en.wikipedia.org/wiki/Standard_Template_Library` retrieved on August 23, 2011.

[6]From `en.wikipedia.org/wiki/Apache_Subversion` retrieved on August 23, 2011.

[7]From `en.wikipedia.org/wiki/XML` retrieved on August 23, 2011.

[8]It is available on `www.invisu.ca`.

[9]From `en.wikipedia.org/wiki/Paraview` retrieved on August 23, 2011.

[10]From `en.wikipedia.org/wiki/LaTeX` retrieved on August 23, 2011.

1 Introduction

> Research is what I am doing when I don't know what I am doing.
>
> Wernher Von Braun

1.1 The big picture

Everyday, engineers design and improve objects such as cars, planes, trains, cell phones, computers, bridges, mines, bikes, heating systems, buildings, protheses and robots. Engineering is a very important human, industrial and economical activity. Most of these designs are virtual and created on a computer using a computer-aided design (CAD) software such as AutoCAD or SolidWorks. To reduce engineering costs, high quality three-dimensional prototypes are used as rarely as possible.

An improved design is faster, lighter, stronger, easier to build and to use, and more powerful while being more energy-efficient, has more features and is cheaper to produce. In short, everybody wants more for less. To achieve an improved object, the designer must be able to evaluate physical properties of the current design. Physical properties of an object include stress analysis, vibration damping, efficiency, heat transfer and dissipation, sound propagation and rigidity.

The old way to proceed was to build a prototype and then measure those physical properties using tests on the prototype. This process is lengthly and very costly. The modern way to proceed is to perform numerical simulations on the virtual prototype with a computer. This is fast and cheap. And if the design is not efficient enough

or does not meet the requirements, then the designer can easily change the virtual design and perform another numerical simulation to see if the new design is improved.

Numerical simulation consists to solve Helmoltz equations for sound propagation in fluid, Navier equations for sound propagation in solid, Maxwell equations for electromagnetic propagation, Laplace equation for heat dissipation and Navier-Stokes equations for fluid flow. These equations represent the conservation of mass, momentum, energy, fluxes, chemical species, thermal energy as well as other properties. These equations are partial differential equations with boundary conditions and initial conditions. They are usually solved with the finite element method (FEM) or the finite volume method (FVM). These methods require that the physical domain, where the equations are applied, be decomposed, or discretized, into a finite set of elements. The elements are usually triangles or quadrilaterals in two dimensions, and tetrahedra, prisms or hexahedra in three dimensions. The set of elements that discretizes a computational domain is called a mesh. Sometimes, it is called a grid when the elements used are only quadrilaterals in two dimensions and hexahedra in three dimensions.

Meshes are usually created by a mesh generator. There are essentially two kinds of mesh generators. The first kind is the unstructured mesh generator. The result is an unstructured set of triangles in two dimensions or tetrahedra in three dimensions. Principal methods are Delaunay mesh generator, advancing front and binary trees. Those methods are usually fully automatic but the user has very little control on the result.

The second kind of mesh generator is the structured mesh generator. The output is a grid of quadrilaterals in two dimensions and a grid of hexahedra in three dimensions. Before being used, the domain must first be decomposed into superblocks. Each of them will later be filled with a structured grid. The decomposition of the domain into blocks is usually not automatic. Therefore it must be done by a user, which is time consuming. However, since the user is doing the decomposition, he has more

control on the process and a skilled user can create pretty good grids, usually better than what can be done with automatic unstructured mesh generators. For more on mesh generation, consult the most comprehensive (and stable!...) web site of Robert Schneiders `www.robertschneiders.de/meshgeneration/meshgeneration.html`.

In this thesis, we will mainly use two-dimensional unstructured meshes composed of triangles. In short, *the goal of the thesis is to develop a new method to increase the user control in automatic unstructured triangular mesh generation.* Clearly, this goal needs to be explained in more detail. This is the topic of the next section.

1.2 Mesh generation problem

We wish for this research on controlled automatic unstructured triangular mesh generation to be as general as possible. This means that we should be able to apply it for numerical simulations in heat transfer, stress analysis, wave propagation, electromagnetism as well as other applications. Obviously, it is not possible in the scope of this thesis to try all application domains. But if this research is generic enough, we should be able to apply it for one application domain in particular. We choose the computational fluid dynamics (CFD). This application domain requires very specific meshes to achieve accurate numerical solutions. Historically, CFD has always pulled research in mesh generation.

Figure 1.1 displays the result of a numerical simulation of airflow around the generic airfoil called the NACA0012. The airflow conditions are a free stream at an angle of attack of zero degree, a speed of 0.85 times the speed of sound (a Mach number of 0.85) and a Reynolds number (the fluid viscosity) of 5 000. This is considered an easy problem because, at this Reynolds number, this is a lot more viscous than real air. The numerical simulation was done with the CFD solver NSC2$k\varepsilon$. The first plot displays the isocontours of the Mach number (the speed) at every 0.025. For more details about this test case, see Section 6.2.7.

This solution presents many interesting features. The most noticeable is the wake

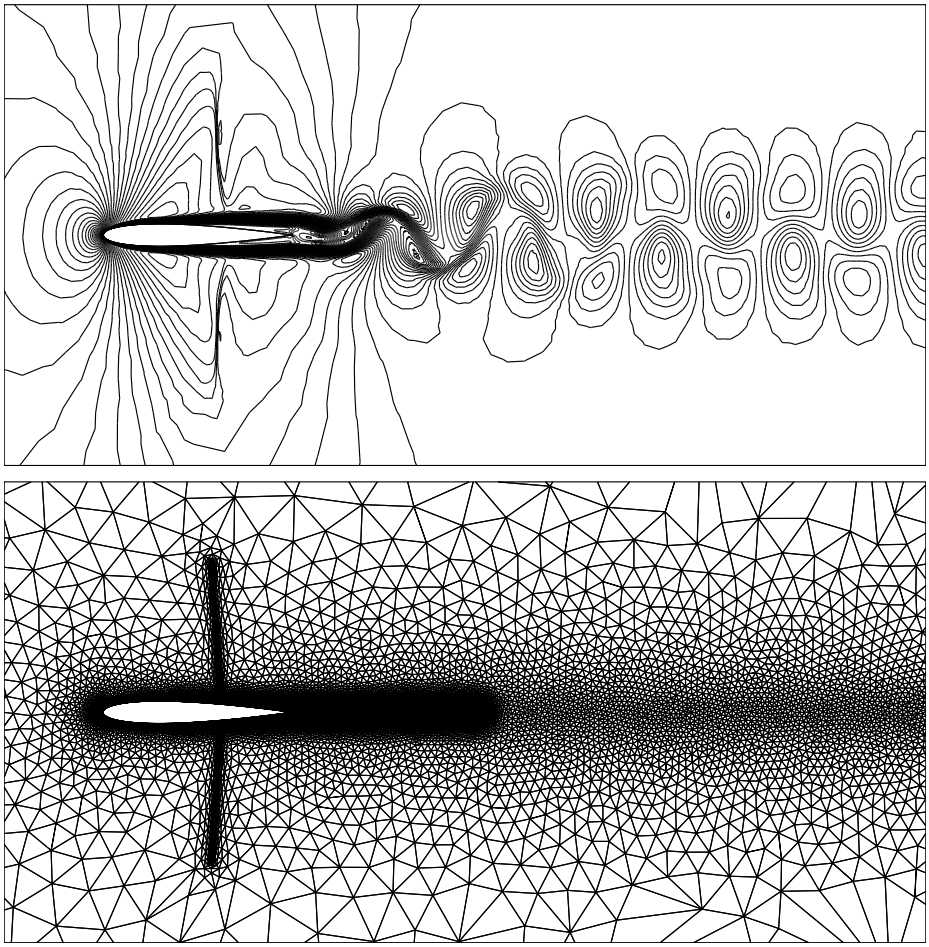


Figure 1.1: Accurate solution and corresponding mesh for the numerical simulation of airflow around a NACA0012 airfoil.

behind the airfoil, composed of vortices travelling to the right in a von Karman alley. But there is also a boundary layer around the airfoil, an upper and lower transonic shock, a flow separation on the airfoil at about two thirds and recirculation around the trailing edge.

To compute such an accurate solution with all these features, a corresponding accurate mesh is needed. Figure 1.1 also shows the unstructured triangular mesh used. This mesh was produced by the method developed in this thesis. This mesh has 17 740 vertices and 35 144 triangles. But the interesting fact is where those vertices and triangles are located. Clearly the mesh is more dense where the solution has

some strong features and is more coarse where the solution is rather uniform. Also, triangles are stretched and aligned with anisotropic features such as boundary layer and shocks.

Why would we do a masters thesis on a new method for unstructured triangular mesh generation? There are many of them available, including commercial, free and open-source generators. For example, this thesis will use Tri△ngle for comparison. This software was written by Jonathan Shewchuk, associate professor in computer science at University of California at Berkeley. This software constructed 15 years ago is free, very fast, robust and is used by so many people that it is now bug free. It is easy to use, generic and fully automatic. The underlying method used to generate an unstructured triangular mesh is the Delaunay method. Jonathan Shewchuk even won the fourth J. H. Wilkinson prize for numerical software in 2003 for his software Tri△ngle. This prize is awarded every four years since 1991 and has a bursary of US $3000. In short, there is nothing better than Tri△ngle.

Figure 1.2 shows an unstructured triangular mesh generated by Tri△ngle over the same NACA0012 airfoil. Parameters were adjusted such that this mesh contains about the same number of vertices and triangles as for the previous mesh of Fig. 1.1. In this case, there are 17 910 vertices and 35 108 triangles. However, the software Tri△ngle has few parameters to control the mesh generated. The user manually sets the number of vertices on the boundaries and some diffusion parameters that increase triangle size as they get farther from the boundaries. But there is no way to concentrate the triangles in a region like the wake or the shocks and there is no way to stretch and align triangles along the airfoil.

Figure 1.2 also shows the solution, with the same airflow conditions, computed on the mesh generated with Tri△ngle. If you compare with the solution of Fig. 1.1, you can roughly see the same solution with the boundary layer and some kind of wake. This solution is not wrong, but it is clearly not accurate. At this moment, the typical sentence used by an engineer is: “Garbage in, garbage out.” Unstructured

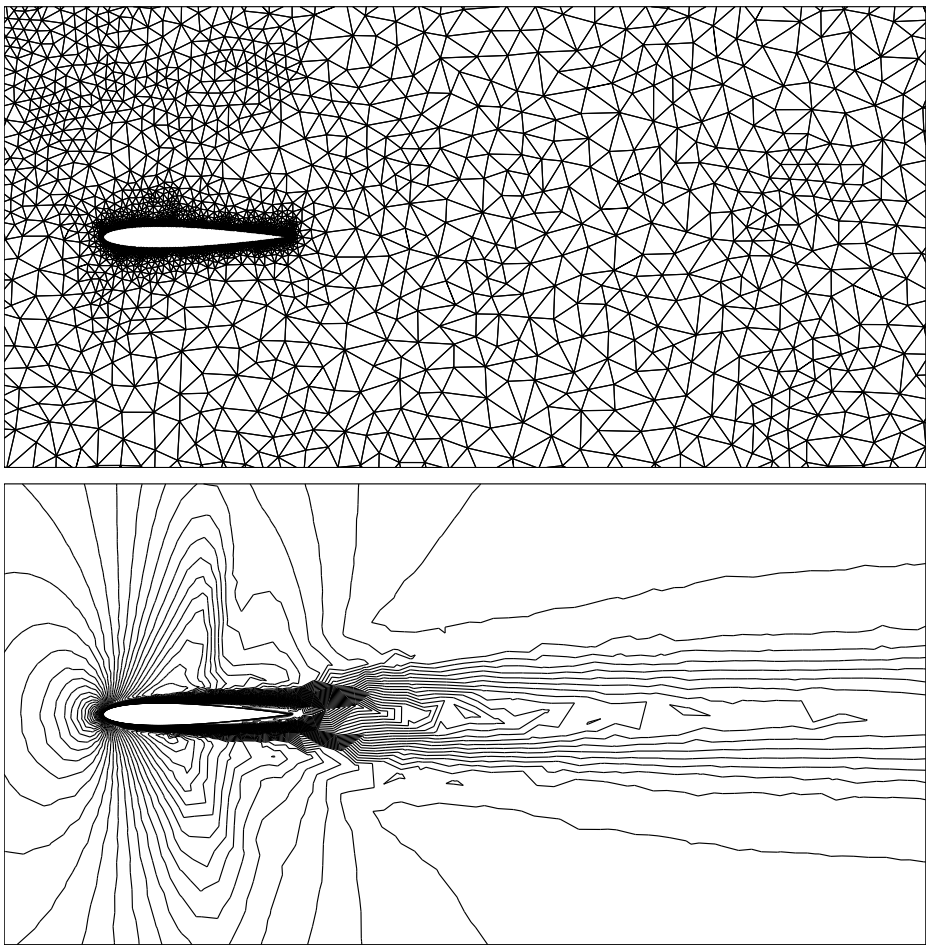


Figure 1.2: Inappropriate triangular mesh automatically generated by the software Tri△ngle and corresponding inaccurate solution.

triangular mesh generation is fully automatic and completely generic, but the lack of control on the mesh generated is such that meshes cannot be used to compute accurate solutions. In the goal of designing a better product and to gain a few more percents of efficiency, the engineer cannot use an inaccurate tool.

More and more efficient planes, cars, turbines and cell phones exists. This means that engineers apparently found some techniques to get accurate numerical solutions on appropriate meshes. The most common way to proceed is to do a hand-made block decomposition of the domain. See Fig. 1.3 for examples. The user, with a graphical user interface, decomposes the domain into blocks. As he is doing the job by hand, he

takes the opportunity to put blocks where there should be a wake and shocks. Also, because he knows that there will be a boundary layer along the walls, he creates thin blocks aligned with the walls. Then each block is filled with a structured grid. The engineer has two controls when creating structured grids. He can control the number of vertices used in the three directions of the grid. He can also control the spacing of the vertices in the three directions. All of those controls are sufficient to create long and thin hexahedra aligned with the walls, which is very important for CFD.

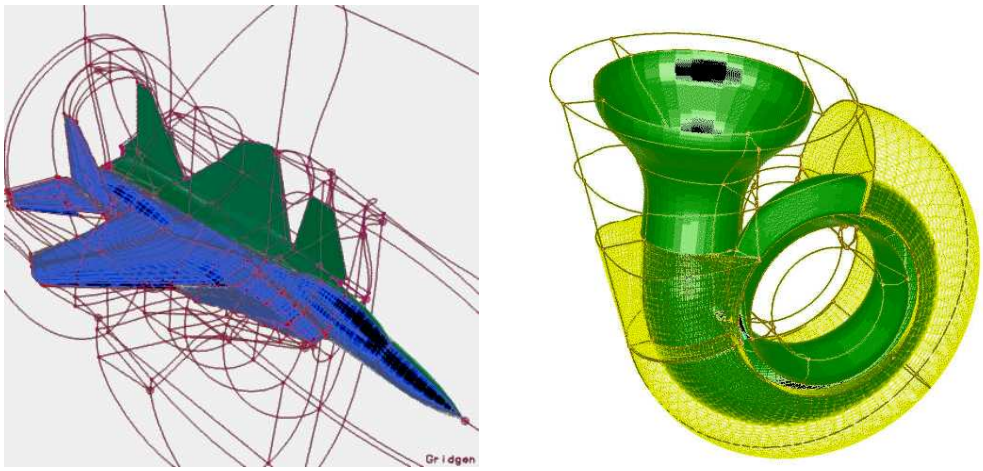


Figure 1.3: Complex hand-made block decomposition of the computational domain. Each block is then filled with a structured grid. Pictures are from the website of Gridgen (`www.pointwise.com/gridgen`).

This process is not automatic. Depending on the problem, with the tools available in the graphic user interface and with the skill of the user, it usually requires several hours to several days to finish. This is not convenient if block decomposition must be adjusted while optimizing the design. The probability of creating an accurate solution on the first attempt is small and this is a problem given the amount of time required to create a single solution.

Many automatic unstructured mesh generators are based on the Delaunay method which is about 80 years old [13]. This problem is so "easy" that the "obvious" solution was found even before the problem existed. The first efficient computer

implementation came with the computer age [24]. On the opposite, the fact that there is still no automatic block decomposition until now, with all the research done, is clearly an indication that this is, at least, a difficult problem, and at worst, a problem with no solution.

A common partial solution is to have a parametrized block decomposition for a fixed topology but a slightly variable geometry. This is commonly used in shape optimization. The topology of the object will not change, meaning that the relation between topological entities remains the same, but only its geometry will change. A block topology could be set forever for all the shapes. This requires a program dealing with CAD and geometric design. The program fits, or matches, the blocks to the curves and surfaces of the domain. Writing such a program usually takes a few months to a few years for a given problem. See [21] as an example.

1.3 Problem statement

In short, nothing is perfect. Everything has some advantages and some disadvantages.

The advantages of the unstructured triangular or tetrahedral mesh generation is that it is automatic, general and computationally cheap. One software can be applied to any possible problem on earth. The disadvantage is the lack of control on the meshes generated. For many applications, the meshes generated is simply unusable.

The advantage of structured grid generation with block decomposition is that the user has some control on the mesh generated, at least enough control that the structured mesh generated can be used for numerical simulations. The disadvantage is that it is not automatic and is also not general. This process is time consuming and the result for one problem cannot be reused for any other problem.

As for university research, we want to contribute to science by improving something, we have two possible choices:

A) Add more automaticity and generality in structured grid generation with block decomposition.

B) Add more user control in automatic unstructured mesh generation.

Given the previous discussion about the fact that problem A) is ill-posed, or even has no solution, then this is probably too difficult for the scope of a MSc thesis. Therefore, B) is clearly the better choice.

The goal of this thesis is to develop, implement and test new methods to add more user control in unstructured triangular mesh generation.

Note that the problem statement of this thesis explicitly uses the word "triangular". It means that this MSc thesis will be in two dimensions. Tackling three-dimensional mesh generation with tetrahedra is the scope of a PhD thesis. However, when there will be choices, if a given method does not have an obvious extension in three dimensions, then this method should not be considered.

2 General framework for intelligent unstructured mesh generation

> A new or novel approach usually means you have not read other people's papers.
>
> Wagdi G. Habashi

2.1 Review of the literature

2.1.1 What adding more user control in mesh generation means?

The goal of this thesis is to add more user control in unstructured triangular mesh generation. But what does this mean?

We want the user to be able to set the size, stretching and orientation of all the triangles of the mesh. There are usually a large amount of triangles in a mesh, so this is a huge task for the user. Therefore, we want the user to be able to set the size, stretching and orientation of all the triangles of the mesh, but with a minimum set of commands and parameters.

For example, with one instruction having few parameters, we want to be able to create a wake with small and stretched triangles along the wake, and with triangle size increasing as they get away from the wake. Another example, with one instruction having few parameters, we want to be able to create a boundary layer with small and stretched triangles aligned with a curved boundary with triangle size increasing as they get away from the boundary. We also want triangle size be inversely proportional

to the curvature of the boundary. Finally, although more difficult, if two boundaries are close together, we want a finer mesh between them.

In the literature this problem is referred to as the *mesh sizing problem*. There are many available publications with some partial solutions, for some given applications and with some constrains. We must know them if we don't want to reinvent the wheel. We must analyze and understand them to figure out what is good and what is bad to be able to design a new and (hopefully) better approach.

The mesh sizing problem can be divided in many sub-problems. Clearly, the final mesh depends on the problem at hand and the shape of the boundaries. The software to be created must take into account curves that define the computational domain. So there must be some link with a CAD system. Which one? Catia, Creo Parametric (previously known as Pro/Engineer), NX (previously known as Unigraphics), Autocad, Capri, or should the user create his own? And how close will the link between the software and the CAD be? We need to compute geometric characteristics such as the distance to a curve and its curvature. This raises many questions. How will these geometric features be computed? Where, and for which vertices, are they going to be computed; we don't want to compute these features for all vertices in $\mathbb{R}^2$. Which data structure will be used to store these results? How will these geometric features be transformed into mesh sizing information? Which sizing information, a singular size value only or a combination of size, stretch and orientation will be chosen? At which level will the user have control and which controls should be made available? How generic will the solution be? Is it good only for CFD, or can it be applied for heat transfer, stress analysis, electromagnetism, etc?

In short, there are many sub-problems to the *mesh sizing problem*, and even more possible solutions. The following sections contain a review of the literature where each contribution contains some solutions to some of the previous sub-problems.

2.1.2 Where to compute?

We want to compute geometric characteristics of the domain, the distance to a curve being an example, at a given point of the geometric domain. This information will then be transformed into a size. The question is which points should be used? In an ideal world, all points inside the geometric domain could be used. But this set is infinite. Realistically, a finite subset must be used. Which one? Many answers are found in the literature.

2.1.2.1 Vertices selected on the fly

In [11, 16], the authors do three tasks at the same time. They evaluate geometric characteristics, transform this information into a size value and generate a vertex to be added in the mesh generated with an advancing front method. This means that geometric characteristics are evaluated at the point where the next vertex of the mesh will be generated.

There is no need for an external data structure with vertex locations that overlaps the domain. The advantage to this approach is that there is no additional storage required in the computer's memory because the geometric characteristics and the size values derived are applied immediately, and the information is then discarded.

While this approach saves in computer storage, a disadvantage may be the loss in terms of speed of execution. The location of each vertex of the generated mesh is the result of trial and error, so geometric characteristics may be computed for many locations for each vertex.

2.1.2.2 Overlapping grid

A grid is a structured mesh of $N_i \times N_j$ vertices composed of $(N_i - 1) \times (N_j - 1)$ quadrilaterals in two dimensions, or a mesh of $N_i \times N_j \times N_k$ vertices composed of $(N_i - 1) \times (N_j - 1) \times (N_k - 1)$ hexahedra in three dimensions. In mesh sizing,

we usually use regularly spaced grids. It is easy to find a big rectangle $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ that bounds the geometric domain in two dimensions, or a big hexahedron $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}]$ that bounds the geometric domain in three dimensions. Then a regular grid is used to discretize the rectangle or hexahedron. Some of the grid's vertices will be inside the geometric domain, some will be outside. There may be some vertices that are exactly on the boundaries of the geometric domain. This is not an obvious task to figure out if a vertex is inside or outside the geometric domain. For each grid vertex, we can compute and store geometric characteristics of the geometric domain. For any given point, it is easy to compute in which quadrilateral or hexahedron it belongs, in order to retrieve stored values at grid vertices around and interpolate stored values.

This approach has many advantages. It is very simple and straightforward to implement. However, it has many disadvantages. No, or few, vertices of the grid will be on the the boundaries. This means that the vertices will have a positive distance to the boundaries. If a point is set on the boundaries and its distance to the boundaries is interpolated from stored values, its distances to the boundaries will not be zero. In short, a grid is not "body-fitted". Also, if the geometric domain is thin, almost all the grid's vertices will be outside the geometric domain and few, even none, will be inside. One can increase the accuracy and the number of vertices inside the geometric domain by using finer and finer grids. But this is costly in computer time and memory.

2.1.2.3 Overlapping quadtree and octree

Quadtrees are data structures similar to grids. The quadtree overlaps the geometric domain. However, each quadrilateral in the quadtree, may or may not be decomposed into four sub-quadrilaterals, totally independent of the neighbour quadrilaterals. The subdivision could be recursive. Octrees are three-dimensional grids where each hexahedron could be subdivided recursively into eight sub-hexahedra. The advantage to

this approach is that the user can choose how many levels of sub-grids are contained at each location on the domain. Using this intuition the user can add more precision in the important areas. Implementations of the grid approach exist in [12, 35].

However, there are drawbacks to this approach. The binary trees are still not body-fitted. It is still a separate entity from the geometric domain, and as a result there is a loss of precision at all points in the grid. If the binary tree is infinitely subdivided, the error converges towards zero. Also, this data structure is not so easy to program, but is a classic computer science exercise of style.

2.1.2.4 Overlapping loose mesh

Some implementations overlap the geometric domain by a "loose" unstructured mesh [19, 32]. The mesh could be loose in many ways. It may not have vertices inside the domain, but only on the boundaries. It may have vertices on the boundaries, but the edges between boundary vertices may not be on boundaries (we say that it is not a constrained Delaunay mesh). It may have edges and triangles outside of the geometric domain (we say that the mesh overlaps the convex hull of the geometric domain). In all cases, the unstructured mesh at least overlaps the geometric domain and some vertices are on the boundaries.

The advantages are that a loose mesh is easy to build and is loosely body-fitted. The disadvantage is that it is still an extra data structure. These days, it is so easy to have a Delaunay mesh of any geometric domain that this approach is somehow unjustified.

2.1.2.5 Conformal mesh

Contrary to the loose mesh, the conformal mesh is a discretization of the geometric domain. There is no vertex, edge or element outside the geometric domain. The boundary vertices of the mesh are actually on the boundaries of the geometric domain. The boundary edges of the mesh are on the boundaries of the geometric domain. We

say that the mesh is body-fitted. The distance of boundary vertices to the boundaries is exactly zero by construction.

Nowadays, such a mesh can be generated quickly by open source software in two and three dimensions. Any CAD system can generate an unstructured mesh of the geometric domain in very few clicks. One disadvantage is that an unstructured set of triangles or tetrahedra is more difficult to work with than a regular grid. But this is the usual task of any unstructured finite element solver and apparently, programmers can survive using connectivity tables. This approach is used in [14].

2.1.3 What would be the output

From the previous section *Where to compute?*, a set of vertices to compute geometric characteristics was selected. Geometric characteristics at these location will be computed and then transformed into some size for the mesh to be built. For the size, there are two main schools of thought, isotropic and anisotropic.

2.1.3.1 Isotropic size value

The isotropic size value h ([10, 11, 12, 16]) is one such way to represent the desired mesh size. The triangles in that area on the final mesh will respect this size value h as closely as possible. However, this value does not specify requirements as stretching and orientation of those triangles and the triangle's edges will be close to equal at all times.

Figure 2.1 from [23] shows a graphical representation of a field of size. On a regular grid that overlaps the geometric domain (which is simply a rectangle in this case), the field of sizes is represented by circles whose diameter corresponds to some specified size. On the right, you can see the corresponding mesh where triangles are about the specified sizes. Note that only the size is specified and since the circles are all perfectly round, and there is no stretching or orientation. The triangles in the

corresponding mesh are all close to being equilateral. In this case, we say that the mesh is isotropic.

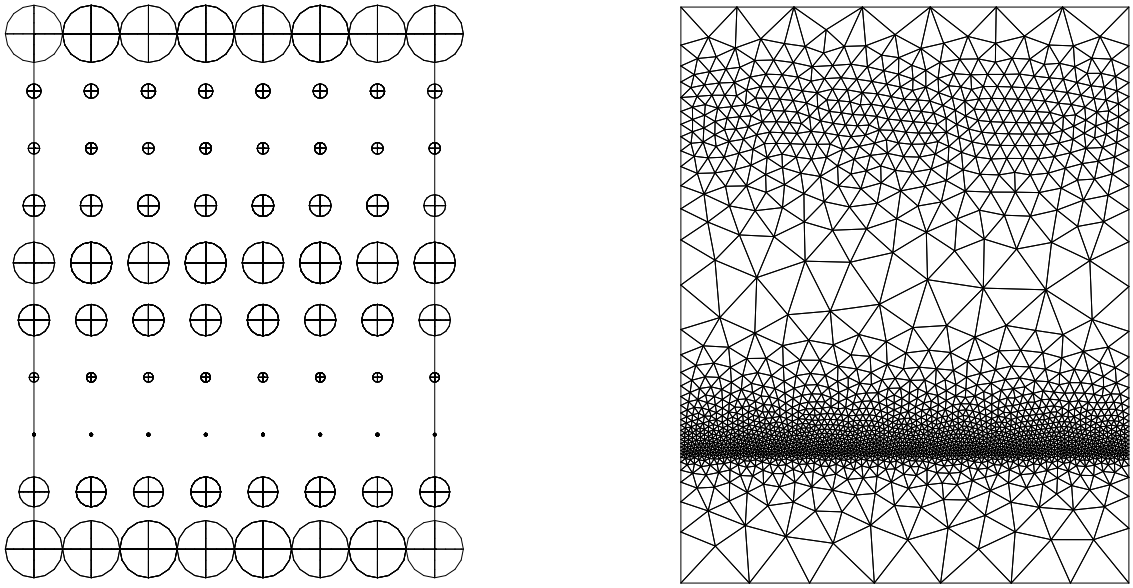


Figure 2.1: Various circles representing isotropic size of triangles.

2.1.3.2 Anisotropic size value

Any engineer with experience in numerical simulations, especially computational fluid dynamics, knows that many phenomenons are localized and directional. Boundary layers, wakes, and shocks are examples in fluid flow. Any engineer that did some numerical simulations knows that stretched and aligned triangles with directional phenomenons give better results. Therefore, they all want the capability to build such meshes. Those meshes are called anisotropic.

An anisotropic size value is represented by a metric tensor, which is a symmetric positive definite matrix. This is used in [2, 5, 14, 19, 20, 25, 26, 27, 35] amongst many other publications. In two dimensions, a 2×2 symmetric positive definite matrix can be graphically represented by an ellipse. An ellipse has two axes, corresponding to the two eigenvectors of the matrix, so the orientation of the mesh triangles can be controlled. The axes have two lengths, corresponding to the two eigenvalues of the

matrix, so the size of the mesh triangles can be controlled. Metric tensors give the user a lot more control on the mesh he wishes to have.

Also, metric tensors generalize isotropic sizes. It is sufficient to set as equal the two sizes of a metric tensor to have an isotropic metric tensor.

Figure 2.2 from [23] shows a graphical representation of a field of metric tensors. On a regular grid which overlaps the geometric domain (which is simply a rectangle in this case), the metric tensors are represented by ellipses. On the right, you can see the corresponding mesh where triangles are about the specified size, stretching and orientation.

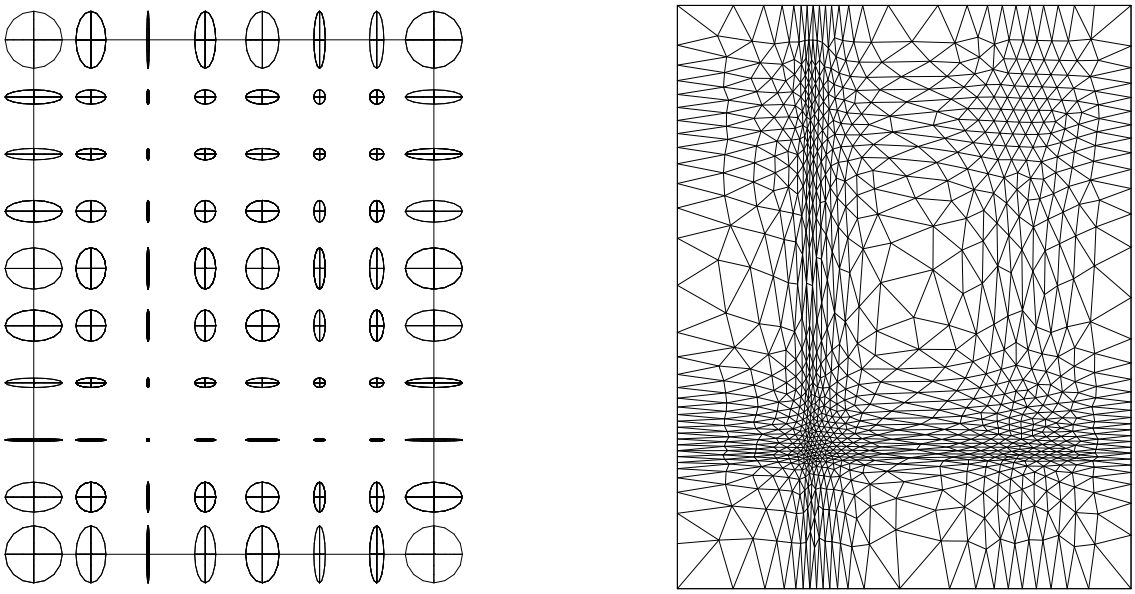


Figure 2.2: Various ellipses representing anisotropic size of triangles.

2.1.4 Computation of geometric characteristics

In the mesh sizing problem, we want the mesh to be sensitive to geometric characteristics of the geometric domain. Depending of the problem at hand, many geometric characteristics could be evaluated.

2.1.4.1 Distance to a boundary

Distance to a boundary is the most common geometric characteristic in literature. We want the mesh to be finer close to the boundary and more and more coarser as we get farther from the boundary. To set the size of the mesh at a given point, we need to know the distance of that point to a boundary. Some solutions such as [10, 14] require a size value to be set according to the distance of mesh points from a specific boundary.

2.1.4.2 Curvature of a boundary

A triangular or quadrilateral mesh in two dimensions, or a tetrahedral or hexahedral mesh in three dimensions, are all composed of linear elements. Geometric models are all composed of curves. The discretization of curves by linear elements introduces discretization error, especially where the curvature of the curve is high. One alternative is to have smaller elements where the curvature is strong. By having the element size set accordingly to the curvature, it is possible to reduce the error between the curve and the edges of the mesh. In short, we need to compute boundary curvature to set the mesh size accordingly. Solutions that calculate the curvature of boundaries include [12, 26, 35].

2.1.4.3 Distance to a vertex

In many numerical simulations, sharp corners on boundaries of the geometric model will introduce some discontinuities in the solution. To have an accurate solution in the vicinity of a sharp corner, a finer mesh is needed. Sharp corners in a geometric model are usually represented by a vertex. In such cases, we need to compute the distance to a vertex and set the mesh size accordingly. Such examples exist in [14].

2.1.4.4 Thickness between two close boundaries

Sometimes, geometric characteristics such as the curvature of boundaries and distance to boundaries are not sufficient to take into account the fact that two boundaries are close proximity of each other. A typical example of narrow passage in CFD are nozzles. Two smooth curves that are close together causes the fluid flow to accelerate, producing interesting flow features that require a finer mesh to be simulated accurately. Some algorithm exists to detect boundary proximities in [12, 14, 35].

2.1.5 Methods to compute geometric characteristics

From the previous section, there are many mesh sizing approaches in the literature that require the evaluation of some, or all geometric characteristics introduced previously. Each paper uses different methods to evaluate these geometric characteristics.

2.1.5.1 Use of proprietary CAD library

Some implementations such as [11, 16] depend on the CAD library in order to perform calculations. Questions such as the distance to a curve or the curvature of a curve are forwarded to the appropriate functions of the CAD library. Such approaches are precise, assuming that the CAD library is well written. However, interfacing with the API of the library can be a difficult task. Also, CAD libraries are commercialized, so they can be expensive and difficult to obtain. And once a software is link to one particular CAD system, it is not done for the others. A workaround is to work with CADNexus/CAPRI CAE Gateway, which is a buffer layer between an application and many CAD libraries.

2.1.5.2 Recreate in-house CAD system

As a workaround for the commercial CAD problem, some have created their own in-house CAD library based on a discrete representation of the geometric domain.

In [7], the design done in AutoCAD is exported as an STL (stereolithography) file. This file contains the boundaries of the geometric design as polygons in two dimensions, or as a mesh of three-dimensional triangles in three dimensions. Then the mesh generation software, which was developed, used this information instead of the initial design in AutoCAD, which makes this approach independent of AutoCAD.

The approach is similar for the software MEF++ (Méthode des éléments finis en C++) developed at GIREF (Groupe Interdisciplinaire de Recherche en Éléments finis) of Laval University in Quebec City. Regardless of the CAD system used, a mesh of the geometric domain is produced in the CAD system, and is exported to a file. Then MEF++ imports this mesh, extracts the boundary triangles and recreates an in-house representation of the geometric domain from the discrete triangulated boundaries.

2.1.5.3 Grid

Another approach is to use the overlapping grid, or binary tree, as an approximation of the boundaries. By storing appropriate information on the grid, questions like the distance to the wall or the curvature of the wall could be answered approximately by some algorithm performed on the grid. Implementations of this approach exist in [12, 35]. Such approaches are simple. However the grid is not body-fitted to the geometric domain, therefore there is a loss of precision for all points.

2.1.6 Execution control given to user

In all approaches, there has to be a user that initiates an execution in order to get a result. What differs is how much control the user has in driving the program to generate the desired result.

2.1.6.1 Hard-coded execution

Most implementations are inspired by a single application domain, and maybe also by a single geometric domain, and therefore it is sufficient to have a single execution procedure that does not vary. While this approach is sufficient to resolve the given problem, it cannot be used to solve other problems in other domains. Another disadvantage is that the user cannot change any properties of the algorithms without modifying and re-compiling the code, which is not always possible if the user is not the original programmer of the software.

2.1.6.2 Execution parameters

Sometimes a list of parameters can be made available to the user where he can assign values to specific parameters that are provided by the program. Examples of such parameters include the number of iterations or the error tolerance value to determine how closely the mesh will try to duplicate a curve. Approaches that allow users to input some parameters for rather specific problems are [21, 26] for example.

Unfortunately, this approach only makes certain parameters available and sometimes the user still does not have enough control on the execution. It is also possible that a user desires having a parameter vary for different regions in the mesh.

2.1.7 Direct versus iterative execution

Is computing geometric characteristics, transforming them in metric tensors and creating the corresponding mesh to be done in one shot (a direct approach) or should it be done in a process with feedback (iterative approach)?

2.1.7.1 Direct

Many approaches do all the steps in one shot. For example, in [11, 12, 16, 35] once the mesh is created, there is no further work that can be done. Re-executing with

the same parameters will not change the result. The only way to further improve the mesh is by changing the input parameters.

2.1.7.2 Iterative

Other implementations can re-execute the same algorithms on an already adapted mesh for further improvements. Such examples include [14, 18]. Since the mesh is used for calculations, if one re-calculates on an adapted mesh, which is different and improved from the previous mesh, the environment is different and thus it is possible to obtain better geometric characteristics, better size and finally a better mesh.

It is believed that mesh generation, adaptation or optimization is in fact a NP-hard problem. Using nomenclature of metaheuristics, a one-shot approach corresponds to a greedy algorithm. Greedy algorithms will not give the optimal result, but could, at least, give a satisfying result. However, an iterative approach, driven by some cost function to optimize, allows modifications from a previous state and should produce better results.

2.1.8 Scope

The mesh sizing problem has been applied to many different application domains. Each domain has a different set of requirements and a different purpose. Developing a software that can satisfy the needs of all application domains is a big challenge.

Computational fluid dynamics (CFD) involves computing fluid flow through a space where there are obstacles. This involves boundary layers with turbulence models, wakes and shocks. In this case, the mesh sizing problem requires the capability of creating anisotropic and localized meshes.

The molding of plastic objects is also an area where mesh sizing is critical. In this application, we simulate the pouring of liquid in the mold and then calculate the cooling period required for the object to harden. The geometric domain is a mold and they are usually characterised by thin walls. This simulation requires enough

elements across the width of the walls in order to calculate the propagation of the liquid throughout the walls. If the mesh is isotropic and if there are enough elements across the width, then it implies a very fine overall mesh. As the mold is a thin wall with flat boundaries close to each other, the ability to detect close boundaries and to have anisotropic elements between boundaries is critical.

Electromagnetism is an application that requires sizing the mesh appropriately in order to accurately run a simulation. This domain applies to calculating the magnetic field created by electric motors, studying the effect of cell phone waves on the human body and studying the effects of lightning striking a lightning rod. In all of these cases, a fine mesh in the area where the phenomenon takes place is required.

In short, all application domains may have their own requirements on meshes. However, a good mesh sizing tool should be flexible enough to allow the experimented user to create the mesh he needs.

2.2 SmartMesh, a new approach of the mesh sizing problem

Given what we have learned in studying previous works in mesh sizing, this section will propose a new approach that will attempt to take advantage of what past approaches do well, while trying to eliminate the drawbacks. *The new approach described in this thesis will be called* SMARTMESH.

2.2.1 Data structure

The data calculated during the execution will be stored in the mesh itself. This means that no additional data structure needs to be created because the vertex is already part of the implementation. Instead, it is only necessary to extend the storage within the already existing vertex.

Unlike all overlapping approaches, the mesh is body-fitted to the domain and therefore the vertices, especially along the domain boundaries, are located exactly

on the boundaries. This approach will also allow the use of neighbour propagation algorithms to calculate geometric properties.

2.2.2 Use of metrics

Each mesh vertex will have a stored 2×2 symmetric positive definite matrix. This matrix can be decomposed into two eigenvalues and two eigenvectors which can then be used to represent an ellipse. The eigenvectors represent the orientation of the ellipse while the eigenvalues represent the corresponding size along each axis of this ellipse. This gives a lot of flexibility to the user in being able to better control the mesh size, stretching and orientation.

2.2.3 Multi-domain capability

In SmartMesh all general algorithms will be made available. This will allow the software to be applicable for all application domains. It is the user's responsibility to select the correct algorithms with the correct parameters in order to achieve the desired result.

Also, additional parameters must be made available so that the user can control the rate of change in mesh size based on a given entity's proximity relative to the region of interest.

2.2.4 Perform calculations on mesh

In SmartMesh, the computations of geometric characteristics are performed on the mesh itself. The mesh must store links towards the topology of the geometric model, but does not need to take into account the geometric part of the geometric model. Therefore, SmartMesh does not have to be bounded to a single computer-aided design software since the calculations can be done without knowledge of the curves defined in the geometric model.

Furthermore, the boundaries, which are curves in the geometric models, will be polygons in the mesh. This makes the implementation of algorithms which calculate geometric characteristics, such as the distance to the boundary, very easy and fail-free.

While the approach of using the mesh for calculations leads to a loss of precision due to the fact that the curves in the geometric model are not used, the user can control this loss of precision by specifying a more precise mesh where needed.

2.2.5 Use of neighbour propagation

In this approach, neighbour propagation techniques will be used. Since we are using the mesh itself to perform calculations, the graph structure of the mesh allows for easy traversal from one vertex to a neighbour vertex. Therefore from a start point, it is easy to complete an algorithm without visiting all the mesh vertices.

Also, the storage of additional information at each vertex can further reduce the amount of work required if neighbours can exchange information amongst themselves.

The goal for all algorithms in SmartMesh is that each vertex can compute its geometric characteristics in $O(1)$ time. The transformation of geometric characteristics into a metric tensor must also be in $O(1)$ time.

2.2.6 Iterative approach

The creation of an initial Delaunay mesh is fast and easy. However such a mesh is not necessarily suitable for CFD simulations. But with an initial bad mesh it is possible to compute geometric characteristics and store the new metric tensors in the mesh vertices in order to achieve a better mesh.

Unfortunately it is impossible for a mesh to be adapted to a user's needs in a single iteration. Typically the initial mesh is very bad for CFD simulations. It may be also very bad for geometric characteristic computations. Computations can only be made at mesh vertices and if these vertices are not close enough, it is impossible to get the correct picture in the refined mesh. In essence, not having enough vertices

(large triangles) means there is not enough of a sample to immediately generate a new solution that satisfies the user.

However, even though the new mesh does not entirely satisfy the user's needs, it will have a much higher amount of vertices in the areas of interest. From here, if we re-apply the user's desired algorithms at these new points, it is now possible to get an even better picture of what the final mesh needs to look like.

Using this iterative approach, it is possible to eventually converge towards a mesh that satisfies the user's request. By iterating, we are able to obtain this result regardless of the quality of the initial mesh.

2.2.7 Specifications on the geometric model

The mesh is a discretization of a geometric model. While the mesh will be modified during the iterations of mesh adaptation, the geometric model will remain unchanged. Each alternative mesh must continue to conform to the boundaries of the geometric model.

Since the geometric model remains constant, this becomes the backbone from which the user will specify his algorithms to execute and their respective parameters. From there, several meshes will be created and eventually converge towards those specifications.

2.2.8 Separate execution parameters from mesh structure and algorithms

It is important that the execution be separate from the rest of SmartMesh components. With this separation the user can then select via execution configuration which algorithms to execute, their order of execution and the parameters to be used.

The parameters specified by the user target topological regions on the geometric domain which does not change during the iterative execution.

2.2.9 Full control to user

With this capability, the user can fully control the execution in order to achieve the result that satisfies the requirements of his particular problem. SmartMesh is flexible to any execution configuration; the user's expertise drives SmartMesh to perform the correct operations.

The user can specify which algorithms to use, the order of their execution and the parameters to use for each algorithm executed. A computer is fast, but not intelligent. The intelligence comes from the user and his capability to script his knowledge into a file of parameters.

2.2.10 Generalized algorithms to be used

In order to create a general framework for intelligent mesh generation, it is important to establish algorithms that go beyond the properties of a single application domain. One must define general algorithms that transform geometric properties into a physical size that can be used to modify the mesh appropriately.

The algorithms in this section form the foundation that will be used in the implementation of all algorithms in SmartMesh.

2.2.10.1 Uniform algorithm

It would be useful to define an algorithm that can assign constant sized isotropic triangles in a particular region of the mesh. Even though SmartMesh is only useful for localized problems requiring anisotropic triangles in the mesh, there are still requirements for such triangles, particularly in the areas of the mesh not affected by any other mesh changes.

This algorithm can also be used to assign a base size for the mesh in order to avoid situations where the mesh suffers from an extreme change in mesh size between algorithm-affected areas and the rest of the mesh.

2.2.10.2 Distance to a vertex

In many problems there are vertices around which the mesh should be more fine. In order to satisfy this problem we must define an algorithm where the size of mesh triangles located near a particular vertex of interest are modified in relation to the distance from this vertex.

2.2.10.3 Distance to a segment

This algorithm defines the distance of a vertex to a segment that is bounded by two points. The distance in question projects the vertex to either one point, the other point, or to a location on the segment. If the projection is on one of the two extremities of the segment, the distance is easy to calculate because the exact coordinates of these two extremities are known. However, if the vertex in question projects to a location on the segment situated somewhere between the two extremities, calculating the distance between the vertex and the segment is more complicated. This is described in [36].

2.2.10.4 Distance to a polyline

While the distance to segment algorithm is useful in determining the distance between a vertex and a segment bounded by two other vertices, this is often too restrictive for real problems. For example, a user would prefer to know the distance between a vertex and a set of segments forming a domain boundary rather than a single segment.

This algorithm is essentially an extension of the distance to segment algorithm. In [9, 36] it is proposed that when projecting the vertex in question to a segment, if the vertex projects to one of the segment's extremities, then one can assume that the closest location on the polyline is either on another segment or exactly on the projected location. If the projection is on the segment somewhere between the two extremities, we know that this segment is the closest and these geometric properties can be used.

There are two ways to find the distance to a polyline: brute force and neighbour propagation.

The brute force approach stipulates that for each point that we wish to test, we must calculate its distance from all segments composing the polyline in question and retain the minimal value. Using this approach, for each of the n vertices of the mesh we would have to calculate the distance to each of the m edges forming the polyline. This gives a complexity of $O(n \times m)$.

The first downside to this approach is that each vertex in the mesh must be tested by each algorithm. Unfortunately, most algorithms only require modifying a small subset of the vertices, so there are a lot of unnecessary computations.

The second downside is that each edge on the polyline must be examined for each vertex. This is unnecessary because polylines can span over a large area and the vertex in question could find the closest location on the polyline more quickly if some helpful information could be retained.

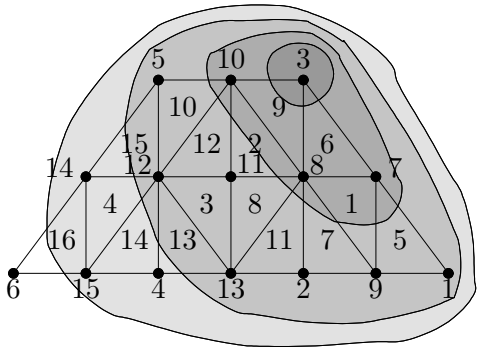
The other approach is to take advantage of neighbour propagation while retaining information regarding each vertex's localization on the polyline. This information is passed to its immediate neighbours as a guess. The idea is that the neighbour's closest segment is either the one passed by the neighbour vertex or one that is very close. This allows each point to find its closest segment on the polyline without having to compare each segment every time.

The neighbour propagation approach allows for each vertex to find its closest location on the polyline in O(1) time for each of the n vertices affected, giving the algorithm a complexity of $O(n)$.

2.2.10.5 Breadth-first search

In order to correctly propagate throughout the graph during the execution of an algorithm, neighbour propagation is used. This will be done using breadth-first search (BFS). The breadth-first search will be aided by a queue, which is a first-in first-out

(FIFO) data structure which determines the order in which vertices on the queue are processed. They are processed in the order that they are added to the queue, and any new vertices are pushed to the back of the queue.



LS1: 3

LS2: 7, 8, 10

LS3: 1, 9, 2, 11, 13, 5, 12

LS4: 4, 14, 15

LS5: 6

Figure 2.3: Example of breadth-first search on a triangular mesh, which is a planar graph.

Figure 2.3 illustrates BFS beginning with vertex 3 being assigned to the level $LS1$. This vertex has three neighbours, vertices 7, 8 and 10 which are assigned to level $LS2$. The vertices in $LS2$ have seven neighbours not already processed, vertices 1, 2, 5, 9, 11, 12 and 13. These vertices are assigned to $LS3$. The process continues until all vertices are assigned to a level.

Suppose that vertex 3 represents the source vertex in a distance to vertex algorithm. The same process can be applied using a queue. Initially vertex 3 is pushed onto the queue. The queue now contains $\langle 3 \rangle$. After popping and processing vertex 3, its three neighbours (vertices 7, 8 and 10) are pushed onto the queue. The queue now contains $\langle 7, 8, 10 \rangle$. Vertex 7 is popped from the queue. Its unprocessed neighbours are vertices 1 and 9. These vertices are pushed to the back of the queue. The queue now contains $\langle 8, 10, 1, 9 \rangle$. This process continues until all vertices are processed.

3 Riemannian metric

> The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them.
>
> Sir William Bragg

3.1 Metrics

3.1.1 Distance

Intuitively, the distance is a function. The letter d is usually used for this function. The inputs are two points in the space. The output is a positive number. This number is the length of the straight line between the two points. A more mathematical definition of the function d is as follows. Let $\mathbb{R}^2$ be the set of points in two-dimensional Euclidean space. Then the function $d : \mathbb{R}^2 \times \mathbb{R}^2 \mapsto \mathbb{R}$ defined as

$$d(P,Q) = d\left(\begin{bmatrix} P_x \\ P_y \end{bmatrix}, \begin{bmatrix} Q_x \\ Q_y \end{bmatrix}\right) = \sqrt{(P_x - Q_x)^2 + (P_y - Q_y)^2}$$

which is the well-known Euclidean distance between the two points P and Q. The distance function satisfies the usual following four properties:

(i) $d(P,Q) \geq 0$ for all P, Q in $\mathbb{R}^2$;

(ii) $d(P,Q) = 0$ if and only if $P = Q$;

(iii) $d(P,Q) = d(Q,P)$ for all P, Q in $\mathbb{R}^2$;

(iv) $d(P,Q) \leq d(P,R) + d(R,Q)$ for all P, Q, R in $\mathbb{R}^2$.

The first property says that the distance between two points is always positive or null. The second property says that the distance between two points is null if and only if the two points are the same. The third property says that the distance between two points is symmetric. The last property says that the distance between two points satisfies the triangle inequality.

3.1.2 Metric, a generalization of distance

*A **metric** is any function between two points that satisfies the above four properties.* Sure, the well-known usual Euclidean distance $\sqrt{(P_x - Q_x)^2 + (P_y - Q_y)^2}$ satisfies the four properties, so it is a metric, but there exists other functions that satisfy the four properties. All those functions are called metric and are some kind of generalization of the distance function.

One common generalization of the distance function is based on the scalar product. Any college student knows that the distance between points P and Q is the norm of the length of the vector from P to Q, and this norm is the square root of the scalar product of the vector $\vec{PQ}$ with itself. In short, we have

$$\begin{aligned} d(P,Q) &= \|Q-P\| = \sqrt{(Q-P)\cdot(Q-P)} = \sqrt{\langle (Q-P),(Q-P)\rangle} \\ &= \sqrt{\begin{bmatrix} Q_x - P_x \\ Q_y - P_y \end{bmatrix} \cdot \begin{bmatrix} Q_x - P_x \\ Q_y - P_y \end{bmatrix}} = \sqrt{(P_x - Q_x)^2 + (P_y - Q_y)^2}. \end{aligned}$$

We generalize the scalar product by introducing a symmetric matrix

$$\mathcal{M} = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

in the scalar product

$$\langle \mathbf{u}, \mathbf{v} \rangle_{\mathcal{M}} = \mathbf{u}^T \mathcal{M} \mathbf{v} = \begin{bmatrix} \mathbf{u}_x \\ \mathbf{u}_y \end{bmatrix}^T \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix} = a(\mathbf{u}_x \mathbf{v}_x) + b(\mathbf{u}_x \mathbf{v}_y + \mathbf{u}_y \mathbf{v}_x) + c(\mathbf{u}_y \mathbf{v}_y).$$

In the latter equation, if both $\mathbf{u}$ and $\mathbf{v}$ are the vector $\vec{PQ}$ from P to Q, then the generalized distance between P and Q is given by

$$\begin{aligned} d^2_{\mathcal{M}}(P,Q) &= \|Q-P\|^2_{\mathcal{M}} = \langle Q-P, Q-P\rangle^2_{\mathcal{M}} = (Q-P)^T\mathcal{M}(Q-P) \\ &= \begin{bmatrix} Q_x-P_x \\ Q_y-P_y \end{bmatrix}^T \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} Q_x-P_x \\ Q_y-P_y \end{bmatrix} \\ &= a(Q_x-P_x)^2 + 2b(Q_x-P_x)(Q_y-P_y) + c(Q_y-P_y)^2. \end{aligned} \tag{3.1}$$

Note that all the terms in the previous equation were squared to avoid writing the square roots. Note also that if the matrix $\mathcal{M}$ is the identity matrix, then $a = 1$, $b = 0$ and $c = 1$. In this case, the metric (3.1) is

$$d_{\mathcal{M}}(P,Q) = d_I(P,Q) = \sqrt{(Q_x-P_x)^2 + (Q_y-P_y)^2} = d(P,Q)$$

which is the well-known usual Euclidean distance between the two points.

3.1.3 Metric tensors

In the Eq. (3.1) of a metric, *the matrix $\mathcal{M}$ is called a* ***metric tensor***. This matrix must be symmetric (strictly) positive definite. These conditions are needed to satisfy the four properties. The matrix must be symmetric to have the symmetry, i.e., such that $d_{\mathcal{M}}(P,Q) = d_{\mathcal{M}}(Q,P)$ for all P and Q. It must be positive definite to have positivity, i.e., such that $d_{\mathcal{M}}(P,Q) \geq 0$ for all P and Q. It must be strictly positive definite to satisfy identity, i.e., such that $d_{\mathcal{M}}(P,Q) = 0$ if and only if $P = Q$.

A 2×2 symmetric positive definite matrix can be represented graphically by an ellipse. The two axes of the ellipse will be the two eigenvectors of the matrix. The two lengths of the ellipse axes will be the eigenvalues of the matrix. In the context of mesh generation and adaptation, representing metric tensors by ellipses is a convenient way to visualize the size, stretching and orientation of the mesh we wish to build. If the eigenvalues are large, the ellipse will be large and it means that we want very small triangles. If the eigenvalues are small, the ellipse will be small and it means that we

want very large triangles. Therefore metric tensors represent the density of the mesh, to be exact, the square density. For this reason, in the context of mesh generation and adaptation, metric tensors are also called *square density tensors*.

3.1.4 Size tensors

Size tensors are an alternative way of representing metric tensors. The first eigenvector of the matrix represents the orientation of one axis in the ellipse representing the tensor. The second eigenvector is perpendicular to the first one. The two eigenvalues represent the size of each axis. The resulting ellipse represents the size of the desired triangle at that position in the mesh.

Contrary to square density tensors, size tensors that are larger represent larger triangles, and smaller tensors imply smaller triangles.

In SmartMesh we use size tensors because their values directly represent the corresponding mesh size. This helps with readability because a user can easily determine mesh size by looking at the corresponding size tensor without any additional conversions of values.

3.1.5 Conversion from size tensor to metric tensor

While it is convenient to work with size tensors in SmartMesh, it is important to have the capability to convert them to metric tensors because having the capacity to express the mesh in both formats offers many advantages when outputting the results and interacting with other programs. In particular, this thesis will use $\mathsf{OO}\mathcal{RT}$ and this program needs metric tensors. This metric tensor $\mathcal{M}$ can be decomposed as

$$\mathcal{M} = P\Lambda P^T \tag{3.2}$$

where P is the permutation matrix composed of the unit eigenvectors in column, Λ is a diagonal matrix with the eigenvalues on the diagonal, and P^T is simply the

transpose of P. We can obtain the size tensor $\mathcal{S}$ from the metric tensor $\mathcal{M}$ as follows

$$\mathcal{S} = P\Lambda^{-2}P^T$$

where Λ^{-2} is the diagonal matrix obtained from Λ where all diagonal elements λ_i of $\mathcal{S}$ are substituted with $1/\lambda_i^2$.

3.1.6 Conversion from metric tensor to size tensor

Inversely, we can obtain the metric tensor $\mathcal{M}$ from the size tensor $\mathcal{S}$ as follows

$$\mathcal{M} = P\Lambda^{-2}P^T$$

where Λ^{-2} is the diagonal matrix obtained from Λ where all diagonal element λ_i is substituted with $1/\lambda_i^2$. The matrix P is the same as defined in Eq. (3.2).

3.1.7 Construction of size tensor

A size tensor in two dimensions has two eigenvalues λ_1 and λ_2 which represent the length of their respective eigenvector. The first eigenvector has components n_1 and n_2 while the second eigenvector is perpendicular to the first eigenvector and has components n_2 and $-n_1$.

3.1.7.1 Anisotropic size tensor

An anisotropic size tensor $\mathcal{S}$ can be constructed given two eigenvalues and their respective eigenvectors with the equation

$$\mathcal{S} = \begin{bmatrix} n_1 & n_2 \\ n_2 & -n_1 \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} n_1 & n_2 \\ n_2 & -n_1 \end{bmatrix}. \tag{3.3}$$

3.1.7.2 Isotropic size tensor

An isotropic size tensor $\mathcal{S}$ can be constructed given one size value s with the equation

$$\mathcal{S} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

3.2 Tensor intersection

While the use of size tensors in mesh adaptation offers the user a lot of flexibility in properly manipulating the mesh size, one must have a method to set the values of these size tensors. What if a user wishes to apply multiple sizes to the same vertex? How does one transform many size tensors into a single size tensor that correctly represents the size, stretch and orientation stored in a particular vertex? The solution is to use tensor intersection.

Given two input size tensors, the goal of an intersection operation is to find a new size tensor that represents what is common between its two inputs. There are many methods that determine the intersection of two tensors, each having advantages and disadvantages.

3.2.1 Alauzet metric tensor intersection

The Alauzet intersection method described in [3, 4, 6] is a series of algebraic operations on matrices. The goal is to simultaneously reduce the matrices until the result corresponds to an ellipse that is the intersection of the ellipses representing the two input matrices.

As an advantage there is no need to decompose the input matrices into ellipses in order to perform the calculations. However, this method has some severe drawbacks. The algorithm does not work if the two matrices are equal or if the ellipse representing one of the matrices is contained completely inside the ellipse of the other matrix.

3.2.2 Robust size metric intersection

Recall that a metric tensor can be represented as an ellipse. In the case of size tensors, which is what we use in SmartMesh, the ellipse directly represents the dimensions of the corresponding triangle. Therefore, if we intersect ellipses representing two different size tensors, the largest ellipse included in the intersection is the representation of the intersected size tensor. This intersected size tensor represents the dimension of the triangle that satisfies the two initial size tensors.

The intersection algorithm proposed in [28] does not have the drawbacks of the Alauzet method. The inspiration is to work with ellipses because they are geometric and directly represent the desired mesh size. This method works correctly for the cases where the Alauzet method fails.

Initially, each size tensor in the mesh is given a uniform size. Then, when an algorithm generates a new size tensor for a given vertex, the intersection algorithm is performed using the vertex's initial tensor along with the new one generated by the algorithm. The resulting tensor is stored as that vertex's new size tensor.

The inputs are two size tensors $\mathcal{E}_{\mathcal{M}_1}$ and $\mathcal{E}_{\mathcal{M}_2}$. The output will be the size tensor $\mathcal{E}_{\mathcal{M}_{1\cap 2}}$.

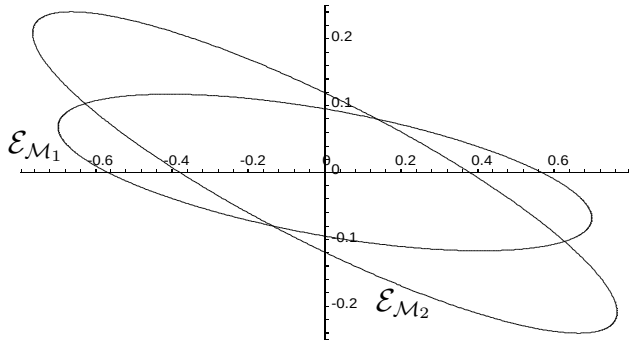


Figure 3.1: $\mathcal{E}_{\mathcal{M}_1}$ and $\mathcal{E}_{\mathcal{M}_2}$ in regular space.

The algorithm begins by transforming $\mathcal{E}_{\mathcal{M}_1}$ into a unit circle using a transformation

T. The same transformation T is then applied to $\mathcal{E}_{\mathcal{M}_2}$.

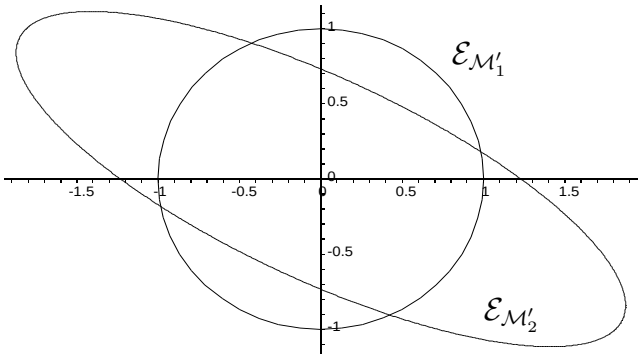


Figure 3.2: $\mathcal{E}_{\mathcal{M}'_1}$ (a unit circle) and $\mathcal{E}_{\mathcal{M}'_2}$ in T space.

At this point we have one of three scenarios:

- $\mathcal{E}_{\mathcal{M}'_1}$ is contained completely in $\mathcal{E}_{\mathcal{M}'_2}$. $\mathcal{E}_{\mathcal{M}'_{1\cap 2}}$ is assigned $\mathcal{E}_{\mathcal{M}'_1}$.
- $\mathcal{E}_{\mathcal{M}'_2}$ is contained completely in $\mathcal{E}_{\mathcal{M}'_1}$. $\mathcal{E}_{\mathcal{M}'_{1\cap 2}}$ is assigned $\mathcal{E}_{\mathcal{M}'_2}$.
- Neither $\mathcal{E}_{\mathcal{M}'_1}$ or $\mathcal{E}_{\mathcal{M}'_2}$ is contained completely inside the other. However we can see in Fig. 3.3 that we can easily calculate the dimensions of $\mathcal{E}_{\mathcal{M}'_{1\cap 2}}$. The axes of $\mathcal{E}_{\mathcal{M}'_{1\cap 2}}$ are the same as the axes of $\mathcal{E}_{\mathcal{M}'_2}$. In matrix notation, the eigenvectors of $\mathcal{M}'_{1\cap 2}$ are the same eigenvectors as the eigenvectors of $\mathcal{M}'_2$. The length of the axes of $\mathcal{E}_{\mathcal{M}'_{1\cap 2}}$ are the minimum between 1 and the length of the axes of $\mathcal{E}_{\mathcal{M}'_2}$. In matrix notation, the eigenvalues of $\mathcal{M}'_{1\cap 2}$ are the minimum between 1 and the eigenvalues of $\mathcal{M}'_2$.

Now, the transformation T^{-1} is applied to $\mathcal{E}_{\mathcal{M}'_{1\cap 2}}$ to return to the original state.

The ellipse $\mathcal{E}_{\mathcal{M}_{1\cap 2}}$ is the largest ellipse included in the intersection of $\mathcal{E}_{\mathcal{M}_1}$ and $\mathcal{E}_{\mathcal{M}_2}$.

Metric intersection is commutative. However, it is important to note that metric intersection is not associative. The final result will depend of the order in which three or more tensors will be intersected, although the difference is very minor. For more details see [31].

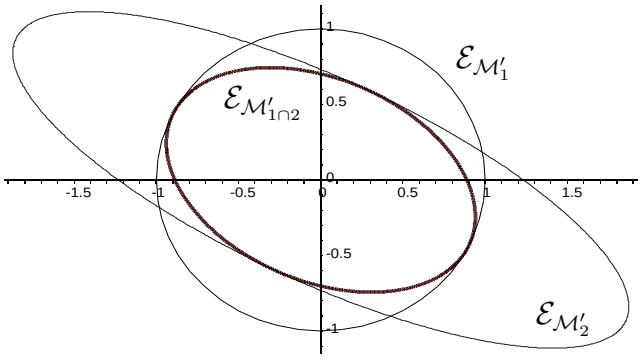


Figure 3.3: $\mathcal{E}_{\mathcal{M}'_{1\cap 2}}$ in T space.

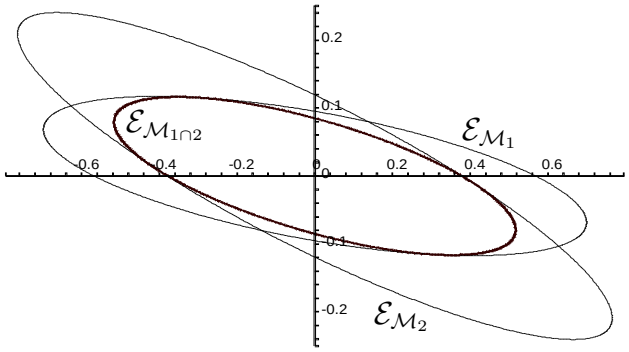


Figure 3.4: $\mathcal{E}_{\mathcal{M}_{1\cap 2}}$ in regular space.

If an implementation uses metric tensors instead of size tensors, there are two ways to proceed: one can first transform the metric tensors into size tensors, perform the intersection and transform the result back to metric tensors, or one can instead find the union of the ellipses representing the two metric tensors. The results of a union operation will give a new metric tensor that represents the combined density of both inputs.

3.3 Functions

In most problems it is essential to have a finer mesh at certain specific regions, which could be a geometric curve or a geometric vertex. It is also essential that the size

of triangles gradually increase as the distance from the geometric region in question increases. How does one determine the mesh size close to an entity of interest? How does one determine the rate of growth for mesh triangles as the distance from that location increases? One solution is to apply functions so that, given one or more inputs such as the distance or the curvature, one can translate these values into a size value that can be stored in the mesh.

SmartMesh needs functions to help determine the dimensions of size tensors throughout the mesh. A function can be used to manipulate one or both eigenvalues of a size tensor.

3.3.1 Distance function

In CFD, we want a mesh where the triangles are smallest at the key areas and gradually increase in size as the distance from those key areas increases. For this to be possible we need to define a distance function that can determine the sizes for size tensors based on the vertex's position relative to the key geometric entity in question. We also need to define the rate at which the size tensors affected by the function will grow as the distance increases.

Linear growth means that the rate is constant. Quadratic and cubic rates of growth means that the rate of growth is small close to the source but as we move away the rate of growth increases. Square root means that the rate of growth is high close to the source and the rate slows down as we move away. The implementation must take into account the different types of growth as well as other cases.

3.3.2 Curvature function

The reason for using curvature functions is because a triangular mesh will approximate a curve into a polyline and we want to limit the loss of precision where the curvature increases. In order to do this, we add more vertices on the curve where the curvature is high, and less vertices where the curvature is small or null (if the curve is straight).

All vertices are on the curve while the mesh edges linking these vertices may not be. See [15] for more details.

An example of a method to calculate the curvature of a polyline is to create a circle that passes by the vertex in question and its two immediate neighbour vertices on the same curve. The radius of this circle is inversely proportional to the curvature. That is, the smaller the circle, the larger the curvature. This is illustrated in Fig. 3.5.

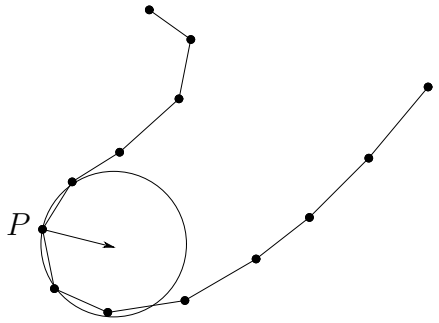


Figure 3.5: Curvature of a polyline at vertex P.

3.4 Metric orientation between two boundaries

When two boundaries in geometric domain are very close, the narrow opening between the two can be a sensitive area when doing numerical simulations. Ideally, metric tensors should be aligned with the nearest boundary. However, in this case both boundaries should have an influence on metric tensor orientation.

The implementation of this idea must ensure that the orientation of a given metric tensor follows the orientation of the two boundaries, but the closest boundary must have more of an influence on the final orientation.

3.5 Object Oriented Remeshing Toolkit (OO$\mathcal{RT}$)

OO$\mathcal{RT}$ is a program that receives metric tensors for all mesh vertices and converts the input mesh into a new mesh that better suits the requirements specified by the

metric tensors. Because of the limitations explained in Section 2.2.6, it is impossible for $OO\mathcal{RT}$ to build a mesh that perfectly satisfies the metric tensors. In some cases the tool may be forced to make some approximations, especially in areas where there are extreme size differences between neighbouring vertices.

Because $OO\mathcal{RT}$ cannot read size tensors, SmartMesh must convert its output to metric tensors so that the data can properly be imported.

4 SmartMesh library

> Any problem in computer science can be solved with another level of indirection.
>
> David Wheeler

4.1 Philosophy

The SmartMesh library contains all the tools necessary to set and modify the size tensors of each vertex and output a field of metric tensors such that a remesher such as $OO\mathcal{RT}$ (Section 3.5) can adapt the mesh to the user's requirements. The primary components of the SmartMesh library are the mesh, the topological part of the geometric model, the size functions and the algorithms.

However, the library does not drive the execution. It simply contains facilities to help a calling program perform the necessary tasks. An example of such a program that uses the SmartMesh library is the SmartMesh program described in Chapter 5.

4.2 Working environment

The creation of SmartMesh was facilitated by the use of many available tools that helped build and document this library while also allowing interaction with other programs.

4.2.1 Programming

SmartMesh is a library written in C++ on Fedora Linux 12. It takes advantage of object-oriented (OO) capabilities provided by C++. It has 6400 lines of code in 46 files. The code is stored on the Mathematics and Computer Science Department's Subversion (SVN) server located at `http://svn.cs.laurentian.ca`.

The Fedora system was installed in Virtual Box within Windows 7. The virtual PC setup allowed to do code editing in Fedora while doing other tasks simultaneously in Windows.

The editor used for code generation is Eclipse. This tool was very useful because of features such as automatic project-wide application of variable name changes and auto-completion, as well as good colouring of various key words.

4.2.2 Visualization

SmartMesh outputs a field of metric tensors that can be used as input for a remesher such as $\mathsf{OO}\mathcal{RT}$ in order to have the mesh modified appropriately. This field of metric tensors can also be visualized. The primary visualization tool used was `Vu`, which translates the input tensor for each vertex into a visible ellipse. The tensors could be size tensors or metric tensors. This is very useful when debugging to make sure the size and orientation of ellipses are correct. $\mathsf{OO}\mathcal{RT}$ outputs a modified mesh which can also be visualized by `Vu`.

4.2.3 Documentation

The SmartMesh library makes use of `Javadoc` comments to document the code and give information to other users. The `doxygen` program makes it possible to extract `Javadoc` comments and object-oriented structures to output a full document in both HTML and LaTeX formats. The `doxygen` output for SmartMesh can be found in Appendix C.

4.2.4 Logging

During the development of SmartMesh, the use of logging was important for debugging errors and also for printing important information to ensure that the work done by the SmartMesh library was correct.

Logging in SmartMesh is done using the `cout` command. When a program calls the SmartMesh library it is responsible for indicating where to redirect the logging output. By default the log messages are displayed on the screen, but they can be redirected to a file. If the user does not want to use logging, he can redirect the logging to `/dev/null` on Linux systems. This an easy, but inefficient, way to manage logging. SmartMesh is five times faster when redirecting the logging to `/dev/null` than when it outputs to a log file. Therefore a better management of logging should be implemented.

4.3 Object-oriented structure

The SmartMesh library takes advantage of object-oriented concepts. The various entities of the mesh and in the topological part of the geometric model are represented as classes. This facilitates storage and documentation of the many properties needed to be retained by each instance of each class.

4.3.1 Mesh representation

A triangular unstructured mesh is a planar graph composed of vertices and adjacencies between vertices. Usually, files contains only a list of vertices and a list of triangles, with each triangle being a list of references to three vertices. The `SM_MESH` class (`SM` for SmartMesh) is designed to hold a list of vertices, a list of triangles and a list of edges. Other adjacencies between these entities can be stored if needed by some algorithms.

One triangle has three vertices and three edges. Each edge has two vertices and

touches one triangle if it is a boundary edge, or two triangles if it is not a boundary edge. Each vertex has an associated size tensor, a 2×2 symmetric positive definite matrix, which indicates the mesh size.

4.3.1.1 Edge creation

The input file has a list of vertices and triangles, but usually has no edges. Therefore they must be created from the information available. Since each triangle has three vertices, a mesh edge can be created for each combination of two vertices in the triangle. The three edges are (vertex 1, vertex 2), (vertex 2, vertex 3) and (vertex 3, vertex 1).

Mesh edges belong to at least one triangle, if they are boundary edges, and two triangles otherwise. Therefore, when creating edges from triangles it is possible to visit the same combination of vertices twice. It is important to first check if a given combination of vertices already exists in the mesh's list of edges in order to avoid duplicates.

4.3.1.2 Storage of mesh entities

The `SM_MESH` class must store separate lists of triangles, edges and vertices. Standard Template Library (STL) vectors are used for this storage. The use of vectors in this situation is favorable because the mesh is static, meaning that it does not get modified by the SmartMesh library. Vectors offer easy access of elements but have a high cost for removal operations, but since the mesh is static this is not an issue in SmartMesh.

Since the vertices are provided by the input file in a specific order, we store in each vertex its index in the mesh's vector of vertices. This index is the vertex identifier. Therefore, we can easily obtain a given vertex's position in the list using its identifier. By the same manner, each triangle in the file is not composed of three vertices, but of three vertex identifiers.

4.3.1.3 Neighbourhood structure

Each vertex in the mesh has a list of pointers to its adjacent edges that link the vertex to its immediate neighbour vertices. This structure makes it easy to traverse the mesh and is ideal for the neighbour propagation techniques used throughout the library.

4.3.2 Topological model representation

The topology of the geometric model contains the relations between entities of the geometric model. Topological entities, in two dimensions, are faces, edges and vertices. In three dimensions, there are also volumes. A topological face is bounded by an outer loop of topological edges. If the geometric model contains holes, then each hole is bounded by an inner loop of topological edges. Finally, each topological edge is bounded by two topological vertices.

Be careful not confuse topological faces, edges and vertices with mesh triangles, edges and vertices. If the context of a paragraph is about meshes, we will simply use triangles, edges and vertices. If the context of a paragraph is about topology, we will simply used faces, edges and vertices. If the context of the paragraph is about mesh and topology, then the words edge and vertex must be specified to belong to either the mesh or the topology.

The `SM_TOPOLOGICAL_MODEL` class implements the topological part of the geometric model. A topological face has a STL vector containing identifiers that refer to the topological edges that bound this face. A topological edge has the identifier that refers to each of the two topological vertices that confine this edge. A topological vertex has no descendants in this structure and therefore does not need to store the identifier of other topological entities.

4.3.3 Mutual knowledge between mesh and topological model

Recall that the mesh is a discretization of the geometric model and that the mesh is body-fitted. So each part of the mesh, triangles, edges and vertices, belongs to some topological entities of the geometric model. The mesh given in the input file contains this relation. Each mesh vertex in the input file has an identifier referencing to either a topological vertex, a topological edge or a topological face. The same way, each mesh triangle has an identifier referencing to a topological face. When building the mesh edges, as explained in Section 4.3.1.1, an identifier to a topological edge or a topological face must be attributed to the mesh edge.

Conversely, the `SM_TOPOLOGICAL_MODEL` class will contain the inverse relation. Each topological entity, face, edge and vertex, will store all the mesh entities, triangles, edges and vertices that belong to it. For topological faces, three STL vectors are used to store the list of associated mesh triangles, mesh edges and mesh vertices. This includes all mesh entities that belongs to the topological face and its boundaries. For topological edges, two STL vectors are used to store the list of associated mesh edges and mesh vertices. This includes all mesh vertices located on this topological edge and the boundary mesh edges that link these mesh vertices together. For each topological vertex, a single pointer to the corresponding mesh vertex is stored.

Recall that the geometric model is steady. There will be different meshes for a given geometric model. In short, meshes will vary, but the geometric model will not. When defining functions or algorithms to be applied on the mesh, it is easier to define them using the topological entities of the geometric model. From there, the targeted topological entities know their associated mesh entities and neighbour propagation can begin (Section 2.2.5).

4.3.4 Size tensor representation

In SmartMesh the `SM_SIZE_TENSOR` class will not store the actual 2×2 positive definite matrix, but rather its decomposition into eigenvectors and eigenvalues. The two unit eigenvectors will be stored in a 2×2 rotation matrix while the eigenvalues will be stored in two strictly positive size variables, the first corresponding to the first eigenvector, while the second corresponds to the second eigenvector. If a size tensor is really needed, then it is sufficient to multiply the rotation matrix by the diagonal matrix composed of the eigenvalues, and then by the transpose of the rotation matrix (see Eq. 3.3). If a metric tensor is needed, then it is sufficient to multiply the rotation matrix by the diagonal matrix composed of the square of the inverse of the eigenvalues, and then by the transpose of the rotation matrix (see Eq. 3.3).

In essence, the `SM_SIZE_TENSOR` class does not store a size tensor, but its decomposition into eigenvectors and eigenvalues. There are two motivations behind this choice.

First, the functions and algorithms, described later in this chapter, will compute two directions, typically one perpendicular to a boundary and one parallel to the boundary, and two sizes along those directions. This is the information stored in the `SM_SIZE_TENSOR` class which makes it is easy to create an object of that class without any computation.

The second motivation relates to the intersection of two size tensors described in Section 3.2.2. The first thing that this algorithm needs is the decomposition into eigenvectors and eigenvalues of the two input size tensors. Clearly, it is more convenient that the `SM_SIZE_TENSOR` class stores the decomposition of a size tensor instead of the size tensor itself.

4.3.5 Functions

In SmartMesh, functions serve as a bridge to translate geometric features of the geometric domain into a physical size that can be used as inputs to generate size tensors.

Depending on the type of functions, some attributes are construction parameters and do not change during an algorithm's execution, while other attributes are variable depending on the context. Parameters must be provided for the constructor of the corresponding function class, and an execute method has the variable attributes as inputs before outputting the appropriate size value.

4.3.5.1 Distance function

The distance function class refers to Section 3.3.1. Parameters include the minimal size, maximal size, maximal distance and exponent value.

The only variable attribute is the current distance between a vertex of the mesh and a topological entity, which varies during the execution of an algorithm because each vertex is at a different location in the mesh. The parameter $d_{\max}$ is the function's maximal distance, $s_{\min}$ is the function's minimal size, s_{max} is the function's maximal size, e is the function's exponent value and d is the input distance. The output size s can be defined as

$$s(d) = s_{\min} + \left(\frac{d}{d_{\max}}\right)^{e} (s_{\max} - s_{\min}) .$$

If $0 \leq d \leq d_{\max}$, then $s_{\min} \leq s(d) \leq s_{\max}$.

4.3.5.2 Size projection function

This function is similar to the distance function, but the minimal size is variable and must be supplied along with the local distance to compute the local size value.

This function is useful when size values s_0 along a boundary are variable, and ensures that at the maximal distance the size will be set to the maximal size.

The conversion from distance to size can be described as

$$s(d, s_0) = s_0 + \left(\frac{d}{d_{\max}}\right)^e (s_{\max} - s_0)\,.$$

If $0 \leq d \leq d_{\max}$, then $s_0 \leq s(d, s_0) \leq s_{\max}$.

4.3.5.3 Curvature-to-size conversion

This function is responsible for translating a curvature into a physical size that will be useful for the curvature of boundary algorithm in Section 4.3.7.2. Typically this is only used along a boundary, and a size projection function is used for points between the boundary and the max distance.

The user supplies a minimal size $s_{\min}$, a maximal size $s_{\max}$, as well as an error value δ between a curve and its linear approximation. From there, a minimal curvature $\kappa_{\min}$ and maximal curvature $\kappa_{\max}$ are computed. The minimal curvature corresponds to maximal size, while the maximal curvature corresponds to minimal size. If the curve is flat, the size should be infinite but will be bounded by $s_{\max}$. If the curve has a very strong bend, the size should be very small but will be bounded by $s_{\min}$.

The variable attribute is the local curvature κ at a given point on the boundary. This is supplied by the algorithm and the output is the size $h(\kappa)$ of the segment that has an error δ with the curve. See figure 4.1.

First, the radius r is calculated as $1/\kappa$. The conversion from curvature to size is given by

$$h(r(\kappa)) = 2\sqrt{\delta(2r - \delta)},$$

deduced from [17]. The output size $h(\kappa)$ must be bounded such that $s_{\min} \leq h \leq s_{\max}$.

4.3.6 Algorithm representation

The main `SM_ALGORITHM` class is an abstract class with the only shared property being the `execute` method. This method has no parameters and triggers the full

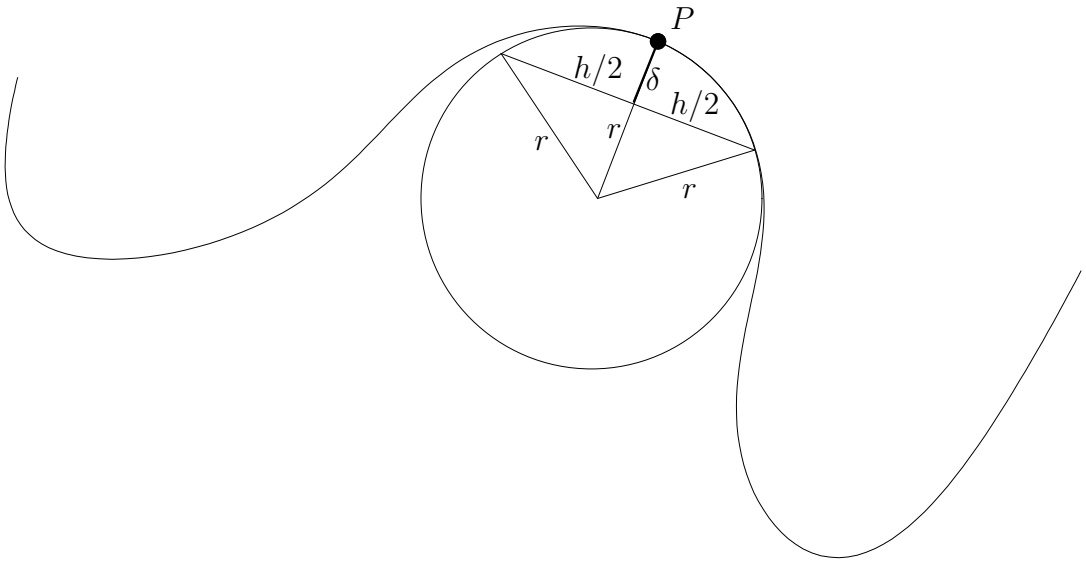


Figure 4.1: A curve at the point P is approximated by the osculating circle of radius r. The length h of the segment that has a maximal error δ with the circle.

execution of any algorithm. This gives inheriting classes the freedom of customizing their construction inputs and class attributes.

A list of topological entities is to be provided at construction time. The constructor of the algorithm in question will extract the associated mesh vertices for those topological entities. The extracted mesh vertices will serve as starting points for the algorithm. From there, neighbour propagation can be accomplished by traversing the mesh through each vertex's neighbouring edge list.

The caller of any algorithm sub-class needs only to call the `execute` method. This will execute the algorithm in question from start to finish.

4.3.6.1 Uniform algorithm

This algorithm implementation refers to the definition in Section 2.2.10.1. The inputs are the desired uniform isotropic size s and the topological entities where the user wishes to apply this algorithm. The algorithm creates an isotropic size tensor of size s for each inputted topological entity's associated mesh vertices. An example of mesh

build from uniform size tensors is shown in Fig. 4.2.

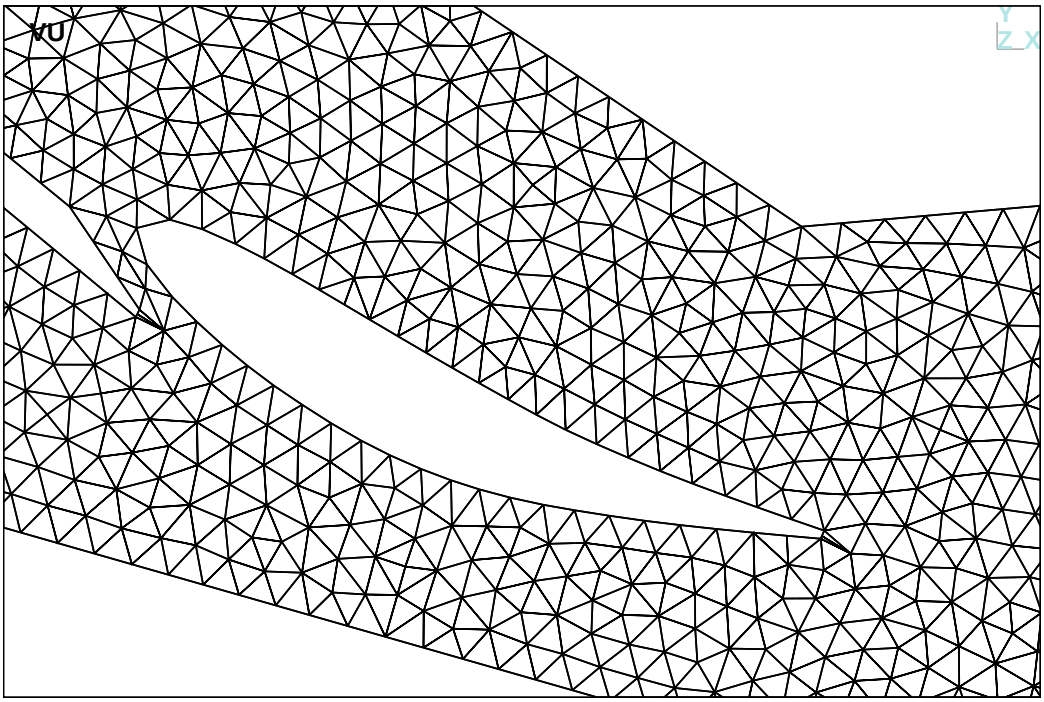


Figure 4.2: Example of uniform mesh of size s.

Due to the fact that the algorithm assigns a constant size, there is no need for any input functions and also no need for the application of neighbour propagation. Here is the logic of the uniform algorithm.

Algorithm 4.1: **Uniform Algorithm**. Input: A topological entity E and the isotropic size value s.

1: **for all** mesh vertices V belonging to topological entity E

2: Generate the isotropic size tensor $\mathcal{M}(s)$ using size value s.

3: Intersect $\mathcal{M}(s)$ with the size tensor $\mathcal{M}_V$ at vertex V.

4: **end**

Here are some remarks about the computation of size tensors (line 2). The input topological entity E of the geometric model can be any of the three types of topological

entity: vertex, edge or face. It does not matter. Each of the three types have a list of associated mesh vertices.

The size tensors created on line 2 will be isotropic as shown in Section 3.1.7.2. This means that the two eigenvalues receive the same value of s. The eigenvectors could be any set of two unit perpendicular vectors. The usual choice is simply $\mathbf{e}_1 = (0, 1)$ and $\mathbf{e}_2 = (1, 0)$.

4.3.6.2 Distance to vertex algorithm

This algorithm refers to the algorithm definition in Section 2.2.10.2. The inputs are the topological vertex to be used as start point and the distance function which is responsible for determining mesh size for vertices in relation to their distance from the topological vertex. See Fig. 4.3 for a more detailed look at the effect of the distance to vertex algorithm on a mesh.

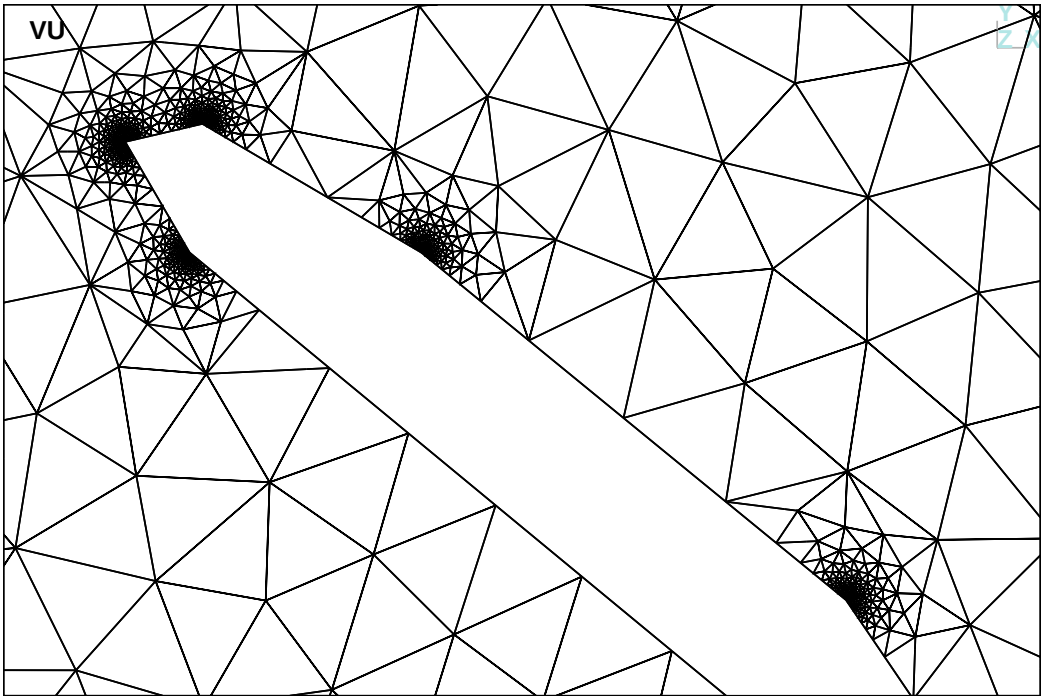


Figure 4.3: Example output for distance to vertex algorithm.

Here is the logic of the distance to vertex algorithm.

Algorithm 4.2: **Distance to vertex algorithm**. Input: A topological vertex E and a distance function with its parameters, amongst them d_{max}.

1: Push onto queue the mesh vertex associated to topological vertex E.
2: **while** queue not empty
3: Pop a vertex V from the queue.
4: Compute the distance d between V and E.
5: **if** $d \leq d_{\max}$ **then**
6: Generate the isotropic size tensor $\mathcal{M}(d)$ using the size function.
7: Intersect $\mathcal{M}(d)$ with the size tensor $\mathcal{M}_V$ at vertex V.
8: Push onto queue all neighbour vertices of V not already processed.
9: **end**
10: **end**

Here are some remarks about the algorithm. The input is a topological vertex E of the geometric model. But for all topological vertices, SmartMesh has a pointer to their unique associated mesh vertex.

Line 6 states that the distance d is used with the input distance function described in Section 4.3.5.1 to compute a local isotropic size tensor $\mathcal{M}(d)$ as defined in Section 3.1.7.2.

4.3.6.3 Common methods

The `SM_COMMON_METHODS` class serves as a container of methods that are required by other classes. This technique removes code duplication. The removal of code duplication reduces the risk of disaster when modifying this code because the change would only have to be applied and tested one time. It is also easier to maintain because all implementing classes will make use of the common code.

The only method in this class is a method that, given a mesh vertex V and a

mesh edge AB, calculates the distance d between the two entities (see Fig. 4.4). The method is simply to project the vertex on the infinite line spanned by the segment. If the projected point V' on the line is outside the segment AB, then the projected point is snapped on the corresponding end vertex of the segment. The return values are the actual distance d, the projection location of the vertex on the edge and the x and y components of the corresponding vector between the vertex and its projection on the edge.

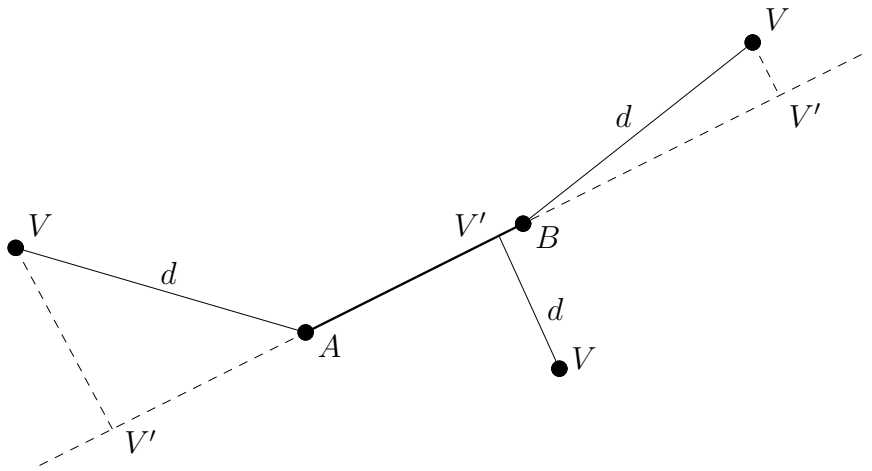


Figure 4.4: Distance between a vertex V and an edge AB. V' is the perpendicular projection of V on the line spanned by the edge. If the projection V' is outside the edge, the closest point is the corresponding end of the edge.

This method is useful for the implementing classes which need to locate the nearest edge of a boundary composed of many edges. The method is able to determine the distance between the two entities. The projection location on one edge decides if this edge is the closest. If the vertex projects inside the segment, then that edge is locally the closest. Otherwise it might be a neighbouring boundary edge that is closest. The algorithm moves from an initial guess edge towards the edge that minimizes the distance, marching across the edges one by one.

4.3.6.4 Distance between two boundaries algorithm

As mentioned in Section 3.4, when two mesh boundaries are located close to one another, it is necessary to provide a user with controls to manipulate the size and orientation of mesh triangles in this region.

For this algorithm, the inputs are the two topological edges that form the two boundaries used as sources for the execution, as well as two distance functions. An example of the mesh resulting from this algorithm can be seen in Fig. 4.5.

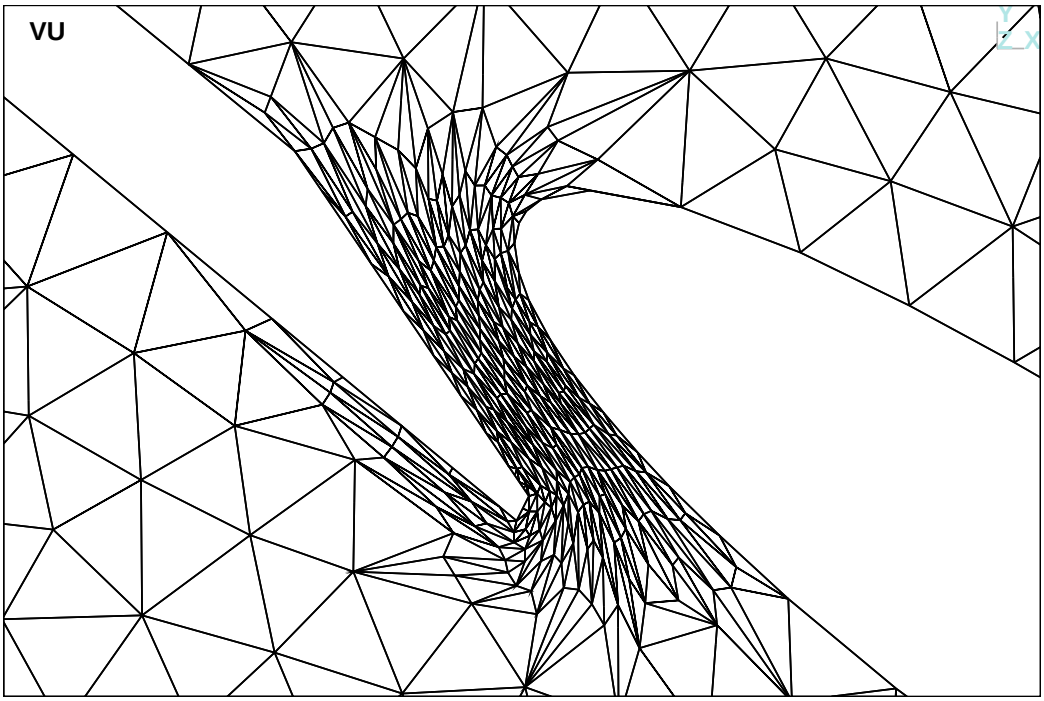


Figure 4.5: Example of a dense mesh stretched and aligned between two nearby boundaries.

Here is the logic for the distance between two boundaries algorithm.

Algorithm 4.3: **Distance between two boundaries algorithm**. Input: Two topological edges E_1 and E_2 and two size functions with their parameters, amongst them d_{max}.

1: **for all** mesh vertices V belonging to topological edge E_1
2: Compute distance d between V and E_2.
3: **if** $d \leq d_{\max}$ **then**
4: Push onto queue mesh vertex V.
5: Store information regarding localization of vertex V on topological edges E_1 and E_2.
6: Generate isotropic size tensor $\mathcal{M}(d)$.
7: Intersect $\mathcal{M}(d)$ with size tensor $\mathcal{M}_V$ at vertex V.
8: Push onto queue all unprocessed neighbours of vertex V.
9: Break out of this loop.
10: **end**
11: **end**
12: **while** queue is not empty
13: Pop a vertex V from the queue.
14: Get a neighbour vertex V_n that is already processed.
15: Using V_n's stored localization information, compute distance d_1 between V and E_1.
16: Using V_n's stored localization information, compute distance d_2 between V and E_2.
17: Compute d as the sum of d_1 and d_2.
18: **if** $d \leq d_{\max}$ **then**
19: Store information regarding localization of vertex V on topological edges E_1 and E_2.
20: Generate isotropic size tensor $\mathcal{M}(d)$.
21: Intersect $\mathcal{M}(d)$ with size tensor $\mathcal{M}_V$ at vertex V.
22: Push onto queue all unprocessed neighbours of vertex V.
23: **end**
24: **end**

Here are some remarks about the computation of distance (lines 5, 14 and 19) and computation of size tensor (line 6 and 20).

The input is a pair of topological edges E_1 and E_2 of the geometric model. But for all topological edges, SmartMesh has the list of mesh vertices and mesh edges that belong to the topological edges. Therefore, for SmartMesh, the topological edges E_1 and E_2 each correspond to a polyline in the geometric model. So, at line 14, computing the distance d between V and the topological edge E_1 or E_2 is in fact computing the distance d between V and a polyline. Computing the distance between a vertex and a polyline consists to find the edge of the polyline closest to the vertex V. Each time a distance is computed, the closest edge of the polyline is stored (line 5 and 19 in the algorithm). Each time a distance is computed, the search of the closest edge is not done over all the edges of the polyline, but with a neighbour search starting from an edge known to be close (line 14 in the algorithm).

Lines 3 and 14 state that geometric features such as distance d, with two input distance functions, will be used to compute a local size tensor $\mathcal{M}(d)$. This is not easy and needs more explanations. There are three cases.

First, if the vertex V is located on topological edge E_1, the distance to the topological edge, and so to the polyline that discretizes the topological edge, is zero. However, this means that the distance between V and E_2 is positive. In this case the orientation of the mesh edge associated to E_1 which is stored at V as closest location on the polyline is used to compute the orientation of a vector which is to be used in construction of a size tensor. Using one distance function described in Section 4.3.5.1, a size can be computed in this direction. Here the distance d becomes the distance between V and E_2. Using one distance function, d is translated into a size value which is used as the length of this vector. With the other distance function described in Section 4.3.5.1 and with d, a second size and vector can be computed in the direction perpendicular to the vector already computed. With these two vectors

and these two sizes, a size tensor can be computed as described in Section 3.1.7.1.

Second, if the vertex V is located on topological edge E_2, the distance to the topological edge, and so to the polyline that discretizes the topological edge, is zero. However, this means that the distance between V and E_1 is positive. In this case the orientation of the mesh edge associated to E_2 which is stored at V as closest location on the polyline is used to compute the orientation of a vector which is to be used in construction of a size tensor. Using one distance function described in Section 4.3.5.1, a size can be computed in this direction. Here the distance d becomes the distance between V and E_1. Using one distance function, d is translated into a size value which is used as the length of this vector. With the other distance function described in Section 4.3.5.1 and with d, a second size and vector can be computed in the direction perpendicular to the vector already computed. With these two vectors and these two sizes, a size tensor can be computed as described in Section 3.1.7.1.

The final case is when the vertex V is not on the edge E_1 or E_2. The variable V_1 is the closest point of V at the distance d_1, on the polyline that discretizes the topological edge E_1. The variable V_2 is the closest point of V at the distance d_2, on the polyline that discretizes the topological edge E_2. Using d as input to the two distance functions, a size value can be computed in both normal and tangent directions. Now that we have the necessary components, we can construct an anisotropic size tensor as defined in Section 3.1.7.1.

In this case, the orientation of size tensors is influenced by the orientation of the closest edge on each of the two boundaries. The final orientation is a weighted calculation based on the given vertex's proximity to each boundary. If the vertex is situated very close to one of the boundaries, its orientation will be very close to the orientation of the closest edge on that boundary. Let $\mathbf{d}_1$ be the vector between the vertex V and its closest point on the topological edge E_1, and let d_1 be the length of $\mathbf{d}_1$. In the same manner, let $\mathbf{d}_2$ be the vector between the vertex V and its closest point on the topological edge E_2, and let d_2 be the length of $\mathbf{d}_2$. Then we can define

a weighted normal $\mathbf{w}$ to both topological edges as

$$\mathbf{w} = d_2 \frac{\mathbf{d}_1}{d_1} - d_1 \frac{\mathbf{d}_2}{d_2}.$$

Note the minus in the previous equation. The normal $\mathbf{n}$ is simply the unit weight normal $\mathbf{n} = \mathbf{w}/\|\mathbf{w}\|$. See Fig. 4.6 for an intuitive visual description on how the computation of $\mathbf{n}$ is done. Perpendicular to the normal vector $\mathbf{n}$, we can define a tangent direction $\mathbf{t}$.

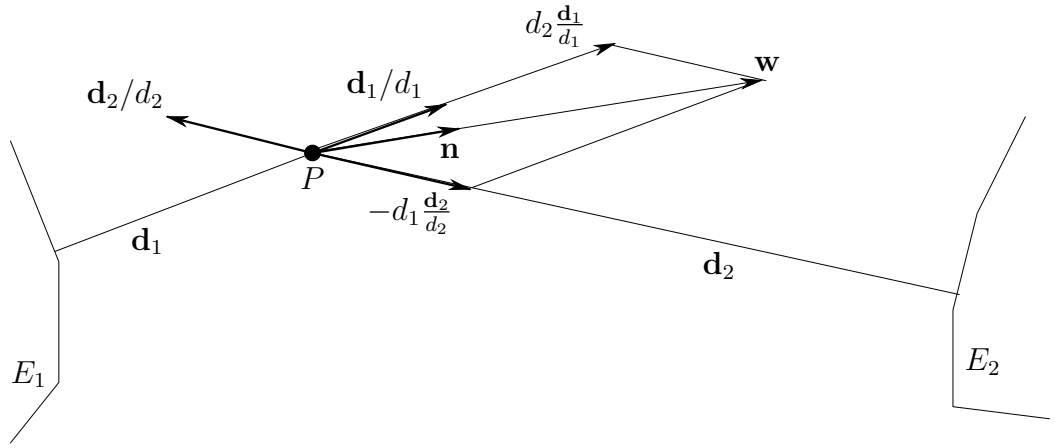


Figure 4.6: Orientation of a size tensor based on its proximity to two boundaries.

Using distance functions, size can be computed in both normal and tangent directions. Now that we have the necessary components, we can construct an anisotropic size tensor as defined in Section 3.1.7.1.

4.3.6.5 Distance to imaginary polyline algorithm

Sometimes a problem requires that geometric properties be calculated somewhere on a topological face. That is, the characteristics are not on a topological vertex or edge. The implementation of such an algorithm can be challenging because it is easy and convenient to associate mesh vertices of a topological edge or vertex and use them as start points for an algorithm execution. However in this case that cannot be done.

This algorithm has four inputs: the polyline which consists of user-specified vertex coordinates, two distance functions and a size projection function. One distance

function is used to determine the length of the vector parallel to the imaginary polyline for mesh vertices on the imaginary polyline. This length increases proportionally to the distance from the first vertex on the polyline.

The other distance function determines the length of the vector perpendicular to the imaginary polyline. The length increases proportionally to the distance from the closest segment on the imaginary polyline.

The projection function is used to determine the length of the vector parallel to the imaginary polyline for mesh vertices which are not on the imaginary polyline. The base length is the vector length at the closest imaginary polyline segment. This length increases proportionally to the distance between the mesh vertex in question and the closest imaginary polyline segment.

An example of the mesh resulting from the use of distance to imaginary polyline algorithm can be seen in Figure 4.7. The goal in this figure was to create a mesh suitable for a wake starting at the trailing edge of a blade.

Here is the logic for the distance to imaginary polyline algorithm.

Algorithm 4.4: **Distance to imaginary polyline algorithm**. Inputs: the polyline P, a topological face E and three size functions with their parameters, amongst them d_{max}.

1: Set $V_{\min}$ as the variable which holds the value of the closest mesh vertex of the face E to the polyline P.
2: Set $d_{\min}$ as the variable which holds the value of the smallest distance computed so far.
3: **for all** mesh vertices V in topological face E
4: **for all** edges P_e in polyline P
5: Compute distance d between V and P_e.
6: **if** $d \leq d_{\min}$ **then**
7: Store P_e in $P_{e_{\min}}$.

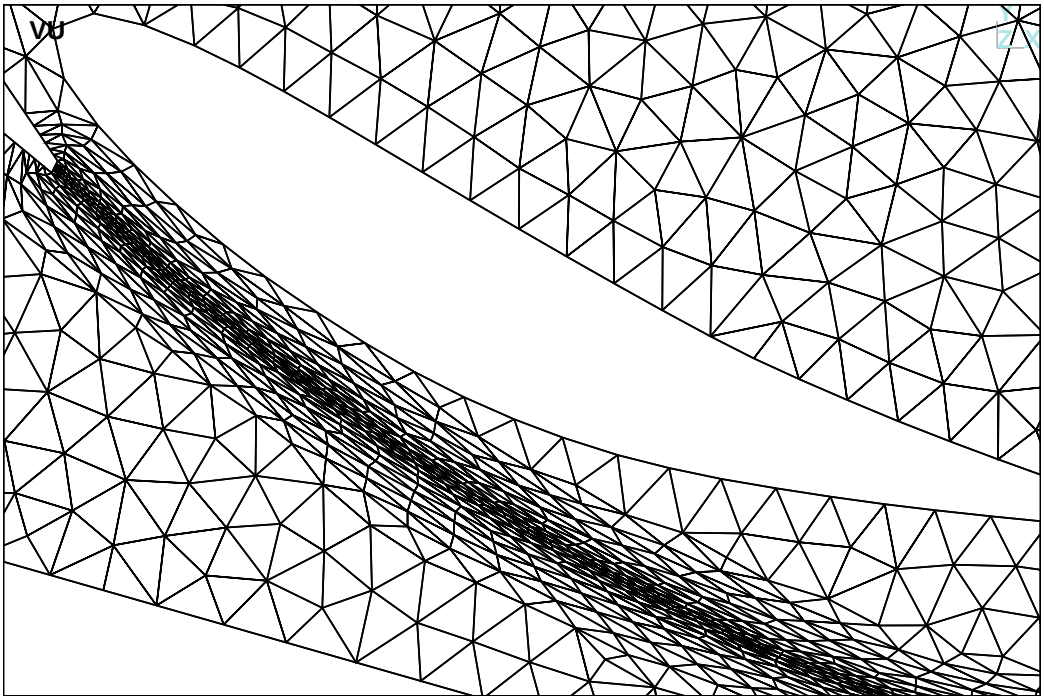


Figure 4.7: Example of a mesh produced with the distance to imaginary polyline algorithm.

8: **end**
9: **end**
10: **if** mesh vertex V is closer to polyline P than $V_{\min}$ **then**
11: Set $V_{\min}$ to V.
12: **end**
13: **end**
14: Push onto queue vertex $V_{\min}$.
15: **while** queue not empty
16: Pop a vertex V from the queue.
17: Get a neighbour vertex V_n that is already processed.
18: Using V_n's localization information, compute distance d between V and P.
19: **if** $d \leq d_{\max}$ **then**
20: Store information regarding localization of vertex V on polyline P.

21: Generate anisotropic size tensor $\mathcal{M}(d)$ using size functions.

22: Intersect $\mathcal{M}(d)$ with the size tensor $\mathcal{M}_V$ at vertex V.

23: Push onto queue all unprocessed neighbour vertices of V.

24: **end**

25: **end**

Here are some remarks about the computation of distance (lines 17 and 20) and computation of size tensors (line 21).

At line 20, computing the distance d between vertex V and a polyline P consists to find the edge of the polyline closest to the vertex V. Each time a distance is computed, the closest edge of the polyline is stored (line 17 in the algorithm). Each time a distance is computed, the search of the closest edge is not done over all the edges of the polyline, but with a neighbour search starting from an edge known to be close (line 20 in the algorithm).

Line 21 states that geometric features such as distance d, with two input distance functions, will be used to compute a local size tensor $\mathcal{M}(d)$. This is not easy and needs more explanations.

Since the polyline is not part of the geometric model, it is highly unlikely that a mesh vertex will be exactly on the polyline, meaning that d will be non zero. The point V' is the closest point of V at the distance d on the polyline P. The vector from V' to V defines a normal direction $\mathbf{n}$. A size in this direction, for a vertex at distance d of the boundary, is computed with the distance function described in Section 4.3.5.1.

Perpendicular to this normal direction $\mathbf{n}$, we can define a tangent direction $\mathbf{t}$. The size in this tangent direction is computed in three steps. First, we need to compute the curvilinear abscissa s' of the point V' on the polyline. The first vertex P_0 on the polyline is at curvilinear abscissa $s = 0$. Each vertex P_j on the polyline is at

curvilinear abscissa s_j given by

$$s_j = \sum_{i=0}^{j-1} \|P_{i+1} - P_i\|$$

which is the sum of the lengths of the previous edges prior to P_j. Knowing that the projected vertex V' on the polyline belongs to the edge P_jP_{j+1}, and knowing the curvilinear abscissa s_j of the vertex P_j, we can easily compute the curvilinear abscissa s' of V'.

Second, a distance function with its parameters will transform curvilinear abscissa s' into a size in the tangent direction at vertex V'. Finally, this size value at vertex V' on the polyline can be extended inside the domain at vertex V, at the distance d of the polyline, using the size projection function described in Section 4.3.5.2.

4.3.7 Boundary algorithm

This class is an abstract class that has common properties for the distance to boundary and curvature of boundary algorithms. Because the two algorithms have a lot in common, this class is more of a container of common code that is shared by the inheriting sub-classes. This way, if a modification needs to be made, it only needs to be made in one file rather than having to make the same change in multiple files.

For each vertex, we want to retain characteristics to share with neighbours and decrease the execution time of the algorithm. A common technique is to create an array mapping each vertex to a property. However, there are a number of problems with this approach. First, the algorithm only uses a sub-set of mesh vertices so there would be a lot of wasted space with the array approach. And second, if we want to store multiple values and of different data types, we need a better solution.

The solution is to use a STL map. Each map entry is composed of a key and a value. The key is a pointer to a mesh vertex. We will use the vertex identifier which is its index in the mesh. The value is a special inner-class which has the necessary attributes.

The inner class has several roles. It must hold a pointer to the closest boundary edge to assist in neighbour propagation, and it also has to hold the values computed during the algorithm execution because it is possible to visit the same vertex multiple times. By holding these computed values they can be used for comparison and are used at the end of the execution to compute new size tensors.

4.3.7.1 Distance to boundary algorithm

This algorithm refers to algorithm definition in Section 2.2.10.4. The first input is the topological edge that forms the polyline in question. Also, there are two distance functions inputted into this algorithm.

An example of a mesh created using the distance to boundary algorithm can be seen in Fig. 4.8.

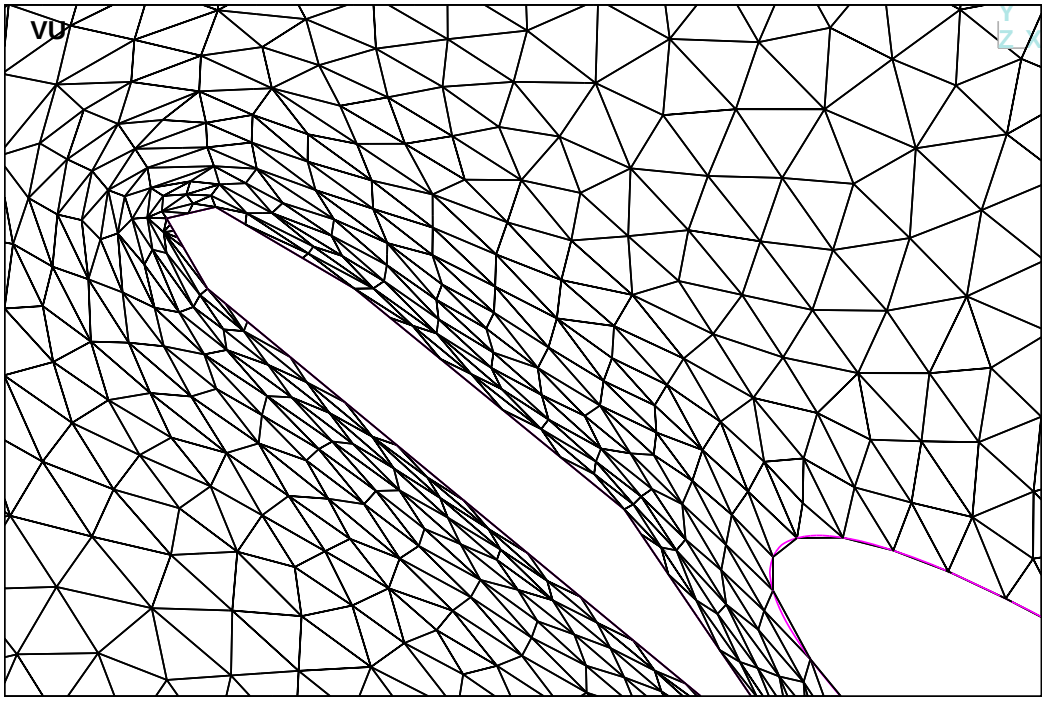


Figure 4.8: Example of a mesh created using the distance to boundary algorithm.

Here is the logic for the distance to boundary algorithm.

Algorithm 4.5: **Distance to boundary algorithm**. Input: The topological edge E and two distance functions with their parameters, amongst them $d_{\max}$.

1: **for all** mesh vertices V belonging to topological edge E
2: Set $d = 0$ because this vertex is on a boundary.
3: Generate anisotropic size tensor $\mathcal{M}(0)$ using the two size functions.
4: Intersect $\mathcal{M}(0)$ with the size tensor $\mathcal{M}_V$ at vertex V.
5: Store information regarding localization of vertex V on topological edge E.
6: Push onto queue all neighbour vertices of V not on E.
7: **end**
8: **while** queue is not empty
9: Pop a vertex V from the queue.
10: Get neighbour vertex V_n that is already processed.
11: Using V_n's stored localization information, compute the distance d between V and E.
12: **if** $d \leq d_{\max}$ **then**
13: Store information regarding localization of vertex V on topological edge E.
14: Generate anisotropic size tensor $\mathcal{M}(d)$ using size functions.
15: Intersect $\mathcal{M}(d)$ with the size tensor $\mathcal{M}_V$ at vertex V.
16: Push onto queue all unprocessed neighbours of vertex V.
17: **end**
18: **end**

Here are some remarks about the computation of distance (lines 5, 10 and 13) and the computation of size tensors (lines 3 and 14).

The input is a topological edge E of the geometric model. But for all topological edges, SmartMesh has the list of mesh vertices and mesh edges that belong to the topological edge. So, for SmartMesh, a topological edge E in the geometric model corresponds to a polyline. So, at line 10, computing the distance d between V and

the topological edge E is in fact computing the distance d between V and a polyline. Computing the distance between a vertex and a polyline consists to find the edge of the polyline closest to the vertex V. Each time a distance is computed, the closest edge of the polyline is stored (line 5 and 13 in the algorithm). Each time a distance is computed, the search of the closest edge is not done over all the edges of the polyline, but with a neighbour search starting from an edge known to be close (line 10 in the algorithm).

Lines 3 and 14 state that geometric features such as distance d, with two input distance functions, will be used to compute a local size tensor $\mathcal{M}(d)$. This is not easy and needs more explanations. There are two different cases. The first case, when the distance is zero in line 3, and the second case, when the distance is positive in line 14.

First, if the vertex V is on the topological edge E, the distance to the edge E, and so to the polyline that discretize the edge, is zero. The orientation of the mesh edge associated to E which is stored at V as closest location on the polyline is used to compute the orientation of a vector which is to be used in construction of a size tensor. Using one distance function described in Section 4.3.5.1, a size can be computed in this direction. As the distance is zero, this size is the minimum size set in this distance function.

With the other distance function described in Section 4.3.5.1, a second size and vector can be computed in the direction perpendicular to the vector already computed. As the distance is zero, this size is the minimum size. With these two directions and these two sizes, a size tensor can be computed as described in Section 3.1.7.1.

The second case is when the vertex V is not on the edge E, and so the distance d is strictly positive. Let V' be the closest point of V at the distance d, on the polyline that discretizes the input topological edge. The vector from V' to V defines a normal direction $\mathbf{n}$. A size in this direction, for a vertex at distance d of the boundary, is computed with the distance function described in Section 4.3.5.1.

Perpendicular to this normal direction $\mathbf{n}$, we can define a tangent direction $\mathbf{t}$. A

size can be computed in the tangent direction $\mathbf{t}$ using the distance function described in Section 4.3.5.1.

Now that we have the necessary components we can compute a new size tensor as described in Section 3.1.7.1.

4.3.7.2 Curvature of boundary algorithm

This algorithm is very similar to the distance to boundary algorithm. The difference, in terms of the inputs, is that this algorithm has three functions: a distance function, a curvature-to-size function and a size projection function.

We need to evaluate the curvature of a curve at any point. See Fig. 4.9 for a detailed image. The curve c is typically a curve from the geometric model. The geometric model is discretized by a triangular mesh therefore the curve c is approximated by piecewise linear edges forming a polyline p.

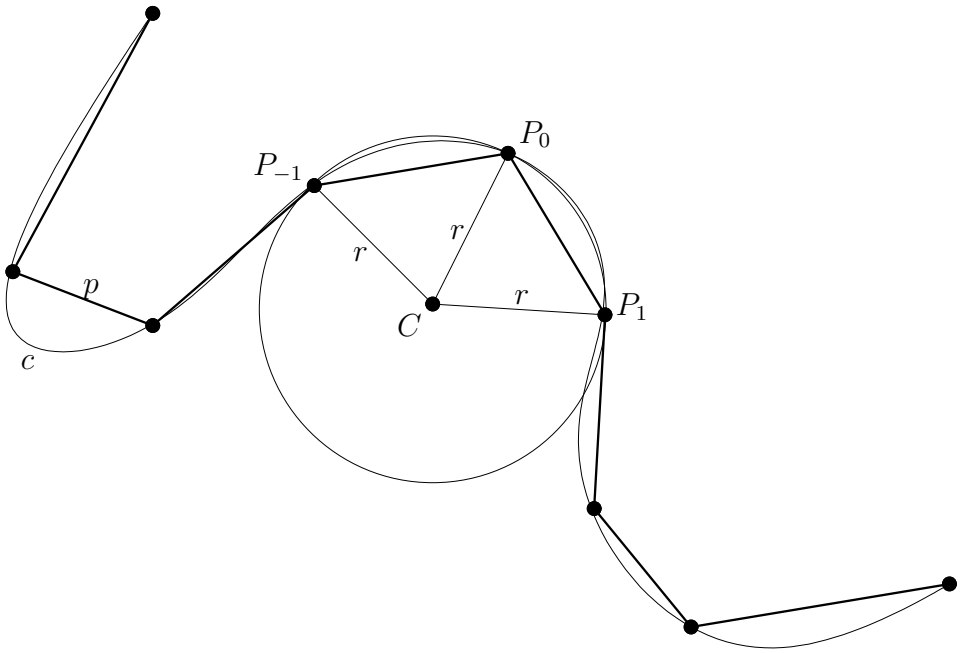


Figure 4.9: Approximation of the curvature of a discretized curve by computing the circle that passes through three points.

Let P_0 be the vertex of the polyline where we want to compute the curvature

while P_{-1} and P_1 are adjacent vertices to the vertex P_0 that belong to the curve c. Let x_{-1}, x_0 and x_1 be the respective x-coordinates for the three vertices and y_{-1}, y_0 and y_1 be the respective y-coordinates for the three vertices.

The curvature at vertex P_0 is defined as the inverse of the radius of the tangent circle to the curve at P_0. The tangent circle to the curve at P_0 is approximated by the circle that goes through the points P_{-1}, P_0 and P_1. To define this circle, we need to compute its center C of coordinates (x_C, y_C).

This center C is at equal distance to the points P_{-1}, P_0 and P_1. So this center must satisfy the equation

$$\begin{aligned} \|C - P_i\|^2 &= \|C - P_j\|^2 \\ (x_C - x_i)^2 + (y_C - y_i)^2 &= (x_C - x_j)^2 + (y_C - y_j)^2 \end{aligned}$$

where $-1 \leq i < j \leq 1$. This equation can be rewritten as

$$x_C(x_j - x_i) + y_C(y_j - y_i) = \frac{1}{2}(x_j^2 + y_j^2 - x_i^2 - y_i^2).$$

From this equation and with all combinations of integer values of i and j, we obtain the following system of three equations and two unknowns

$$\begin{bmatrix} x_0 - x_{-1} & y_0 - y_{-1} \\ x_1 - x_{-1} & y_1 - y_{-1} \\ x_1 - x_0 & y_1 - y_0 \end{bmatrix} \begin{bmatrix} x_C \\ y_C \end{bmatrix} = \frac{1}{2} \begin{bmatrix} x_0^2 + y_0^2 - x_{-1}^2 - y_{-1}^2 \\ x_1^2 + y_1^2 - x_{-1}^2 - y_{-1}^2 \\ x_1^2 + y_1^2 - x_0^2 - y_0^2 \end{bmatrix}.$$

This system has more equations than unknowns. It may have no solution and we may find only a least-squares solution using a QR decomposition. But in fact, the three equations are consistent up numerical round-off error and the third equation is a linear combination of the first two. So we can ignore one row and simply solve a 2×2 system.

Now that we have the center, it is easy to calculate the radius r as the distance between the center C and any of the three points. The curvature κ can be calculated as

$$\kappa = \frac{1}{r}.$$

The normal vector $\mathbf{n}$ at vertex P_0 can be calculated as

$$\mathbf{n} = \frac{C - P_0}{r}.$$

The normal and curvature exist for any interior points of a sufficiently smooth curve c. But now that this curve is approximated by a polyline p, the previous method gives only values of normal and curvature at vertex locations. What if we want values somewhere between the vertices? The solution used, and also the easiest one, is simply to do linear interpolation.

Figure 4.10 shows an example of what we want the curvature of boundary algorithm to accomplish. Note that the edges are smaller where the curvature is higher around to the leading edge of the blade.

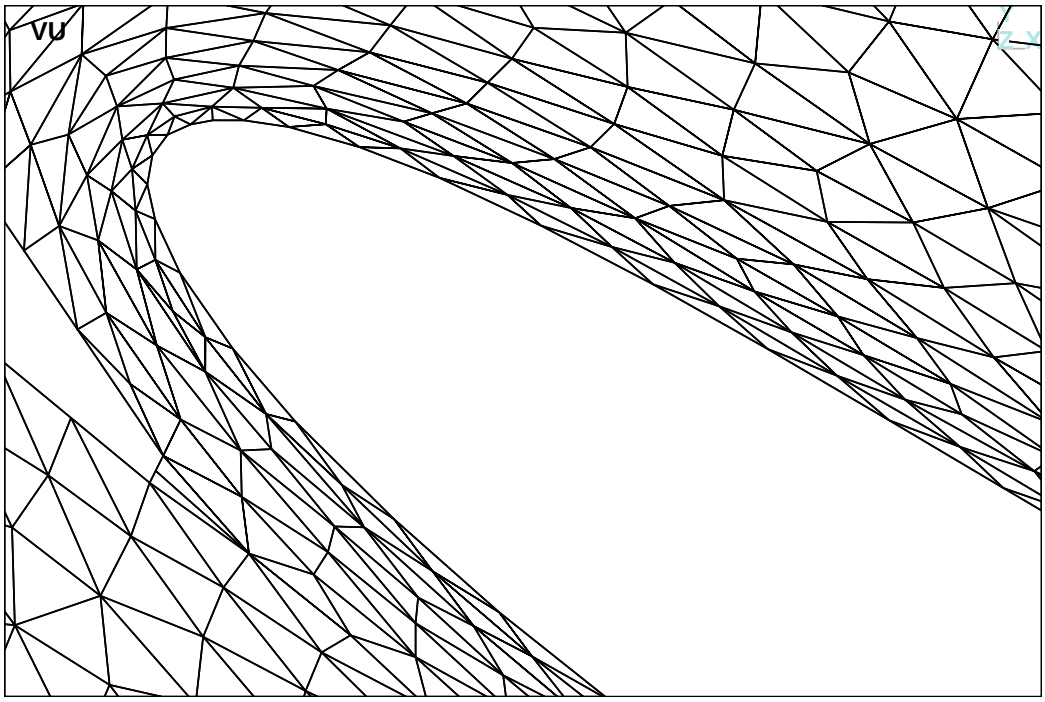


Figure 4.10: Example of a mesh produced by the curvature of boundary algorithm.

One obstacle in the implementation of this algorithm is the case of vertices at each end of a curve. These mesh vertices always corresponds to topological vertices of the geometric model. Mathematically the curvature is not defined at the ends of

the curve, however in SmartMesh there is still one size tensor for each vertex in the mesh. Therefore, it is necessary to define a curvature for these points so that the algorithm can be applied consistently.

In the case where the curve ends, we simply copy the curvature value of the neighbouring vertex on that curve. However, in practice it is possible for two different curves to end at the same vertex. Based on the algorithm, the topological point will select its two neighbours on the boundary in question, and the curvature is calculated. Unfortunately, this is not feasible because the two curves can have a very different orientation (see Fig. 4.11 where two boundaries are about perpendicular at their intersection).

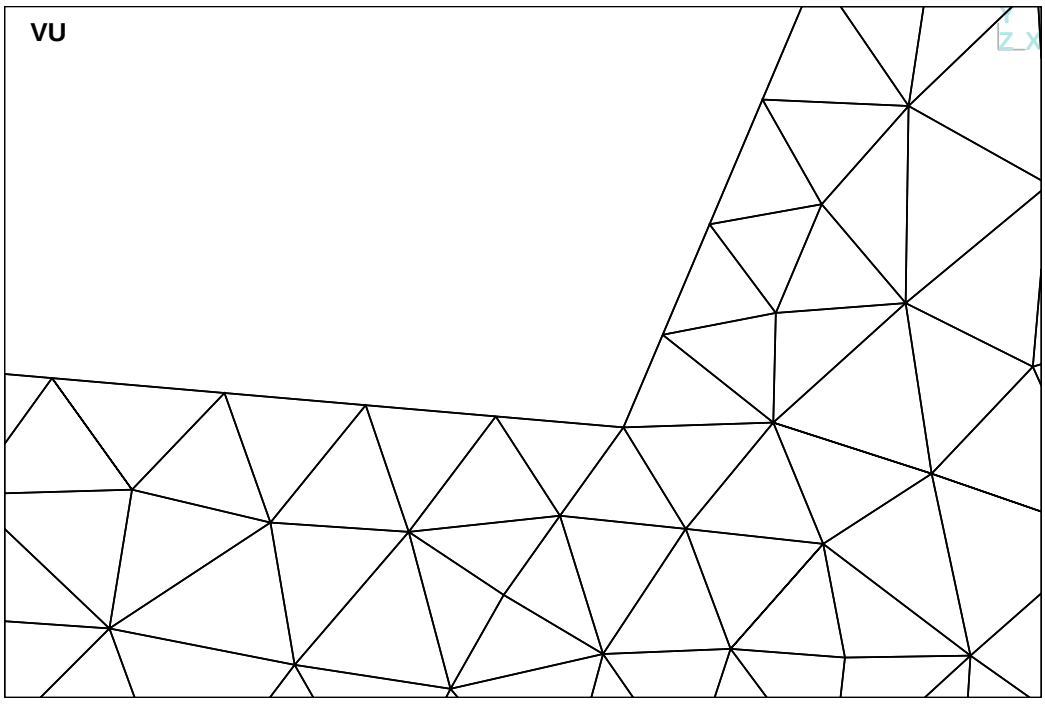


Figure 4.11: Two curves having a different orientation meeting at a common topological vertex.

Such extreme changes in the orientation of the two curves at the topological vertex leads to extremely high curvature values and therefore abnormally small triangles in that region. The lack of consistency in the environment can lead to undesired results.

Currently, the solution for such topological vertices is to have its size tensor orientation and size determined by the average orientation and size of the two neighbouring vertices on the boundary in question.

Here is the logic for the curvature of boundary algorithm.

Algorithm 4.6: **Curvature of boundary algorithm**. Input: A topological edge E and three size functions with their parameters, amongst them $d_{\max}$.

1: **for all** mesh vertices V belonging to topological edge E.
2: Compute and store the curvature κ and normal $\mathbf{n}$ at vertex V.
3: **end**
4: **for all** mesh vertices V belonging to topological edge E
5: Set $d = 0$ because this vertex is on the boundary.
6: Retrieve curvature κ and normal $\mathbf{n}$ at this vertex.
7: Generate the size tensor $\mathcal{M}(\mathbf{n}, \kappa, 0)$ using size functions.
8: Intersect $\mathcal{M}(\mathbf{n}, \kappa, 0)$ with the size tensor $\mathcal{M}_V$ at vertex V.
9: Store information regarding localization of vertex V on topological edge E.
10: Push onto queue all neighbour vertices of V not on E.
11: **end**
12: **while** queue is not empty
13: Pop a vertex V from the queue.
14: Get a neighbour vertex V_n that is already processed.
15: Using V_n's stored localization information, compute the distance d between V and the topological edge E.
16: **if** $d \leq d_{\max}$ **then**
17: Store information regarding localization of vertex V on topological edge E.
18: Generate the size tensor $\mathcal{M}(\mathbf{n}, \kappa, d)$ using size functions.
19: Intersect $\mathcal{M}(\mathbf{n}, \kappa, d)$ with the size tensor $\mathcal{M}_V$ at vertex V.
20: Push onto queue all unprocessed neighbour vertices of V.

21: **end**

22: **end**

Here are some remarks about the computation of distance (lines 9, 15 and 17) and the computation of size tensors (lines 7 and 18).

The input is a topological edge E of the geometric model. But for all topological edges, SmartMesh has the list of mesh vertices and mesh edges that belongs to the topological edge. So, for SmartMesh, a topological edge E in the geometric model corresponds to a polyline. So, at line 15, computing the distance d between V and the topological edge E is in fact computing the distance d between V and a polyline. Computing the distance between a vertex and a polyline consists to find the edge of the polyline closest to the vertex V. Each time a distance is computed, the closest edge of the polyline is stored (lines 9 and 17 in the algorithm). Each time a distance is computed, the search of the closest edge is not done over all the edges of the polyline, but with a neighbour search starting from an edge known to be close (line 15 in the algorithm).

Lines 7 and 18 state that geometric features such as normal $\mathbf{n}$, curvature κ and distance d, with three input functions, will be used to compute a local size tensor $\mathcal{M}(\mathbf{n}, \kappa, d)$. This is not easy and needs more explanations. There are two different cases. The first case, when the distance is zero in line 7, and the second case when the distance is positive in line 18.

First, if the vertex V is on the topological edge E, the distance to the edge E, and so to the polyline that discretize the edge, is zero. From results of this section, we can compute the normal $\mathbf{n}$ to the polyline at vertex V and the curvature κ, again at vertex V. Given the normal, we can deduce the tangent vector $\mathbf{t}$ perpendicular to the normal. Given the curvature, with a curvature to size function described in Section 4.3.5.3, a size can be computed in the tangent direction. With the other distance function described in Section 4.3.5.1, a size can be computed in the normal direction. As the

distance is zero, this size is the minimum size. With these two directions and these two sizes, a size tensor can be computed as described in Section 3.1.7.1. Note that in this case, we only need two functions to transform geometric characteristics of the boundary into a size tensor.

The second case is when the vertex V is not on the edge E, and so the distance d is strictly positive. Let the point V' be the closest point of V at the distance d, on the polyline that discretizes the edge. The vector from V' to V defines a normal direction $\mathbf{n}$. A size in this direction, for a vertex at distance d of the boundary, is computed with the distance function described in Section 4.3.5.1.

Perpendicular to this normal direction $\mathbf{n}$, we can define a tangent direction $\mathbf{t}$. From results of this section, we can compute the curvature κ at vertex V', which is on the boundary. Given this curvature, with the size projection function described in Section 4.3.5.2, a size can be computed in the tangent direction, on the boundary. This size, at vertex V' on the boundary, can be extended inside the domain, at vertex V at the distance d, using the size projection function described in Section 4.3.5.2.

Now that we have the necessary components we can compute a new size tensor as described in Section 3.1.7.1.

5 SmartMesh program

> Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.
>
> Rich Cook

5.1 Technical description

SmartMesh is also a program that calls the SmartMesh library. It is composed of 5 files totalizing 1500 lines of C++ code. The interface is a simple command line. It uses the C++ open source `tclap` library[11] to parse the command line. It uses the C++ open source `libconfig` library[12] to parse the configuration file.

The SmartMesh program uses the `pirate` library. `Pirate` is an in-house CAD system. It is a C++ library that reads, stores, manipulates and writes all the data used in numerical simulations. The data includes a CAD model with the topological model and the geometric model, two- and three-dimensional meshes, and solutions composed of scalars, vectors and tensors. It was developed mainly by François Guibault and Paul Labbé at École Polytechnique de Montréal from 1993 until now. It has now 111 000 lines of codes in 391 files.

[11] Available at `tclap.sourceforge.net`.

[12] Available at `sourceforge.net/projects/libconfig`.

5.2 Philosophy

The SmartMesh program is to be the bridge that links the user to the SmartMesh library defined in Chapter 4. Its role is to provide the user with facilities that allow to translate his knowledge into a personalized execution that will drive the mesh generation to produce a suitable mesh for the user's specific needs.

Although this program allows for customized executions, this is not a requirement. Another person could write a different program that is specific to a single geometric domain or a single application domain. Using this approach could make user training easier but the software would not be as flexible as the SmartMesh program.

The SmartMesh program is dependent of the CAD library `pirate`. This library can read from a file, store in memory, manipulate and write in a file, a geometric model and the triangular unstructured mesh that discretizes the geometric model. In particular, the SmartMesh program will use the `pirate` library to read a file that contains the geometric model and extract the topological part. The SmartMesh program will also use the `pirate` library to read a file that contains the mesh and extract the list of vertices and the list of triangles. Finally, the field of metric tensors computed by the SmartMesh library will be written in a file using the `pirate` format. A different program could interface with a different CAD library provided that the data is translated into SmartMesh internal format.

Recall that the SmartMesh library has no mechanism of execution. A calling program must be executed and make use of the library. This program is responsible for translating the inputs into SmartMesh library entities in order to solve the user's problem.

5.3 Configuration

While it is easy to provide the input data necessary to create the mesh structure and topological model in the SmartMesh library, it is not as easy to translate the user's

knowledge and requirements into a customized execution that solves the problems of a given application domain.

In order to solve this problem one must give the user an interface which allows the user to set the order of algorithms to be executed with their respective parameters.

A configuration text file can be used to receive the user's inputs which can then be read by the SmartMesh program to translate these inputs into a personalized execution. This configuration file must follow a specific syntax and provide key words that SmartMesh can read. Two options of configuration file formatting are XML and `libconfig`.

5.3.1 XML

Extended Markup Language (XML) is a syntax extended from Hypertext Markup Language (HTML) that encodes documents. Values are set by including an opening tag, one or more values, and a corresponding closing tag. The basic notation for XML is

<VARIABLE_NAME> VALUE </VARIABLE_NAME>.

Among its many uses, XML is used to create configuration files to set parameters for executions. Also, there exists many C++ libraries to parse XML files. One major disadvantage to using XML is that documents tend to become full of text due to the presence of tags. This hurts readability and can make a large configuration file more difficult to modify.

5.3.2 `libconfig`

`Libconfig` is an open source C++ library used for parsing configuration files. It is part of the standard distributions of Linux but is also available for most systems. Contrary to XML, `libconfig` offers a more readable syntax that looks a lot like a programming language.

`libconfig` provides a more clean grammar which reads itself in a way similar to a sentence in the English language. The basic notation for Libconfig is

VARIABLE_NAME = VALUE;

Values in `libconfig` can be expressed in many different formats, including numerical, text, lists of values and even more complex structures containing multiple types of values. An example of `libconfig` syntax can be found in Fig. 5.1.

```
1   CREATE_REGIONS =
2   (
3     { NAME = "CornerInlet";  TOPOLOGICAL_ENTITIES = ( [ 1, 2 ] ); },
4     { NAME = "CornerOutlet"; TOPOLOGICAL_ENTITIES = ( [ 3, 4 ] ); },
5     { NAME = "AllCorners";   TOPOLOGICAL_ENTITIES = ( [ 1, 2 ], [ 3, 4 ] ); },
6     { NAME = "Inlet";        TOPOLOGICAL_ENTITIES = ( [ 11 ] ); },
7     { NAME = "Outlet";       TOPOLOGICAL_ENTITIES = ( [ 12 ] ); },
8     { NAME = "Walls";        TOPOLOGICAL_ENTITIES = ( [ 11, 12, 13, 14 ] ); },
9     { NAME = "Fluid";        TOPOLOGICAL_ENTITIES = ( [ 100 ] ); }
10  );
```

Figure 5.1: Example of configuration file using the `libconfig` format.

In SmartMesh, we use `libconfig` to define the parameters of execution. While `libconfig` is able to parse the configuration file into tokens and associated values, these tokens have no context. An additional reader must be created on the SmartMesh end to convert these tokens and values into a full execution of the program.

In the example in Fig. 5.1, we are defining regions composed of various topological entities of the geometric domain. These regions are defined as a list of entries, each having a name and an array of topological entity identifiers.

Additionally, we must define the functions and algorithms to be used during the execution. Functions and algorithms are also defined as lists with each entry con-

taining the necessary attributes for constructor of the corresponding classes. For additional details on the SmartMesh `libconfig` syntax see Appendix A.

It is important to note that the contents in these lists are up to the user. That is, the user can control the execution.

5.3.3 Advantages of configuration file

There are many advantages to using this approach to configuring the execution. By allowing the user to choose which algorithms to execute and their respective parameters, the user is able to achieve the result that he desires. Two different configuration files can be created for the same geometric domain, with each file leading to a different result. Thus, the intelligence in SmartMesh comes from the user.

5.3.4 Preprocessor

SmartMesh has the capability to use the C preprocessor (`cpp`) on configuration files which allows users to make use of preprocessor directives such as `#define`. By allowing users to create such labels associated to values, it is possible to simulate the use of reusable variables in programming where a variable can be referenced by its name and the corresponding value is used in its place. Therefore, the same label can be invoked many times in a configuration file and the preprocessor will copy the corresponding value everywhere that it occurs in the file.

5.4 Execution of SmartMesh library

5.4.1 SmartMesh input data

The SmartMesh program receives two `pirate` files and one `libconfig` file. One `pirate` file contains a two-dimensional geometric model with the topological part and the geometric part. The SmartMesh program will extract the topological part

and will pass it to the SmartMesh library. The other pirate file contains a two-dimensional mesh that discretizes the geometric model. The SmartMesh program will extract the list of vertices, the list of triangles, the corresponding identifiers to topological entities and will pass these information to the SmartMesh library. Once the mesh and geometric model are created in the SmartMesh library, the program can begin executing algorithms with their parameters, as specified in the `libconfig` configuration file, in order to create an output field of metric tensors that can be given to a remesher such as $\mathsf{OO}\mathcal{RT}$ to create a new mesh.

Algorithm 5.4.1 is the general procedure for inputting data into SmartMesh prior to executing algorithms.

Algorithm 5.1: SmartMesh program. general layout of the program.

1: Read `pirate` file with the geometric model.
2: Read `pirate` file with the mesh.
3: Assert that the geometric model and mesh are two-dimensional.
4: Extract mesh vertices, their coordinates and their associated identifier to a topological entity.
5: Extract mesh triangles, their three pointers to vertices and their identifier to a topological entity.
6: Extract topology with topological faces, edges and vertices from the geometric model.
7: Create a SmartMesh `SM_TOPOLOGICAL_MODEL` object from extracted topology.
8: Create a SmartMesh `SM_MESH` object from extracted vertices and triangles.
9: Create mesh edges from the mesh triangles.
10: Create dual links between mesh entities (triangles, edges and vertices) and topological entities (faces, edges and vertices)
11: Read `libconfig` configuration file.
12: Execute algorithms with their parameters, as described in the configuration file.

13: Write in a `pirate` file the metric tensors, one for each vertex of the input file.

In the previous algorithm, additional details must be given regarding line 5. It is assumed that meshes will be two-dimensional, triangular and unstructured. If the input mesh contains some quadrilaterals, then, on the fly, each quadrilateral is decomposed into two triangles, and these two triangles will be passed to the SmartMesh library. SmartMesh library deals only with triangles. Suppose that the vertices of a quadrilateral are V_1, V_2, V_3 and V_4. Then the first triangle will be composed of the vertices V_1, V_2 and V_3, and the second triangle will be composed of the vertices V_1, V_3 and V_4.

In the previous algorithm, line 9 also needs more comments. Usually, `pirate` mesh files do not includes mesh edges. It is the duty of SmartMesh to build them from the triangles. This is not an obvious task, and therefore must be described in the following algorithm.

Algorithm 5.2: **BuildEdges Algorithm**. The input is the list of mesh triangles, each of them with their three pointers to vertices.

1: **for all** triangles T
2: Retrieve the vertices V_1, V_2 and V_3 of the triangle T.
3: **for** $edge = 1, 2, 3$
4: Compute $i = (edge - 1) \mod 3 + 1$
5: Compute $j = edge \mod 3 + 1$
6: Search an edge composed of vertices V_i and V_j in the mesh's list of edges
7: **if** the search fails **then**
8: Create E as a new edge composed of vertices V_i and V_j
9: Set left triangle of edge E to T
10: Add E to mesh's list of edges
11: Add V_j to neighbour list of V_i
12: Add V_i to neighbour list of V_j

13: **else**

14: Retrieve from the mesh's list of edges the edge E composed of vertices V_i and V_j

15: Set right triangle of edge E to T

16: **end**

17: **end**

18: **end**

5.4.1.1 SmartMesh output data

Once the SmartMesh execution is complete, a field of metric tensors, one for each mesh vertex, is written in an output `pirate` file. These metric tensors represent the desired size, stretch and orientation for the mesh to be built. This information can be visualized by a software such as `Vu` or given to a remesher such as $\mathsf{OO}\mathcal{RT}$ in order to create a mesh that satisfies as much as possible the information contained in the field of metric tensors.

5.4.2 `Layer`

`Layer` is a program that inserts quadrilateral structured grids around topological edges. In order to make room for this new grid, existing mesh vertices, edges and triangles must be pushed away from the boundaries. Since there is less space than previously, the vertices will be squeezed together more closely and triangles will be more stretched. In a post-processing step, each quadrilateral could be divided into two triangles. An example of a mesh with and without the added layers is shown in Fig. 5.2.

The goal of adding a structured layer of quadrilaterals is to have very long, thin and perpendicular elements aligned with boundaries. This is very critical in CFD for turbulence modeling. High Reynolds flows have very thin boundary layers and an

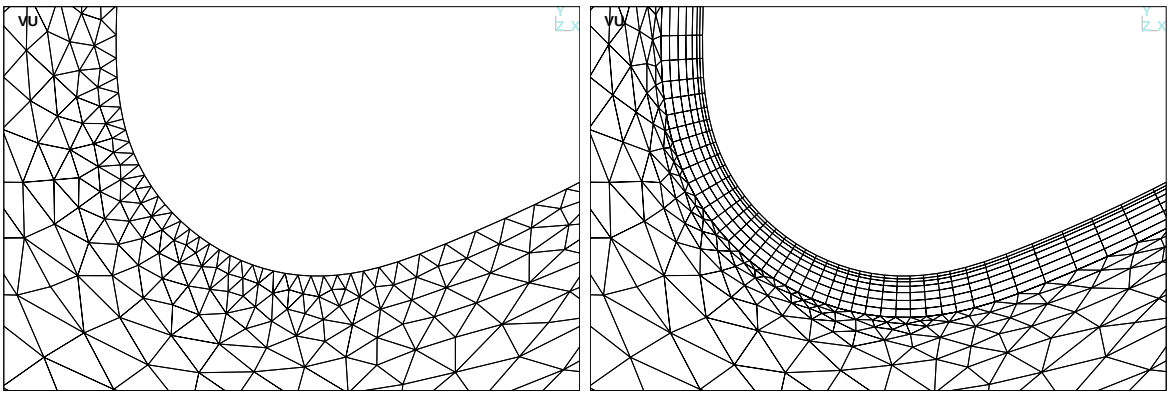


Figure 5.2: Example of mesh with and without an added structured layer of quadrilaterals.

appropriate mesh close to the boundaries is mandatory to achieve certifiable numerical simulations.

5.5 SmartMesh script

The SMARTMESH program can be executed to translate the inputs and user knowledge into SMARTMESH library components and execute a series of algorithms with customized parameters. The output field of metric tensors can be inputted into $OO\mathcal{RT}$ in order to build a new adapted mesh. However, this is not sufficient to create a mesh that fully captures the user's requirements.

Note that the output field of metric tensors from the SMARTMESH program execution represents the desired size, stretching and orientation of the mesh triangles at each mesh vertex. What if this mesh is very bad and coarse? When there is a large distance between vertices and their neighbours, what does the remesher program do in the space between those vertices where no metric tensor is defined?

Typically the remesher program will resolve this problem by applying a mesh size in this space by interpolating values form nearby vertices. But if the distance between neighbouring vertices is large, this is not an accurate solution. Figure 5.3 shows that the adapted mesh after a single iteration of the SMARTMESH program does

not satisfy the user's requirements. In particular, see the wake to the right of the two blades where a line of precise and stretched triangles should be located. Instead, we get inconsistent application of the algorithm because OO$\mathcal{RT}$ has insufficient metric tensors available to give an accurate result.

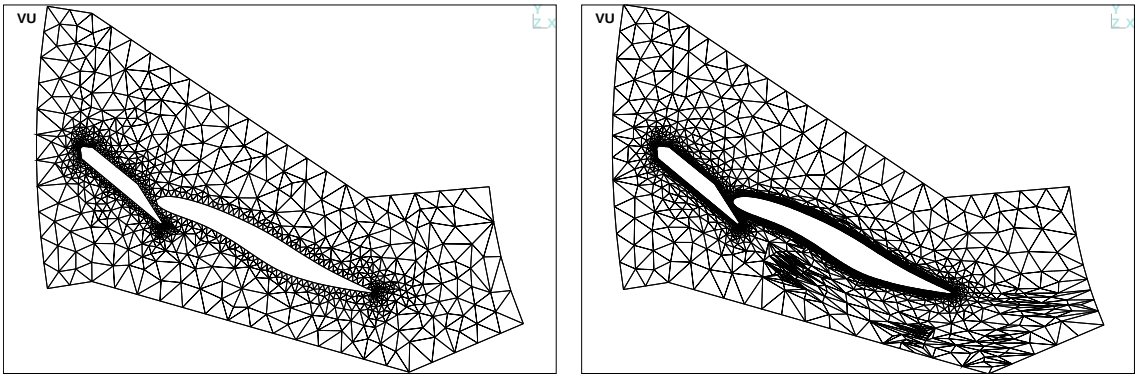


Figure 5.3: Initial mesh and corresponding adapted mesh following SmartMesh program execution.

What if SmartMesh could be re-executed with the newly adapted mesh using the same execution parameters? Surely, this would create a new field of metric tensors which OO$\mathcal{RT}$ can use to once again create a new adapted mesh. The iterative approach of re-executing the SmartMesh program using the latest adapted mesh as input helps the process converge towards a final mesh that satisfies the user's requirements specified in the `libconfig` configuration file.

The SmartMesh script is a script that takes advantage of the iterative approach. This is a 300-line Bourne shell script which calls the SmartMesh program several times, each time using the latest adapted mesh.

5.5.1 Shell script usage

```
SmartMesh.sh ....
--OORT-config-file=<file>
--SmartMesh-config-file=<file>
```

```
--Number-of-iterations=<int>
                    Default value = 1
--Debug=<0|1>
                    1 for log output
                    0 for no output (default)
--Use-Layer=<0|1>
                    1 to use of Layer program
                    0 to skip Layer (default)
--Geometric-model-file=<file>
--Mesh-file=<file>
```

The `OORT-config-file` parameter is a link to the file containing the configuration for OO$\mathcal{RT}$ to use.

The `SmartMesh-config-file` parameter is the configuration in `libconfig` format to be used by SmartMesh. This contains the customized instructions specified by the user.

The `Number-of-iterations` is the amount of times that the script will execute SmartMesh and OO$\mathcal{RT}$.

The `Debug` parameter tells to SmartMesh to output a log file if set to "1", and nothing otherwise.

The `Geometric-model-file parameter` is the `pirate` format file that contains the geometric model of the computational domain.

The `Mesh-file` parameter is the `pirate` format initial mesh to be given to SmartMesh.

The `Use-Layer` parameter is a switch that will call the program "Layer" if set to "1", and otherwise will skip this step.

5.5.2 Command-line output

```
[eric@Eric-Fedora test1] smartmesh
```

```
   --OORT-config-file=oort.cfg --SmartMesh-config-file=config1.txt \
   --Number-of-iterations=5 --Geometric-model-file=geometrie.pie \
   --Mesh-file=maillage_ini.pie

1- Parse command line entries

Begining of a script that performs a big loop over the
solver and the mesher. This script should end with:
   ---> HAPPY END OF THE BIG LOOP <---
If it is not the case, then the script failed......

Date of the begining: Sat Mar 19 19:43:18 EDT 2011
Current directory: /home/eric/SmartMesh/apps/PirateCommandLine/tests/test1
Current computer: Linux Eric-Fedora 2.6.32.26-175.fc12.x86_64 #1 \
   SMP Wed Dec 1 21:39:34 UTC 2010 x86_64 x86_64 x86_64 GNU/Linux

BEGINING THE BIG LOOP
  Doing iteration 001
  Doing iteration 002
  Doing iteration 003
  Doing iteration 004
  Doing iteration 005

ENDING THE BIG LOOP

End of this script at: Sat Mar 19 19:43:49 EDT 2011
The size of this directory is now 12444 kilo-bytes...

---> HAPPY END OF THE BIG LOOP <---
```

5.5.3 SmartMesh and OO$\mathcal{RT}$ working together

In an execution, SMARTMESH and OO$\mathcal{RT}$ work together to iteratively refine the mesh into the result that the user desires. SMARTMESH outputs a metric tensor for all mesh vertices. This becomes the input for OO$\mathcal{RT}$, which uses the metric tensors as a guide to adapt the mesh. The new mesh outputted by OO$\mathcal{RT}$ serves as an input for the next SMARTMESH program execution while using the same configuration file. Since OO$\mathcal{RT}$ cannot perfectly conform to the outputted field of metric tensors, it is necessary to make multiple iterations of executing SMARTMESH and then OO$\mathcal{RT}$ in order to converge towards the correct answer.

5.5.4 Logic of SmartMesh script execution

Here we can see the logic of a SMARTMESH execution. In essence, the output of SMARTMESH program is the input of OO$\mathcal{RT}$, and the output of OO$\mathcal{RT}$ is the input of SMARTMESH. This cycle continues for the amount of iterations desired by the user.

Algorithm 5.3: **SmartMesh.sh**. Input: a geometric model (with topological and geometrical parts) and an initial mesh that discretizes the geometric model. Output: an unstructured triangular intelligent mesh.

1: Set initial mesh M_1

2: **for** $i = 1, 2, \ldots, MaxIter$

3: Execute `SmartMesh.exe` with the mesh M_i and the topological model as inputs. This will generate the output field of metric tensors $\mathcal{M}_i$.

4: Execute `oort.exe` with the mesh M_i, the field of metric tensors $\mathcal{M}_i$ and the geometric model as inputs. This will generate the adapted mesh M_{i+1}.

5: **end**

6: **if** the user wishes to use `Layer` program **then**

7: Call `Layer` with mesh $M_{MaxIter}$ as input. The output will be a modified mesh $M_{MaxIter}$ with structured layers added where desired.

8: **end**

5.5.5 Evolution of the input mesh

The following figures show the evolution of a mesh during the iterations of a SmartMesh script execution. This example shows the convergence towards the final mesh in five iterations. Clearly, one iteration is not enough and there is clearly an improvement in continuing to execute the loop with SmartMesh and OO$\mathcal{RT}$.

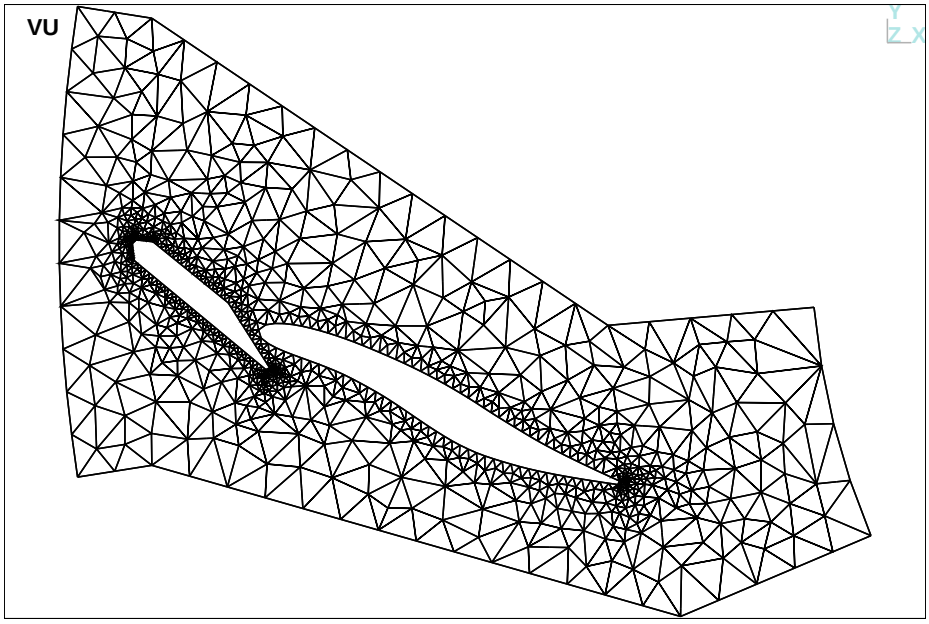


Figure 5.4: Initial mesh.

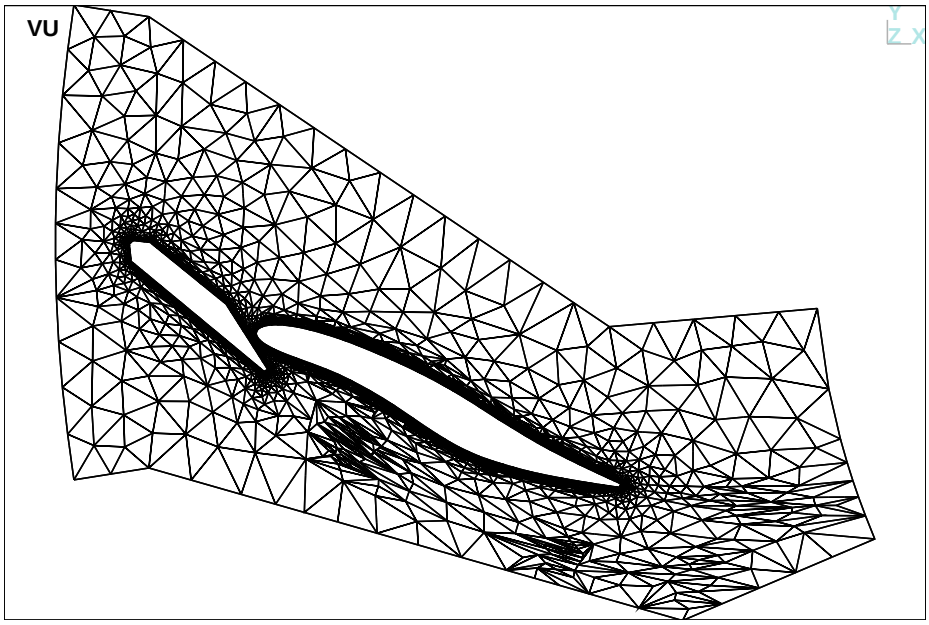


Figure 5.5: Step 1 in the SmartMesh script execution.

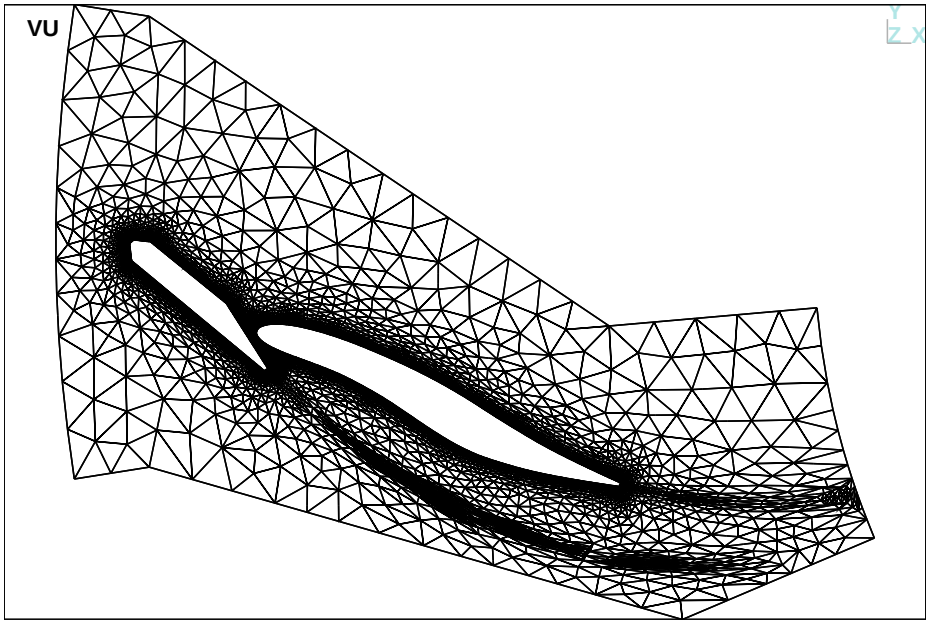


Figure 5.6: Step 2 in the SmartMesh script execution.

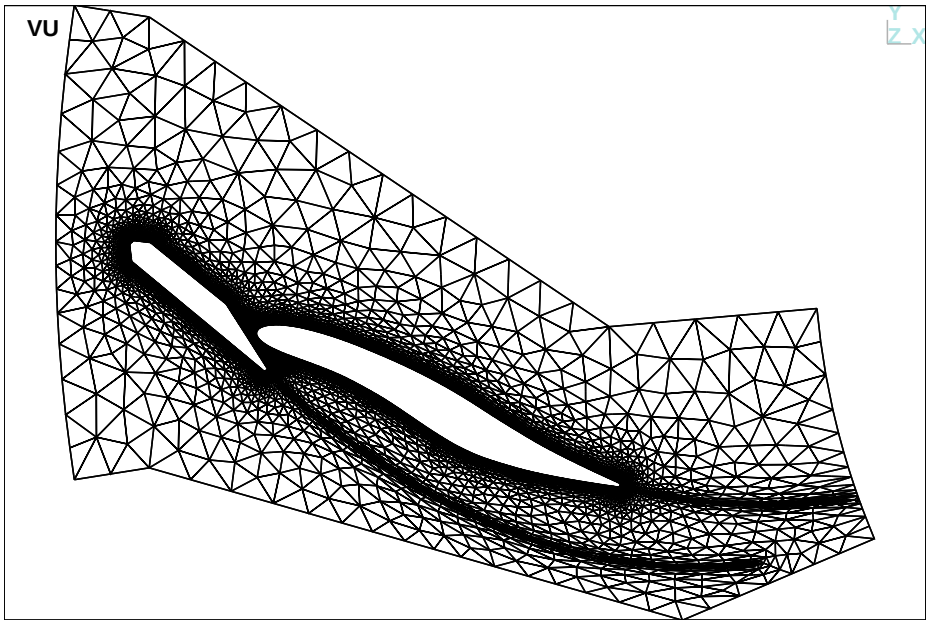


Figure 5.7: Step 3 in the SmartMesh script execution.

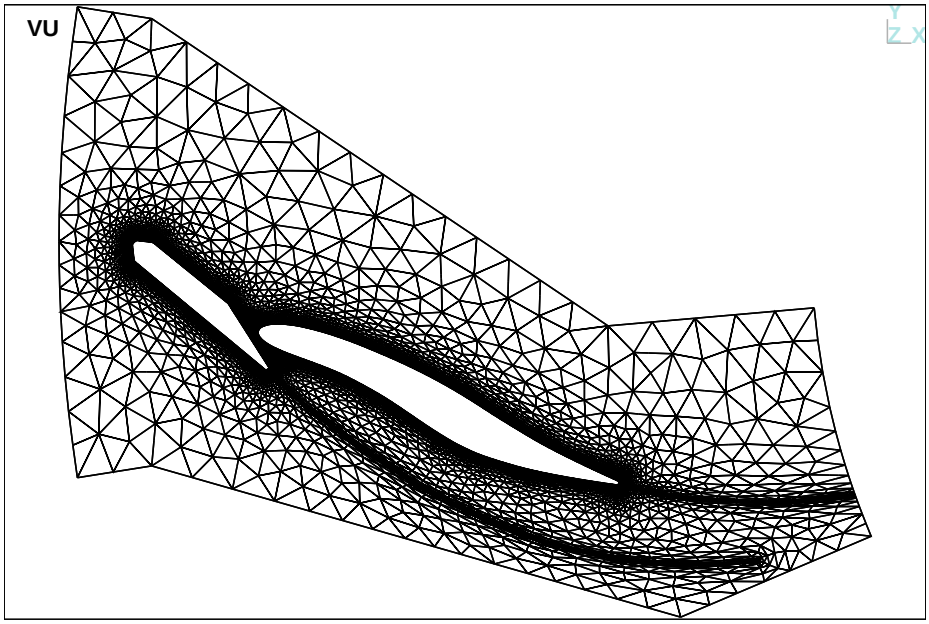


Figure 5.8: Step 4 in the SmartMesh script execution.

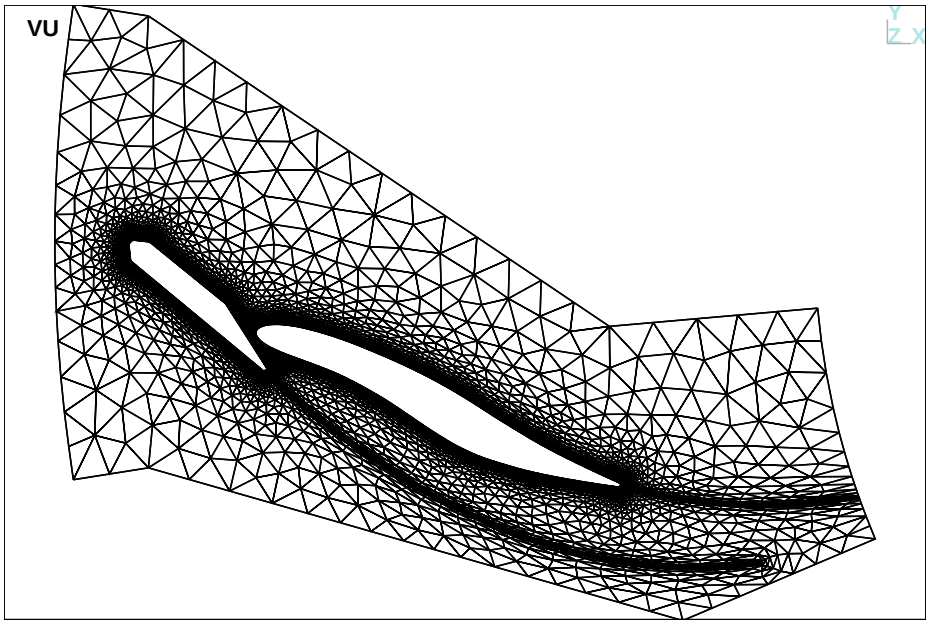


Figure 5.9: Step 5 in the SmartMesh script execution (final mesh).

6 Testing and results

> Science is facts; just as houses are made of stones, so is science made of facts; but a pile of stones is not a house and a collection of facts is not necessarily science.
>
> Henri Poincaré

This chapter covers the test cases used to validate that SmartMesh allows the user to control enough unstructured triangular mesh generation such that he can solve problems in different application domains. The application domain used is CFD. Three problems were tested: the U-duct, airplane wings and hydraulic turbines. Each problem has its own requirements that must be met so that the user can compute an accurate solution.

6.1 U-duct

The U-duct is a tube in which water must pass. The test case used in this section is from [34]. The geometry of this test case is shown in Fig. 6.1.

It is worth noting that this geometric domain has no holes inside the domain creating obstacles for fluid flow. Rather, the external boundaries are the obstacles.

Figure 6.2 shows a quadrilateral mesh that was generated manually in [34] as well as a triangular mesh created by SmartMesh with similar properties using a SmartMesh configuration file. Here we see that there is a need for additional precision along the outer boundaries and even more precision where external boundaries curve.

Figure 6.2 shows that by using SmartMesh components a user can re-create a

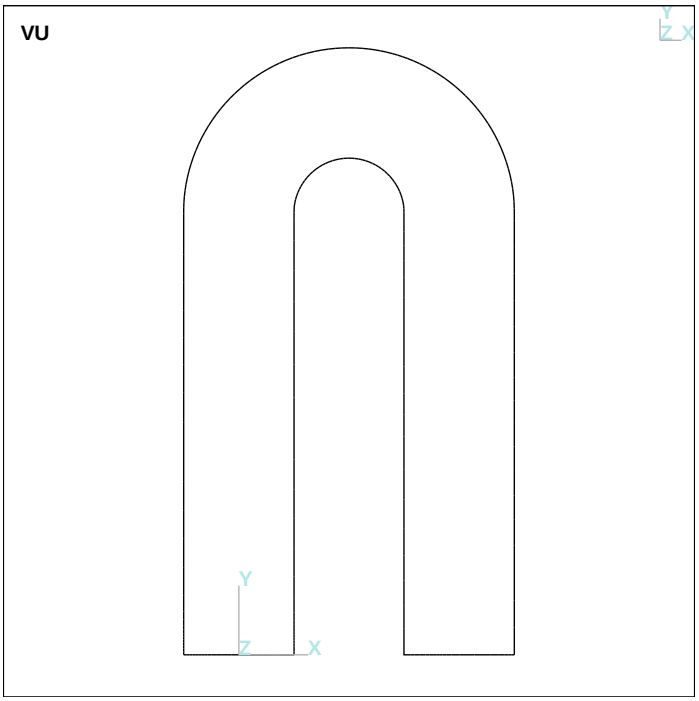


Figure 6.1: Geometry of U-duct problem.

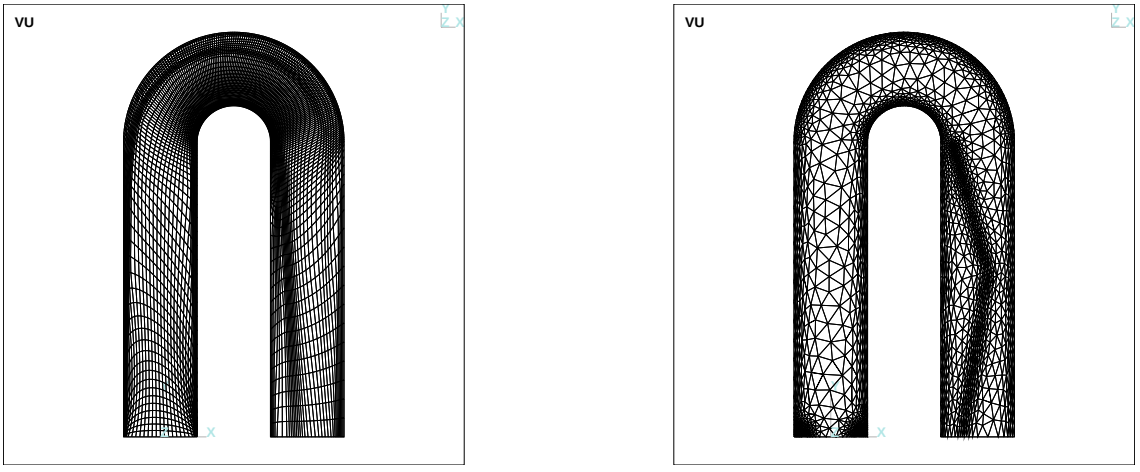


Figure 6.2: Different meshes for the same U-duct problem.

mesh that is similar to a mesh created by a different software.

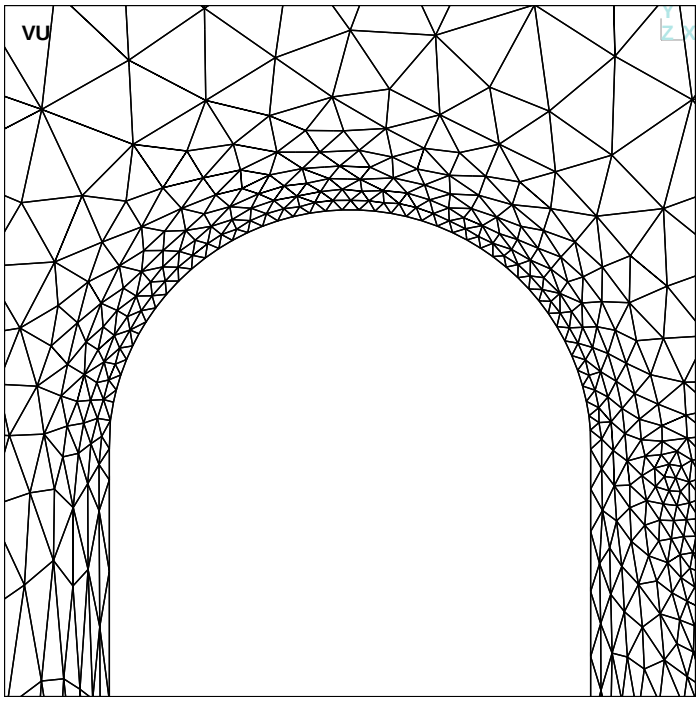


Figure 6.3: Additional precision where curvature is strong (SmartMesh output).

In order to achieve this mesh outputted by SmartMesh, the `DISTANCE_TO_POINT` algorithm was used on the two corners of the left input of the U-duct. Then, the `CURVATURE_OF_BORDER` algorithm was used on the curve that crosses the top portion of the domain, beginning at the outer corner of the left side of the U-duct and ending at the outer corner on the opposite size. Another `CURVATURE_OF_BORDER` algorithm was used on the curve that crosses the bottom part of the domain. Finally, a `DISTANCE_TO_IMAGINARY_POLYLINE` algorithm was used on the right side of the U-duct to simulate the wake created by particles passing through that region.

6.2 Airplane wings

External flow around wings is characterized by thin boundary layers, shocks in transonic and supersonic regime, and a wake possibly with von Karman alleys. All these features are localized compared to the size of the computational domain. All these

features are also strongly directional. The brute force approach, which consists of having a uniform and dense mesh over all the computational domain, is too costly. Therefore tools must be developed to drive the mesh generator to concentrate and align the mesh where needed. For these reasons, computational fluid dynamics are historically one of the leading sources of research in mesh generation.

In this section, two mesh generation approaches will be compared on many classic test cases found in the literature. The first approach is to use the a posteriori mesh adaptation. This is done first by computing a solution on the mesh, then by computing an error estimate based on the second derivatives of the computed solution, finally by computing an adapted mesh that equidistributes the error estimate. These three steps are performed many times by using a loop. The CFD solver used is NSC2$k\varepsilon$ by [29]. The mesh adapter used is OO$\mathcal{RT}$. See [1] for a more detailed description of this approach.

The second approach is to use SmartMesh. This is a priori man driven mesh adaptation, unlike the first approach, which is an a posteriori automatic error estimator driven mesh adaptation. The same solver NSC2$k\varepsilon$ will be used. The solution obtained with the first approach is used to help the user localizing flow features. The goal is to show that SmartMesh is flexible enough that a user can automatically reproduce by hand the meshes produced by the first approach and to show that these man made meshes are good enough to compute about the same solution with the same solver.

6.2.1 AGARD test cases on a NACA0012 airfoil

The AGARD01-05 test cases of the AGARD Working Group 07 ([22, 30]) are presented to validate SmartMesh. They consist of inviscid, transonic and supersonic external flows over a NACA 0012 airfoil. The free-stream conditions are summarized in Table 6.1 and the flow fields include shocks, either attached or detached, which are difficult to capture, particularly on coarse grids.

Table 6.1: Free-stream Mach number (M_∞) and angle of attack (AoA) for the AGARD01–05 test cases.

Test case	M_∞	AoA
AGARD01	0.80	1.25°
AGARD02	0.85	1.00°
AGARD03	0.95	0.00°
AGARD04	1.20	0.00°
AGARD05	1.20	7.00°

6.2.2 AGARD01

Figure 6.4 displays a generic Delaunay mesh generated by the software Tri△ngle from [33]. This mesh has 3691 vertices and 6730 triangles. Sure, a solution can be computed on this mesh, but this is clearly not suitable to compute accurate shocks. Figure 6.4 also displays the solution computed with NSC2$k\varepsilon$. The contour plots of this subsection will show contours of the Mach number with steps of 0.05. Notice that the upper shock is present but not well captured and also notice that the lower shock is missing.

Even if the mesh in Fig. 6.4 is not suitable for this flow and even if the solution is not accurate, they can be used to derive a posteriori error estimator based on second derivatives of the solution and mesh adaptation can be used to equidistribute this error. This will lead to a better mesh and will allow to compute a better solution. This is done again and again. Figure 6.5 shows the result of this process after 10 iterations of the mesh adaptation loop. This mesh has 4112 vertices and 8008 triangles. Note how smooth are the contours, how accurate and fine is the shock, and also note the presence of the weak lower shock.

SmartMesh is used to mimic the mesh of Fig. 6.5 obtained by a posteriori mesh adaptation. The mesh created with SmartMesh is shown in Fig. 6.6. This mesh was

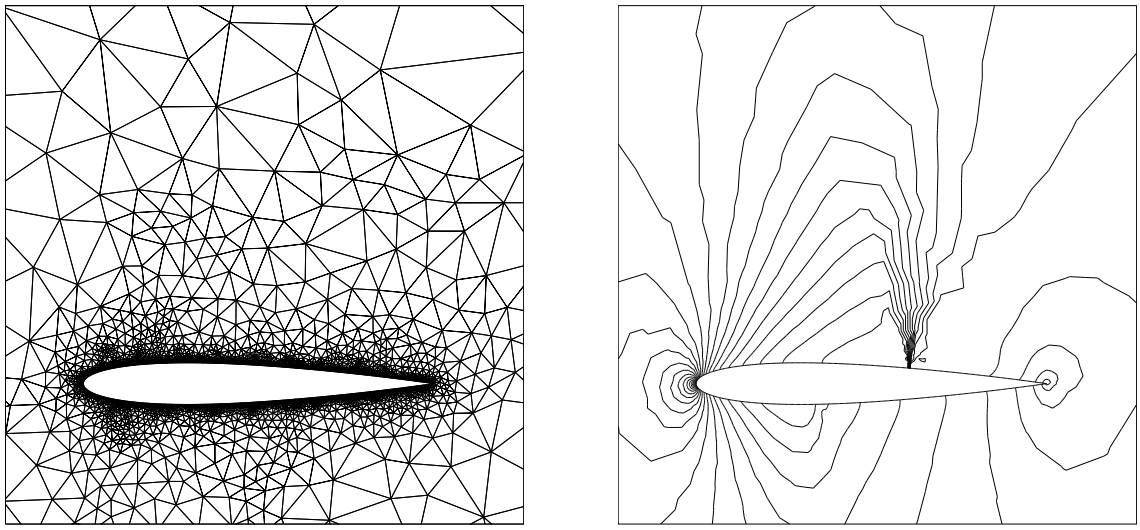


Figure 6.4: Test case AGARD01. Generic Delaunay mesh generated by Tri△ngle and Mach contours of the solution computed with NSC2$k\varepsilon$.

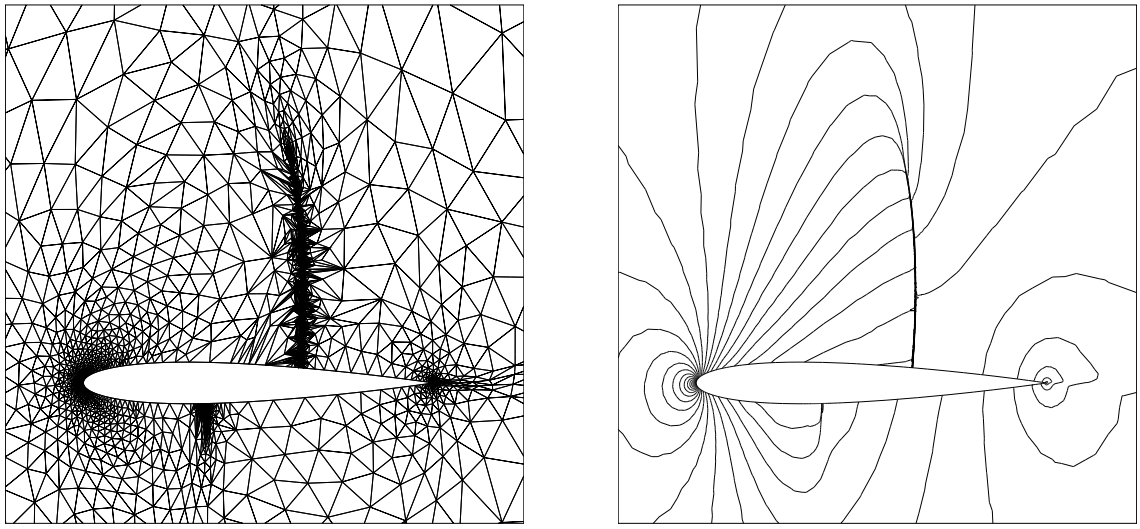


Figure 6.5: Test case AGARD01. Adapted mesh generated by OO$\mathcal{RT}$ and Mach contours of the solution computed with NSC2$k\varepsilon$.

simply created by using the algorithm DISTANCE_TO_POINT on the leading edge and the trailing edge, and with the algorithm DISTANCE_TO_IMAGINARY_POLYLINE with two polylines, one for each shock. The position of the polylines were roughly extracted from the previous mesh. This mesh has only 2465 vertices and 4727 triangles, which is significantly less than the two previous ones. Essentially, there are less vertices in the shocks because parameters used in the algorithm DISTANCE_TO_IMAGINARY_POLYLINE were set to create anisotropic (stretched) triangles. The mesh obtained with a posteriori mesh adaptation has quite small isotropic triangles in the shocks.

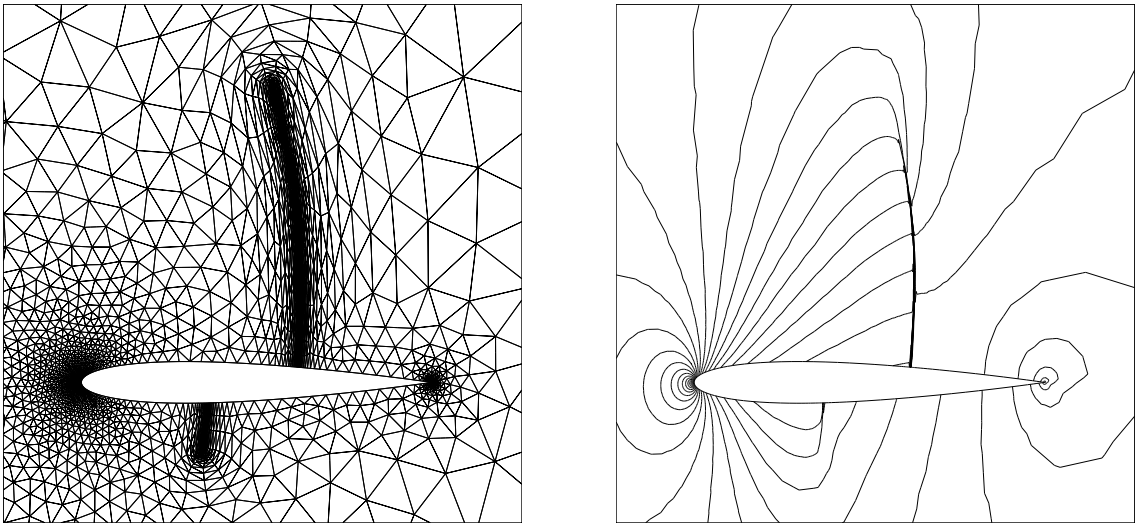


Figure 6.6: Test case AGARD01. Mesh generated by SmartMesh and Mach contours of the solution computed with NSC2$k\varepsilon$.

The corresponding solution computed with NSC2$k\varepsilon$ is also shown in Fig. 6.6. This solution computed is very similar to the one computed with anisotropic mesh adaptation, especially considering that there are 40% less vertices. A more significant comparison, for an airplane engineer, is the pressure coefficient C_p on the wall of the airfoil. This is given in Fig. 6.7. Note that the two shocks obtained with the initial mesh are inaccurate compared to those obtained with OO$\mathcal{RT}$ and SmartMesh. The upper shock is strong and both OO$\mathcal{RT}$ and SmartMesh give the same location. However, the shock with SmartMesh has more oscillations. For the lower weak

shock, the three meshes produce three different locations.

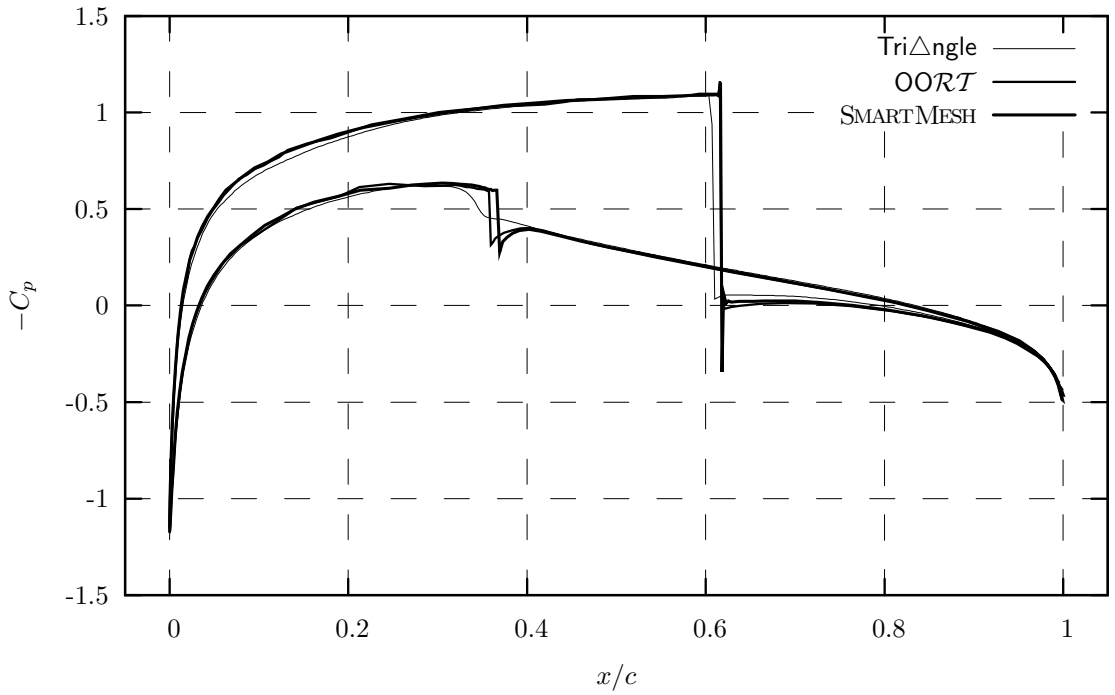


Figure 6.7: Test case AGARD01. Pressure coefficient C_p on the wall of the airfoil for the three different meshes.

6.2.3 AGARD02

Using the same techniques as for the AGARD01 test case of the previous subsection, the test case AGARD02 was computed. The results are shown in Fig. 6.8. The mesh with the a posteriori mesh adaptation using OO$\mathcal{RT}$ has 7503 vertices and 14754 triangles. The mesh hand made with SmartMesh has 3222 vertices and 6228 triangles, which is less than half of the total elements in the first mesh. Playing with the parameters, we can control the number of vertices and have 10 times more if needed. But, in this case, the parameters were the same for both meshes. The main difference is in the shocks. With error estimation and mesh adaptation with OO$\mathcal{RT}$, the triangles are mostly isotropic in the shocks, even if the shocks are directional features. This is caused by inviscid flows that require infinitely sharp shocks. With

viscous flows, shocks have a physical width and mesh adaptation produces anisotropic triangles aligned with the shocks.

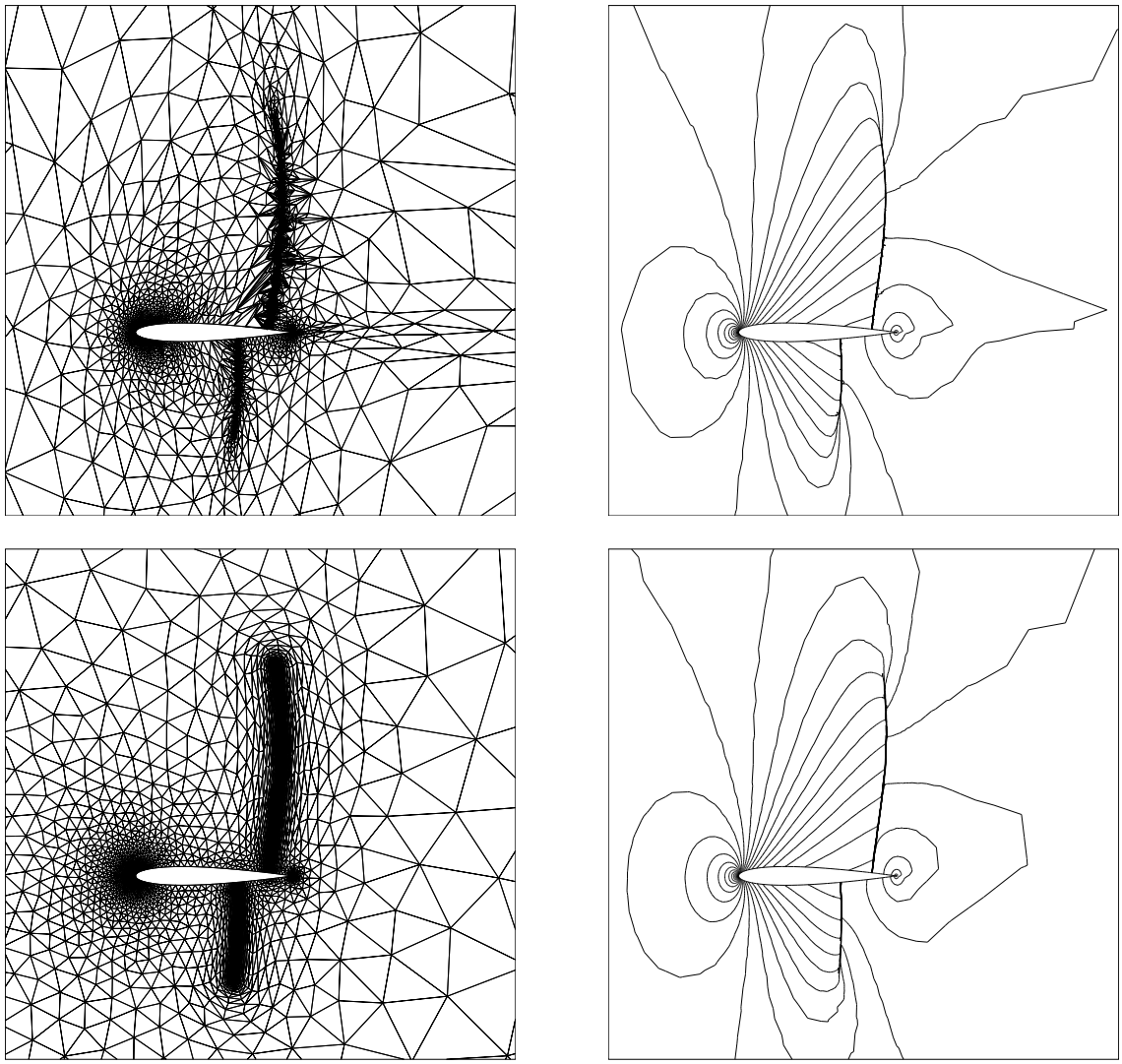


Figure 6.8: Test case AGARD02. At the top, mesh generated by $\mathsf{OO}\mathcal{RT}$ and Mach contours of the solution computed with $\mathsf{NSC2}k\varepsilon$. Same thing at the bottom, but mesh generated with SmartMesh.

On the contrary, meshes produced with SmartMesh has less vertices because triangles in the shocks are stretched and aligned. The stretching and direction is controlled by hand with the `DISTANCE_TO_IMAGINARY_POLYLINE` algorithm where the parameters are set to impose the stretching of triangles. Nevertheless, the solutions

in Fig. 6.8 are very similar.

The pressure coefficient C_p on the airfoil wall is plotted if Fig. 6.9. Note small oscillations of the pressure coefficient C_p around $x/c = 0.1$ for the solution computed on the mesh produced by OO$\mathcal{RT}$. This mesh may not be fine or regular enough at this location. However, the pressure coefficient C_p before and after the shocks has more oscillations for the solution computed on the mesh produced by SmartMesh. Overall, both solutions are remarkably similar.

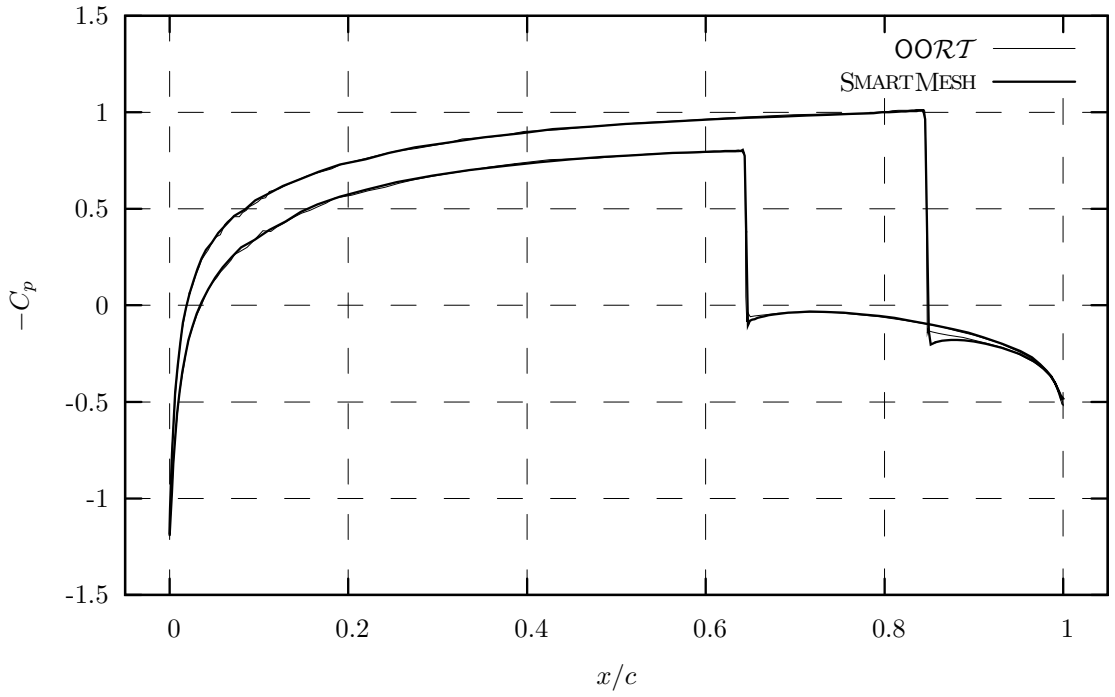


Figure 6.9: Test case AGARD02. Pressure coefficient C_p on the wall of the airfoil with meshes produced by OO$\mathcal{RT}$ and SmartMesh.

6.2.4 AGARD03

The test case AGARD03 has a Mach number of 0.95 and an angle of attack of zero degrees. So the solution should have a symmetry of reflexion through the x-axis. See Fig. 6.10 for a general overview of the meshes used and of the solutions obtained. The mesh obtained with mesh adaptation and OO$\mathcal{RT}$ has 104 530 vertices and 208 771

triangles. The mesh handcrafted with SmartMesh, using shock location obtained with the first approach, has 64 441 vertices and 128 432 triangles.

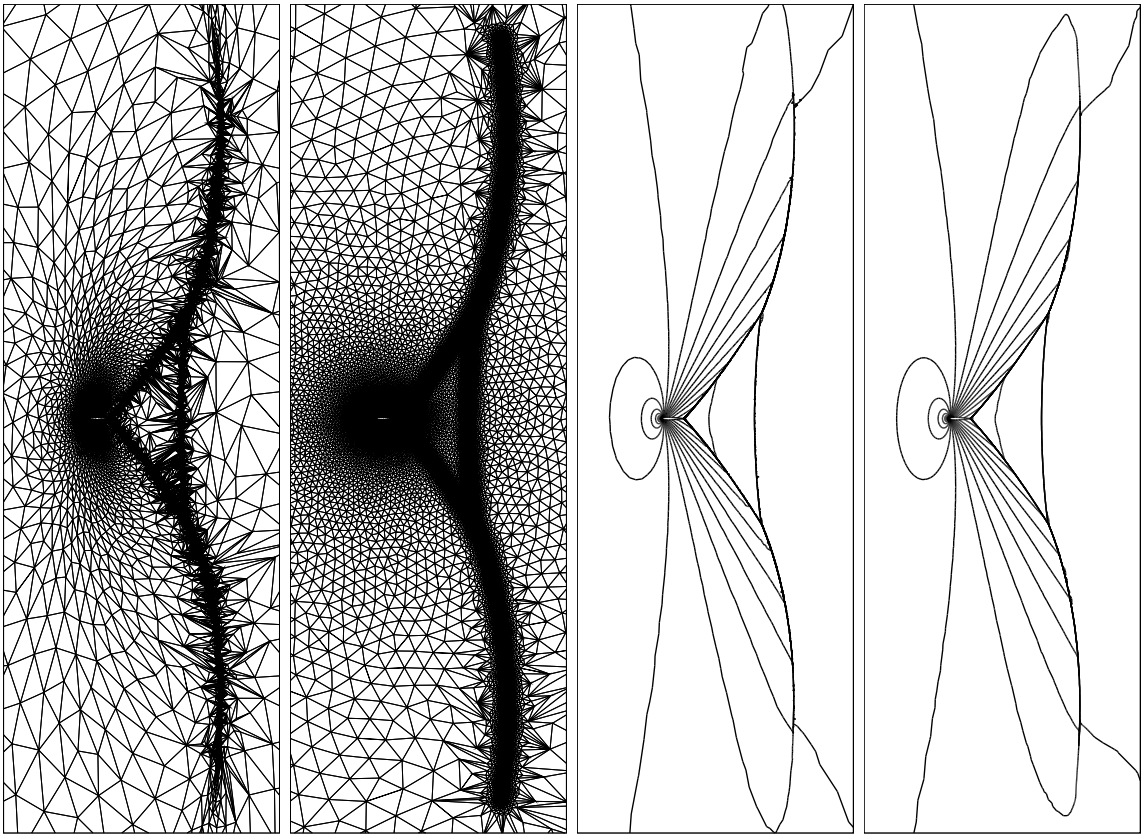


Figure 6.10: Test case AGARD03. From left to right. The mesh obtained with mesh adaptation with OO$\mathcal{RT}$ and the mesh handcrafted with SmartMesh. Then the solution computed with NSC2$k\varepsilon$ on the corresponding meshes.

The solution is characterized by two expansion shocks at the trailing edge. These two shocks are joined together by a weak vertical shock called the fish-tail shock. It is usually difficult to get this shock and its position along the x-axis is very sensitive. For example, the two expansion shocks at the trailing edge extends up to 35 chords. The chord for the NACA0012 is not 1.0, but 1.00893. So the domain must be large enough such that the boundaries and the boundary conditions do not affect expansion shocks, which in turn affect the fish-tail shock. Here, the domain used has a radius of 75.0. Another example is the accuracy of the solution at the leading edge and

along the profile. If the mesh is not fine enough, the solution will not be accurate enough and this will change the solution upstream to the two expansion shocks at the trailing edge which in turn affect their location which finally alters the location of the fish-tail shock.

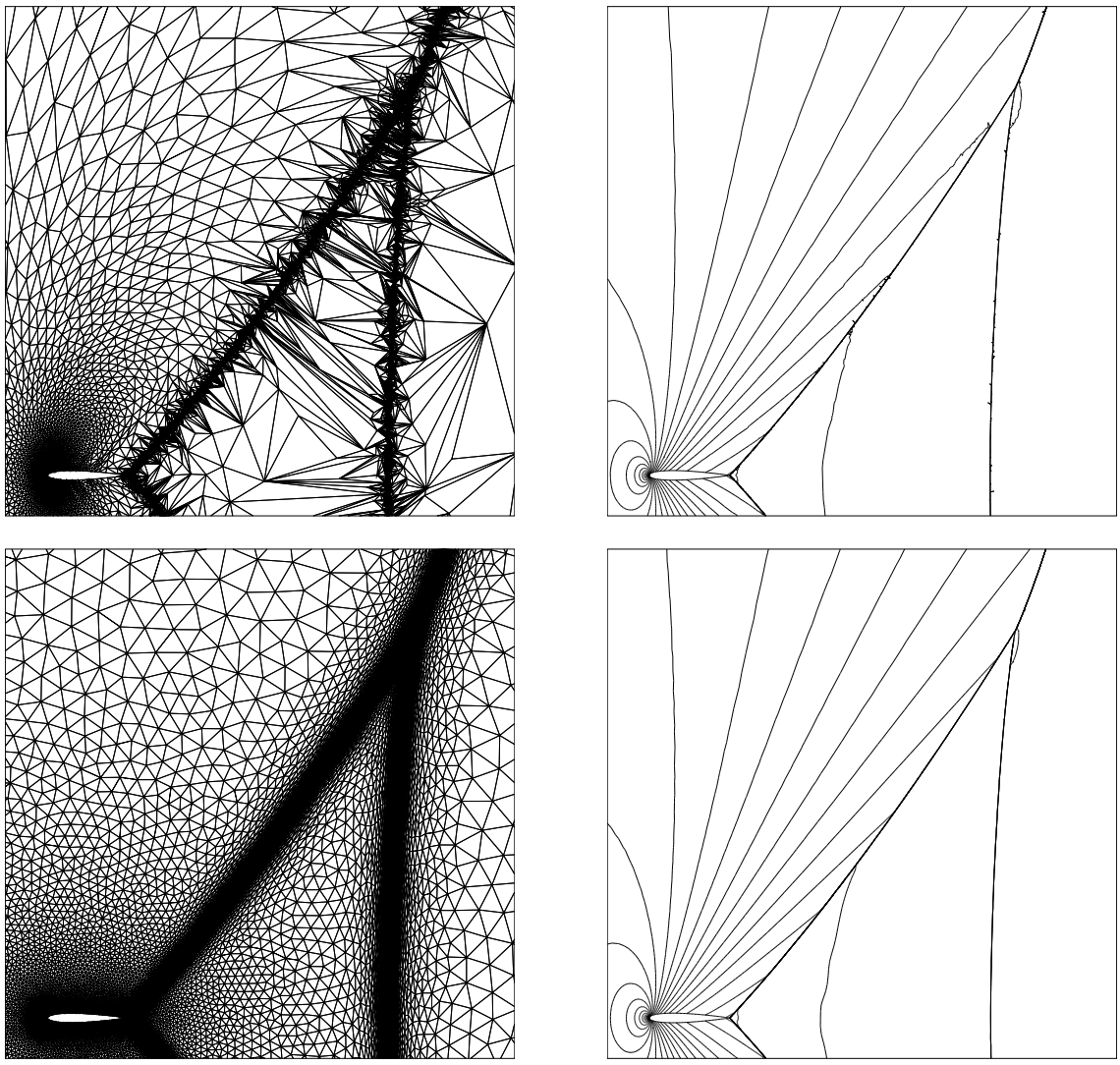


Figure 6.11: Test case AGARD03. At the top, mesh generated by OO$\mathcal{RT}$ and Mach contours of the solution computed with NSC2$k\varepsilon$. Same thing at the bottom, but mesh generated with SmartMesh.

Figure 6.11 shows a zoom of the expansion shock and the fish-tail shock with both approaches. The fish-tail shock position on the x-axis is 3.234 chords ± 0.001 with

the first approach (mesh adaptation and OO$\mathcal{RT}$) and 3.227 chords ± 0.001 with the second approach (SmartMesh).

A grid dependency study performed by [37] on 65×25, 129×49, 257×97 and 2049×765 C-grids, in a 100-chord domain, has shown that the fish-tail shock asymptotically moves to a location 3.35 chords behind the trailing edge on their finest mesh (2049×765 represents 1 567 485 vertices), which was approximately 15 and 25 times finer than the present meshes. Their conclusion is that with even further extrapolation, the position of the fish-tail shock, on an asymptotically infinitely fine grid, would be 3.32 chords behind the trailing edge.

6.2.5 AGARD04

The test case AGARD04 has a Mach number of 1.20 and an angle of attack of zero degree. Therefore the solution should have a symmetry of reflexion through the x-axis. See Fig. 6.12 for an upper-half overview of the meshes used and the solutions obtained. The mesh obtained with mesh adaptation and OO$\mathcal{RT}$ has 47 779 vertices and 95 268 triangles. The mesh handcrafted with SmartMesh, using shock location obtained by the first approach, has 26 325 vertices and 52 256 triangles.

6.2.6 AGARD05

The test case AGARD05 is very similar to AGARD04, both have a Mach number of 1.20, but now with an angle of attack of 7 degrees. This means that the solution is no longer symmetric through the x-axis. See Fig. 6.13 for a general overview of the meshes used and the solutions obtained. The mesh obtained with mesh adaptation and OO$\mathcal{RT}$ has 55 123 vertices and 109 990 triangles. The mesh handcrafted with SmartMesh, using shock location obtained by the first approach, has 38 427 vertices and 76 421 triangles.

Both test cases AGARD04 and 05 have a supersonic detach bow shock upstream to the wing and tail shocks attached to the trailing edge. The bottom tail shock

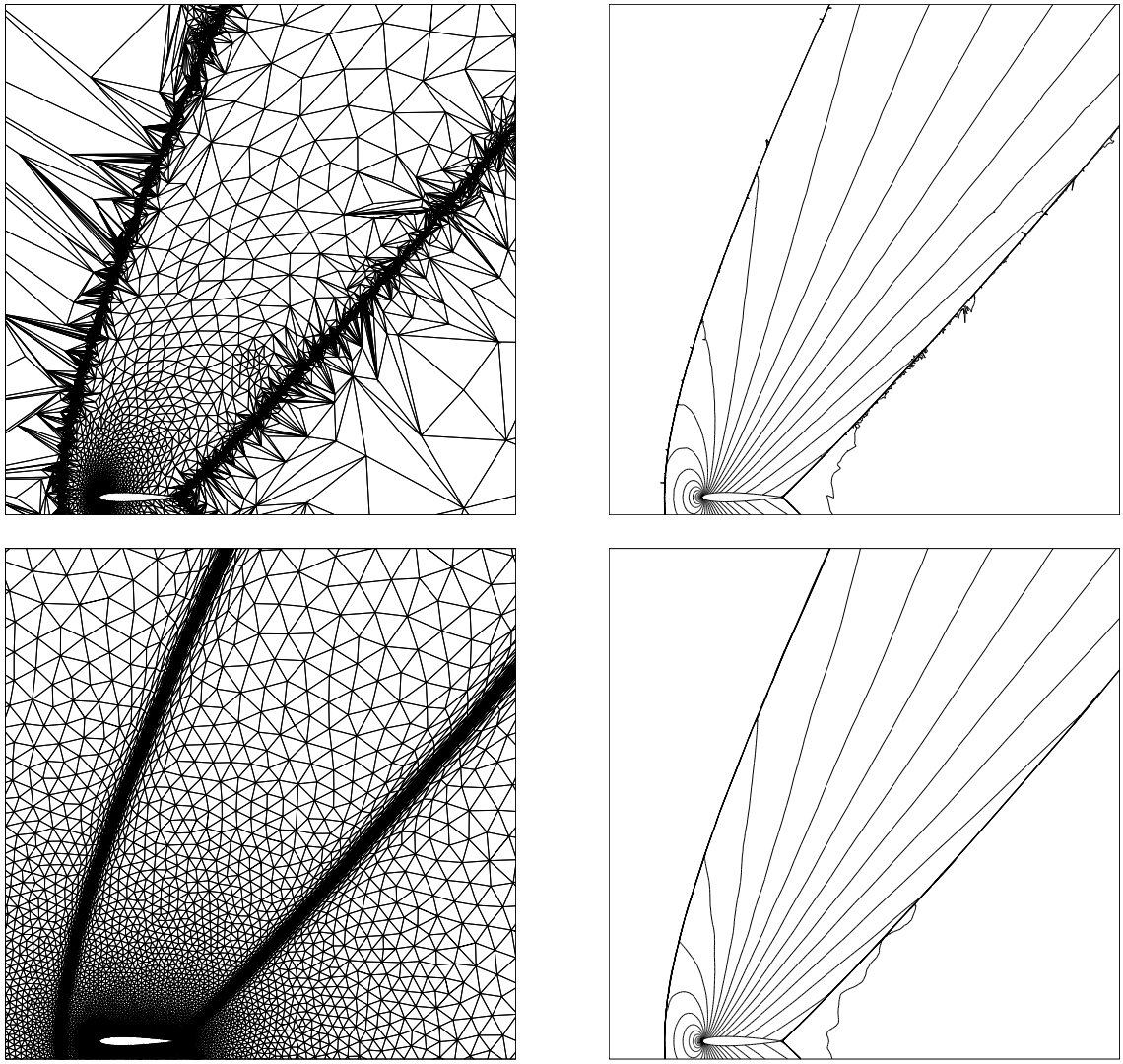


Figure 6.12: Test case AGARD04. From left to right. The mesh obtained with mesh adaptation with OO$\mathcal{RT}$ and the mesh handcrafted with SmartMesh. Then the solution computed with NSC2$k\varepsilon$ on the corresponding meshes.

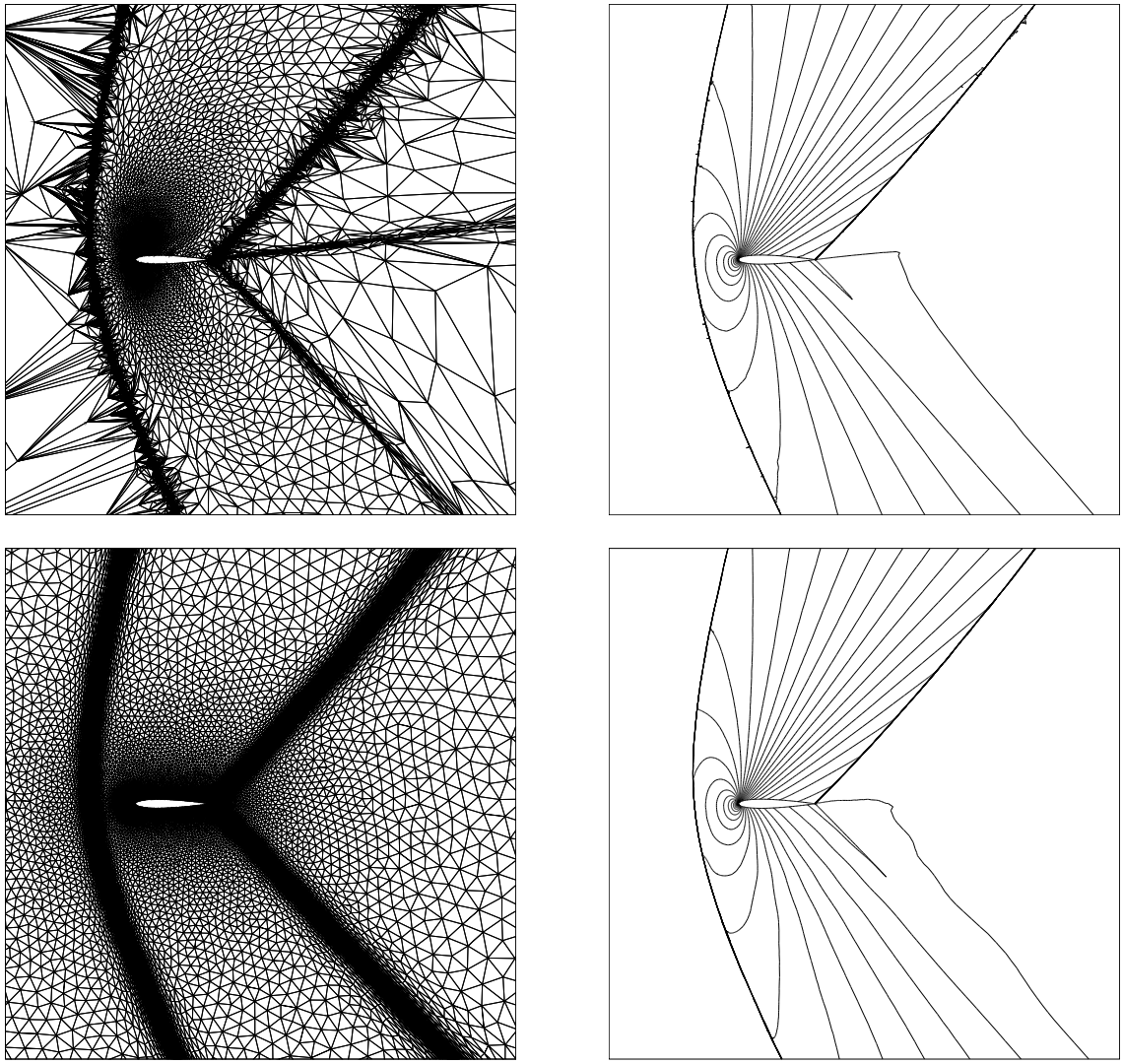


Figure 6.13: Test case AGARD05. From left to right. The mesh obtained with mesh adaptation with OO$\mathcal{RT}$ and the mesh handcrafted with SmartMesh. Then the solution computed with NSC2$k\varepsilon$ on the corresponding meshes.

on the AGARD05 test case is known to be difficult to capture. The reader may also notice the "wake" behind the trailing edge of the AGARD05 test case. This is an Euler inviscid flow simulation and there should not be any wake. This small wake, slightly visible in the solution and more visible looking at the adapted mesh, is probably numerical errors generated at the trailing edge.

These two test cases show the limits of SmartMesh. Without mesh adaptation, it is very difficult to forecast the location of the bow shock. An approach could be to use a very fine mesh everywhere to compute the bow shock location, but this would be very costly. With mesh adaptation, an initial mesh was used, a solution computed with NSC2$k\varepsilon$, an error estimator deduced and a new mesh generated with OO$\mathcal{RT}$, and again and again ten times. With this procedure, shock locations can be computed automatically at a low cost without any prior knowledge. Without this knowledge, SmartMesh is unusable. But with this knowledge, SmartMesh is used to redo by hand the meshes. Those meshes are even better because they are smoother, there are less vertices still having the same density. On those meshes, NSC2$k\varepsilon$ computes very similar solutions usually smoother.

6.2.7 The GAMM A7 test case on a NACA0012 airfoil

The GAMM-Workshop [8] problem A7 is a viscous flow at Mach 0.85, a zero degree angle of attack and a Reynolds number of 10 000, again over a NACA0012 airfoil. This flow is viscous, so it is characterized by a boundary layer and a wake. It is also transonic, so it has two attached supersonic shocks above and below the wing. Finally, it is an unsteady flow characterized by a von Karman alley in the wake. See Fig. 6.14 for additional details.

Back in 1987, none of the contributors of this GAMM-Workshop were able to compute a von Karman alley. At that time, this problem was too costly and the numerical schemes used to solve the Navier-Stokes equations were too viscous. This is now quite an easy problem to solve on a simple laptop due to improved processing

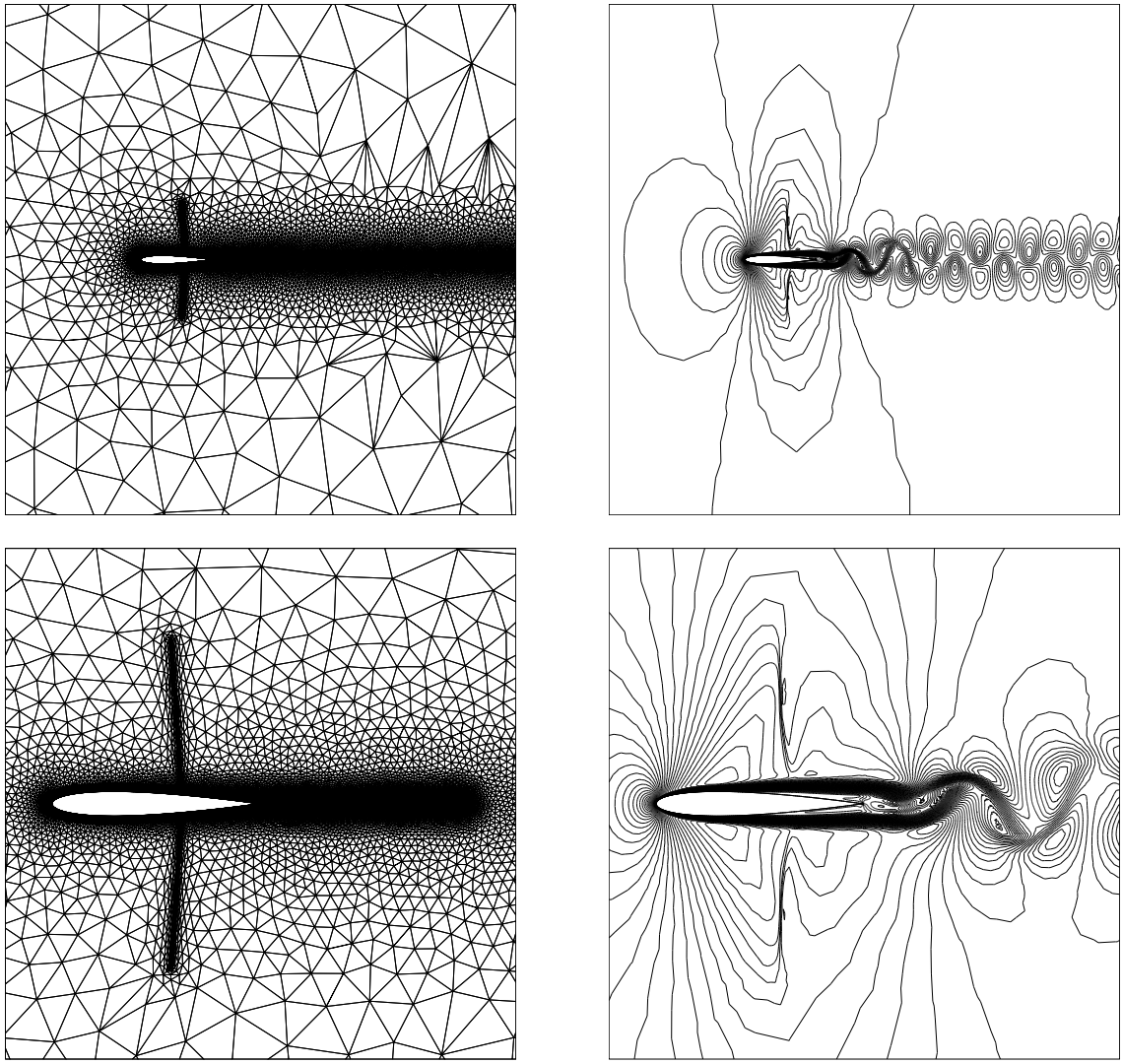


Figure 6.14: Test case GAMM A7. At the top, general view of the handcrafted mesh obtained with SmartMesh and the corresponding unsteady solution. Close view around the wing at the bottom.

power. In the literature, this problem is now usually done with a Reynolds number of 5000 instead. This is more viscous and less prone to be unsteady, so in some way more difficult. The result shown in Fig. 6.14 is done with a Reynolds number of 5000. Also, in this case only, isocontours of the Mach number are at every 0.025 instead of 0.05 as in all other results on the NACA0012. This mesh has 17 740 vertices and 35 144 triangles.

For all the AGARD test cases, flows are steady, and meshes and solutions were presented with both approaches, namely automatic mesh adaptation with OO$\mathcal{RT}$ and handmade mesh generation with SmartMesh. But in this GAMM A7 test case, flow is unsteady and presents difficulties for automatic mesh adaptation. The reason for this requires some explanations. Suppose that the unsteady solution was computed on a given mesh up to the time t_1. Then an error estimator is used to approximate the error on the solution at time t_1. From this error estimator, a new mesh is produced with OO$\mathcal{RT}$ to minimize this error at time t_1 and the solution is interpolated from the old mesh to the new mesh. Then NSC2$k\varepsilon$ is used again to advance simulation time from time t_1 to time t_2. And everything is done again, i.e., error estimator, mesh adaptation, solution interpolation. And so on for time t_3, t_4, ... Clearly this approach has two problems:

- There are solution interpolations between meshes. This induces diffusion, smoothing and mass loss.

- The mesh adaptation process as described above is an a posteriori mesh adaptation. The mesh is adapted to the solution computed before, not to the future solution to be computed. For unsteady phenomenons that change in time and move inside the computational domain, a posteriori mesh adaptation cannot build a fine mesh where the phenomenon will be in future time.

However, in this case, the unsteady phenomenons are swirls that move from left to right in the wake of the airfoil, which means the unsteady phenomenons are limited

to a horizontal band downstream the airfoil. With SmartMesh, a large wake was created, with standard additional features as fine and aligned mesh around the airfoil to catch the boundary layer, a fine mesh at leading and trailing edge and a fine mesh around supersonic shocks. This is the mesh shown in Fig. 6.14 and the solution clearly shows the von Karman alley, the boundary layer and the transonic shocks.

6.3 Hydraulic turbine

The need for mesh adaptation in the hydraulic turbine design problem is described in [14, 21]. The goal is to create a mesh that can accurately test whether or not a particular distributor design performs well in a CFD simulation. Of particular interest is the fluid flow close to the domain obstacles which are distributor blades and also the fluid flow in the wake of these blades.

This section will cover some simple distributor test cases to show that if the topological part of the geometric model remains the same but the geometric part changes, the same SmartMesh configuration parameters can be used for each design. Also, a series of industrial test cases done using SmartMesh and H2OMESH will be covered. Each test case covered in this section has two blades: the stay vane which is situated at the front closest to the inlet where the water arrives into the geometric domain and the guide vane which sits behind the stay vane.

6.3.1 Simple distributor test cases

Enginners usually want to optimize hydraulic turbine design. This is done in a loop where a turbine shape is tested. This design is the geometric model which can be used as input for the SmartMesh script in order to create an appropriate mesh. The output mesh and geometric model can be inputted into a program that performs a CFD simulation and evaluates the efficiency of the turbine design. Using this approach, it is possible to evaluate many shapes of distributor blades.

Recall that SmartMesh is independent of the geometric part of the geometric model. When proposing a new turbine design, the relation between topological entities remains the same while the curve definitions and positions change. In essence, the topological part of the geometric model remains the same while the geometric part changes. Since the topology is static, it is possible to run this hydraulic turbine optimization loop while always using the same SmartMesh configuration file.

For each of the following simple distributor test cases, the mesh is created by first applying the `DISTANCE_TO_POINT` algorithm on two topological vertices located at the trailing edge of the guide vane to the right side. There are an additional five topological vertices on the stay vane on the left side where the `DISTANCE_TO_POINT` algorithm is executed. The `DISTANCE_TO_BORDER` algorithm is applied to the stay vane while the `CURVATURE_OF_BORDER` algorithm is applied to the guide vane. Finally, the `DISTANCE_TO_IMAGINARY_POLYLINE` algorithm is applied to simulate the wake to the right of both blades.

6.3.1.1 Distributor 1

The first distributor test case has a very narrow guide vane. The goal is to show the effects of the curvature of border algorithm at the leading edge of the guide vane where the curvature will be very strong. Using the static configuration parameters defined in this section, SmartMesh was able to produce an output mesh. Figure 6.15 shows the domain geometry for this case and the corresponding mesh generated by SmartMesh.

Of particular interest in this problem is the leading edge of the guide vane where the narrow blade causes a strong curvature. Figure 6.16 shows the effect of this algorithm on the mesh.

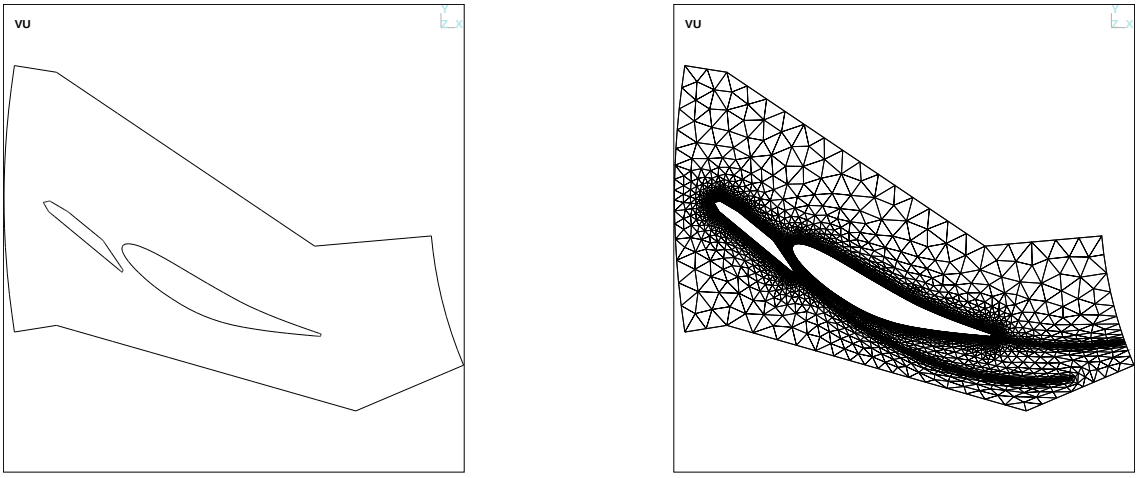


Figure 6.15: Geometric domain and mesh for the first hydraulic turbine distributor.

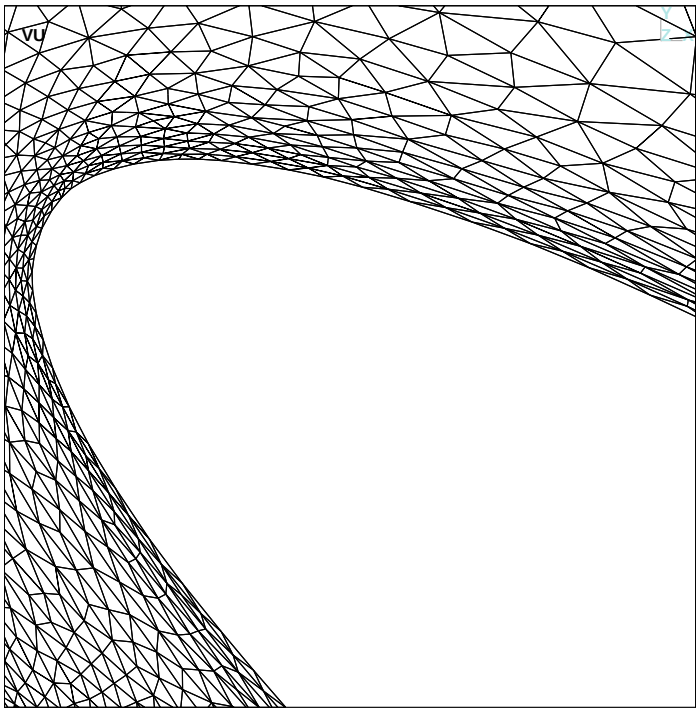


Figure 6.16: Effect of curvature of border algorithm on the leading edge of the guide vane.

6.3.1.2 Distributor 2

The second distributor test case has a very thick guide vane and a larger gap between the two blades. The goal is to show the effects of the distance between two borders algorithm in a context featuring a larger distance between the stay vane and the guide vane. Using the static configuration parameters defined in this section, SMARTMESH was able to produce an output mesh. Figure 6.17 shows the domain geometry for this case and the corresponding mesh generated by SMARTMESH.

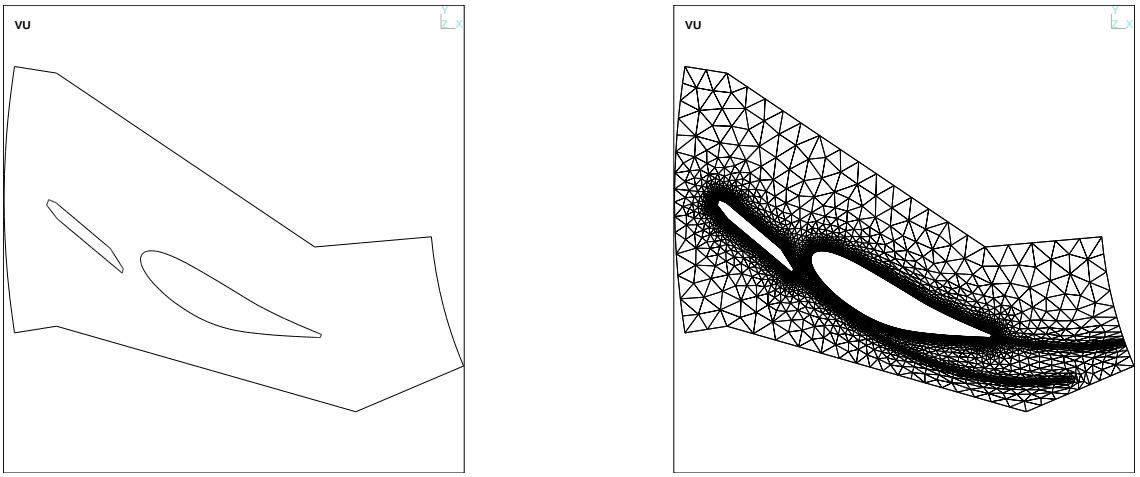


Figure 6.17: Geometric domain and mesh for the second hydraulic turbine distributor.

Of particular interest in this problem is the area between the stay vane and guide vane where the wide gap should lead to the distance between two borders algorithm has less effect on the mesh. Figure 6.18 shows the effect of this algorithm on the mesh.

6.3.1.3 Distributor 3

The third distributor test case has a thin S-shaped guide vane and a very small gap between the two blades. Contrary to the second distributor test case, the goal of the third test case is to show the effects of the distance between two borders algorithm in a context featuring a very small distance between the stay vane and the guide vane.

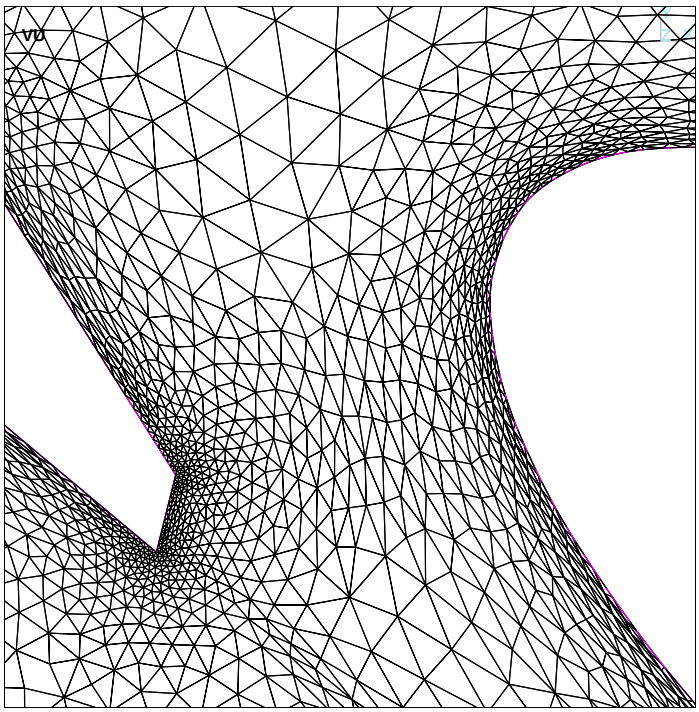


Figure 6.18: Effect of distance between two borders algorithm on the second distributor.

Recall that the same configuration parameters are being used. The expectation is that the algorithm will have a much more significant effect on the final mesh. Using the static configuration parameters defined in this section, SmartMesh was able to produce an output mesh. Figure 6.19 shows the domain geometry for this case and the corresponding mesh generated by SmartMesh.

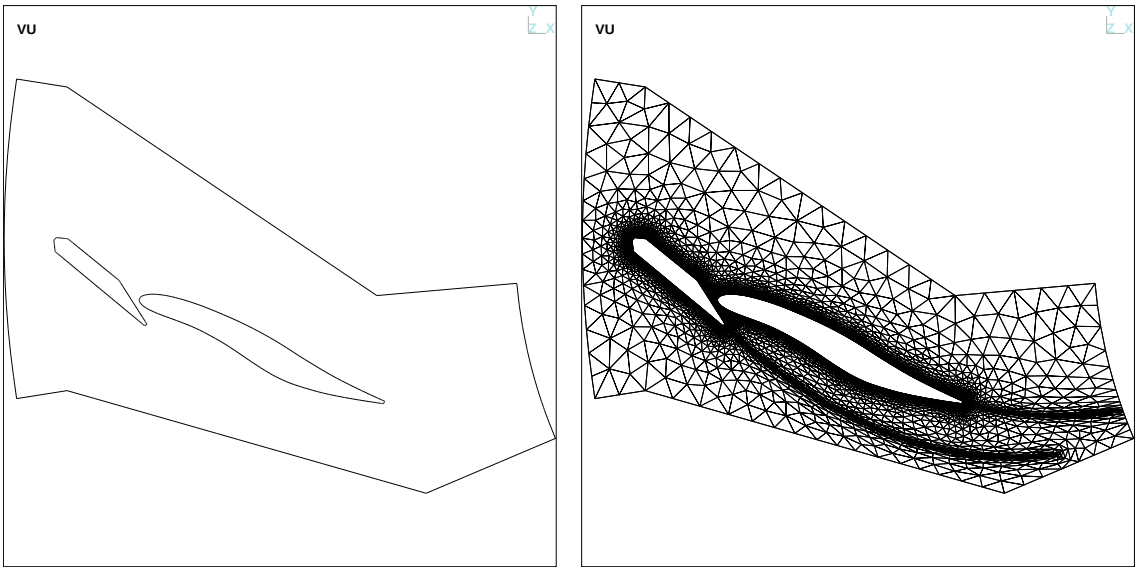


Figure 6.19: Geometric domain and mesh for the third hydraulic turbine distributor.

Of particular interest in this problem is the area between the stay vane and guide vane where the narrow gap should lead to the distance between two borders algorithm having a significant effect on the mesh. Figure 6.20 shows the effect of this algorithm on the mesh.

In conclusion, these three different distributors were in fact not much different. They all have the same topology, only the geometric description of the blade curves changes slightly. The purpose of this was to mimic a blade optimizer that tests different shapes. The three test cases had the same execution with the same parameters for SmartMesh and $OO\mathcal{RT}$. In all cases, the meshes produced are not the same, but have the same characteristics and quality. We believe, even if it was not tested, that the mesh produced can be used by a shape optimization software.

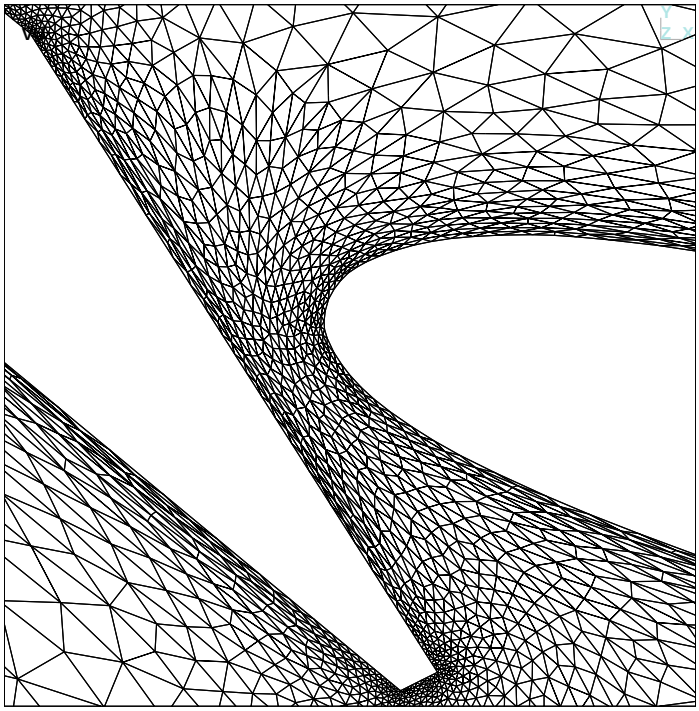


Figure 6.20: Effect of distance between two borders algorithm on third distributor.

6.3.2 Industrial testing

At École Polytechnique in Montréal, there is a program called H2OMESH that does the full process of generating the computational domain in question, creating an initial mesh, modifying the mesh intelligently and performing a CFD simulation. CFX is the CFD solver used to run simulations and perform calculations.

One area of interest in the industrial testing is the use of the `Layer` program to add a structured quadrilateral grid around the distributor vanes. The process of creating this additional structured grid is automated in H2OMESH.

Additionally, H2OMESH creates fictitious periodic boundaries between vanes of the distributors. The goal is to be able to do numerical simulations only on a tandem composed of two vanes, one stay vane and one guide vane. The segmented borders must adhere to periodicity so that the same mesh can be duplicated and placed side-by-side for the simulation. Fictitious periodic boundaries could be a piecewise linear curve or a higher order curve. Also, the mesh on these fictitious periodic boundaries

may, or may not, be periodic.

Prior to the testing performed in this section, it was not known if

- CFX is able to solve without a structured grid of quadrilaterals around the vanes;
- CFX is able to deal with non periodic meshes on periodic boundaries.

As these test cases will show, CFX is compatible in both cases.

Note that this test case in under a non-disclosure agreement with Andritz AG and its subsidiary in Pointe-Claire, in west suburb of Montréal. All pictures in this section were slightly skewed, scaled, twisted and so on.

Figure 6.21 shows the resulting mesh and solution obtained by H2OMESH. This test is generated by an expert in the field. The output mesh has 6 901 vertices and 10 334 combined triangles and quadrilaterals.

Figure 6.21: Mesh generated by H2OMESH with a structured layer and isocontours of the velocity field computed by CFX.

It is worth noting that H2OMESH does not provide a mechanism to create a fine mesh past the trailing edge of the vane for accurate wake simulation. This results in a loss of accuracy beyond the edges of the distributor blades where the water particles continue their course. SmartMesh results will improve upon this drawback.

In this section we will show the solution with a simple Delaunay mesh done with Tri△ngle, as well as the solutions associated to SmartMesh outputs which include meshes with and without periodic mesh on periodic boundaries and also with and without an added structured skin around the stay vane and guide vane.

6.3.2.1 SmartMesh – Simple Delaunay mesh

Any Delaunay mesh can be used to obtain a solution and perform calculations. The process of creating a simple Delaunay mesh is fast and easy. However, such a mesh is not satisfactory when it comes to running simulations and performing accurate calculations. Figure 6.22 shows a generic Delaunay mesh created by Tri△ngle along with the corresponding isocontours of velocity field which was output by the CFX solver. The output mesh has 2 887 vertices and 5 196 triangles.

When comparing to the isocontours shows in the H2OMESH output in Fig. 6.21 one can see that this result is not accurate. Therefore, SmartMesh must be used to intelligently adapt this mesh in an attempt to produce a solution with isocontours resembling the H2OMESH solution.

6.3.2.2 SmartMesh – Curved boundaries, added structured skin

This sub-section will present a mesh generated by SmartMesh where the periodic boundaries are curves and the `Layer` program is used to create a structured skin around the the stay vane and guide vane. The mesh along the periodic boundaries are not periodic, and as the periodic boundaries are curved, the periodic boundaries no longer match. Figure 6.23 shows velocity isocontours of the solution generated by CFX with this mesh as input. This mesh has 12 214 vertices and 20 710 combined

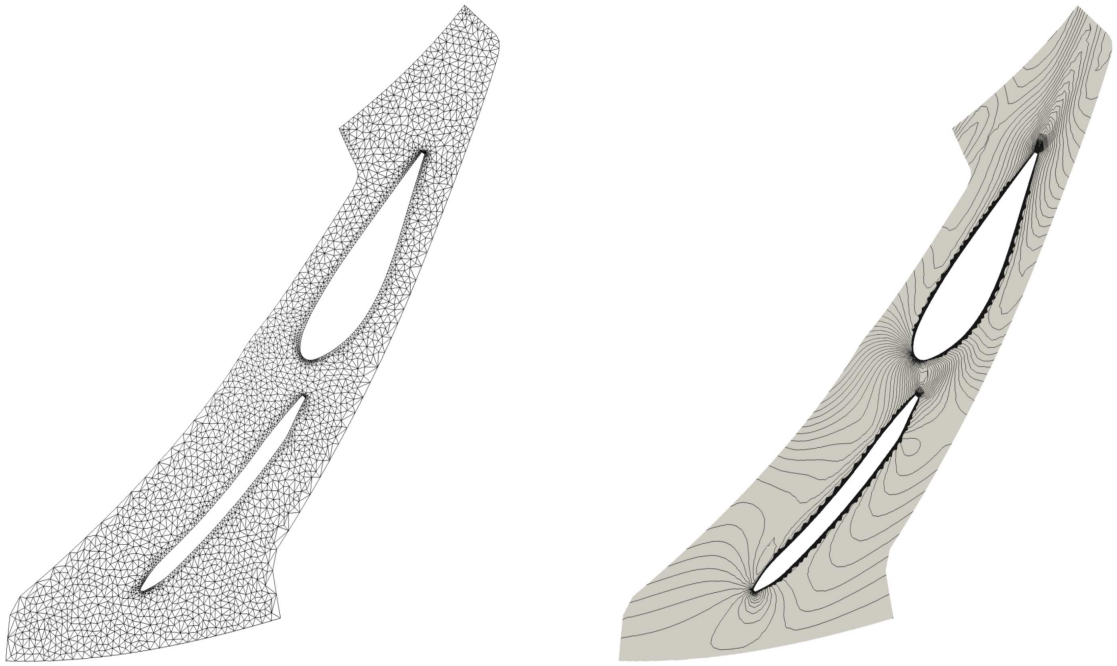


Figure 6.22: Generic Delaunay mesh generated by Tri△ngle and isocontours of the velocity field computed by CFX.

triangles and quadrilaterals. Since the mesh generated by SmartMesh does account for the wake while the mesh generated by H2OMESH does not, a mesh without the wake was created by SmartMesh in order to obtain a number of mesh elements that can directly be comparable to the result from H2OMESH.

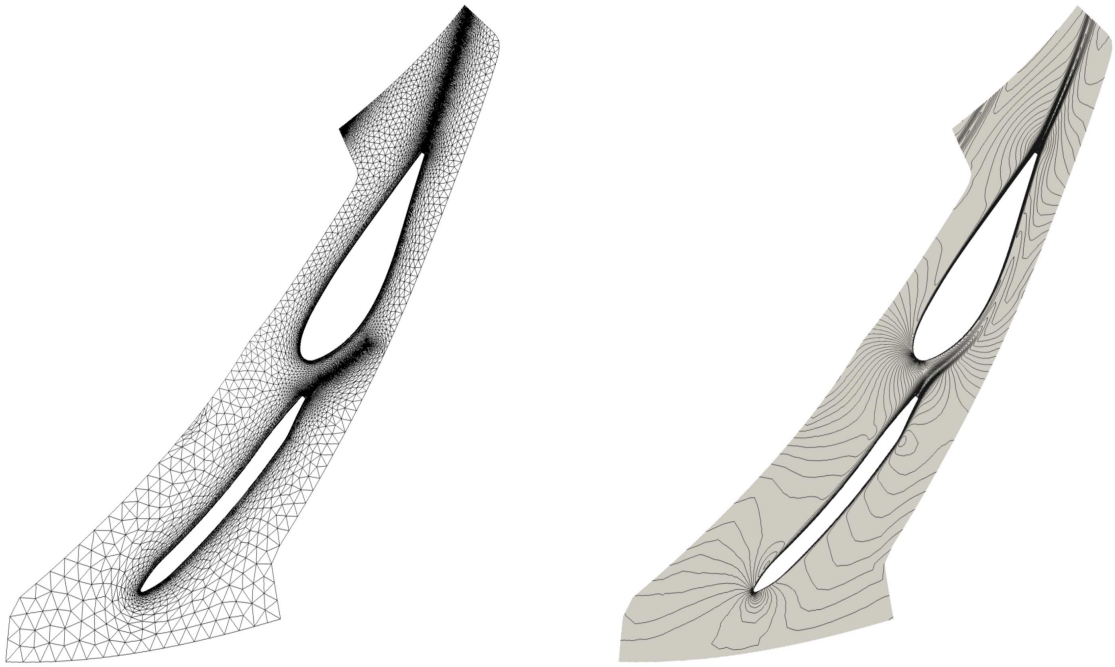


Figure 6.23: Mesh generated by SmartMesh with curved boundaries and added skin around blades (left) and isocontours of the velocity field computed by CFX (right).

This test case confirms that the CFX solver is compatible with input meshes that do not fit on curved periodic boundaries. However, one can see that there is some disturbance in the velocity field isocontours close to this mesh's boundary when comparing to the H2OMESH solution. It is clear that the lines in this solution do not follow the same pattern close to the boundaries. So while it is useful to know that CFX can handle such meshes, the solution loses some accuracy because the boundaries are not periodic.

Also worth noting is the wake of both blades in this solution's velocity field isocontours. When comparing with the H2OMESH solution, one can see that this solution

offers additional precision in this area. This is because SmartMesh supports the distance to imaginary polyline algorithm while H2OMESH does not. This may not be very important for the guide vane, but an accurate wake for the stay vane will change significantly the solution on the guide vane as the stay vane wake hits the guide vane.

6.3.2.3 SmartMesh – Curved boundaries, no added structured skin

This sub-section will present a mesh generated by SmartMesh where the periodic boundaries are curves and has no added structured skin. Figure 6.24 shows velocity isocontours of the solution generated by CFX with this mesh as input. The mesh has 9 113 vertices and 17 609 triangles. Since the mesh generated by SmartMesh does account for the wake while the mesh generated by H2OMESH does not, a mesh without the wake was created by SmartMesh in order to obtain a number of mesh elements that can directly be comparable to the result from H2OMESH.

This test case confirms that the CFX solver is compatible with input meshes that do not have an added structured skin around the stay vane and guide vane. However, one can see that there is some disturbance in the velocity field isocontours close to this mesh's stay vane and guide vane when comparing to the H2OMESH solution. It is clear that the lines in this solution do not follow the same pattern close to the two distributor blades. So while it is useful to know that CFX can handle such meshes, the solution loses some accuracy because there is no added structured skin. The same can be said for the isocontours at the external boundary of the mesh. It is clear that the lines in this solution do not follow the same pattern close to the boundaries.

Also worth noting is the wake of both blades in this solution's velocity field isocontours. When comparing with the H2OMESH solution, one can see that this solution offers additional precision in this area. This is because SmartMesh supports the distance to imaginary polyline algorithm while H2OMESH does not.

Note that this was our first and unique trial with that type of mesh. All test cases were done using the same parameters for CFX. Obviously, further research should be

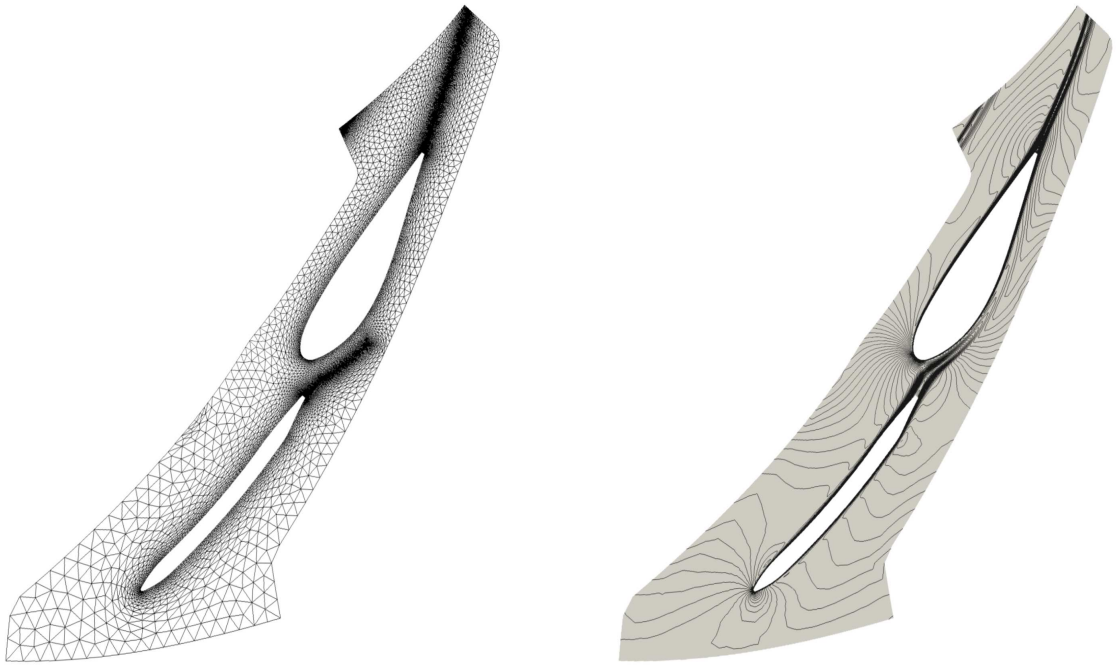


Figure 6.24: Mesh generated by SmartMesh with curved boundaries and no added skin around blades (left) and isocontours of the velocity field computed by CFX (right).

done to tweak CFX parameters and SmartMesh parameters to obtain better solutions with meshes containing no added structured skin.

6.3.2.4 SmartMesh – Periodic boundaries, added structured skin

This sub-section will present a mesh generated by SmartMesh where the periodic boundaries are not curves but rather polylines. Even if the mesh on periodic boundaries is not periodic, piecewise linear periodic boundaries will always perfectly match, provided that there is a mesh vertex between each segment of the periodic boundaries. In addition, this test case will have a structured skin around the distributor blades. Figure 6.25 shows velocity isocontours of the solution generated by CFX with this mesh as input. This mesh has 11 974 vertices and 20 313 combined triangles and quadrilaterals. Since the mesh generated by SmartMesh does account for the wake while the mesh generated by H2OMESH does not, a mesh without the wake was created by SmartMesh in order to obtain a number of mesh elements that can directly be comparable to the result from H2OMESH.

This test case should mimic the H2OMESH solution most closely because meshes outputted by H2OMESH also have periodic meshes on periodic boundaries as well as a structured skin around the stay vane and guide vane. When looking at both solutions along the outer boundary and close to the distributor blades we can see that the isocontours of the velocity field are almost the same.

Also worth noting is the wake of both blades in this solution's velocity field isocontours. When comparing with the H2OMESH solution, one can see that this solution offers additional precision in this area. This is because SmartMesh supports the distance to imaginary polyline algorithm while H2OMESH does not.

6.3.2.5 SmartMesh – Periodic boundaries, no added structured skin

This sub-section will present a mesh generated by SmartMesh where the periodic boundaries are piecewise linear and where there is no skin around the distributor

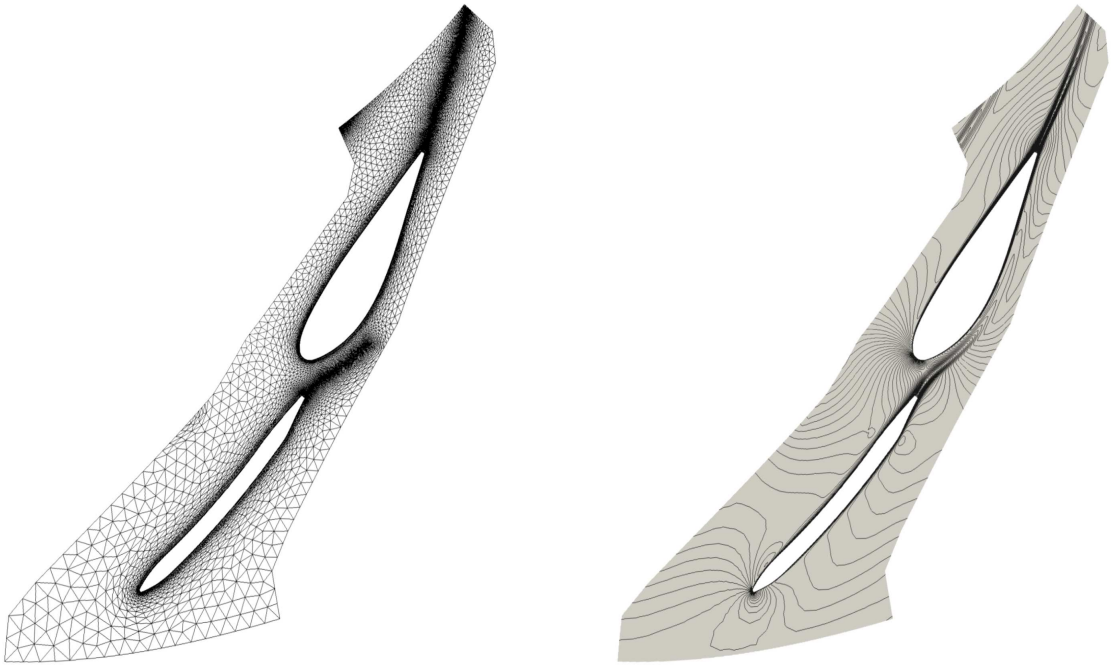


Figure 6.25: Mesh generated by SmartMesh with piecewise linear periodic boundaries and added skin around blades (left) and isocontours of the velocity field computed by CFX (right).

blades. Figure 6.26 shows velocity isocontours of the solution generated by CFX with this mesh as input. This mesh has 8 934 vertices and 17 282 triangles. Since the mesh generated by SmartMesh does account for the wake while the mesh generated by H2OMESH does not, a mesh without the wake was created by SmartMesh in order to obtain a number of mesh elements that can directly be comparable to the result from H2OMESH.

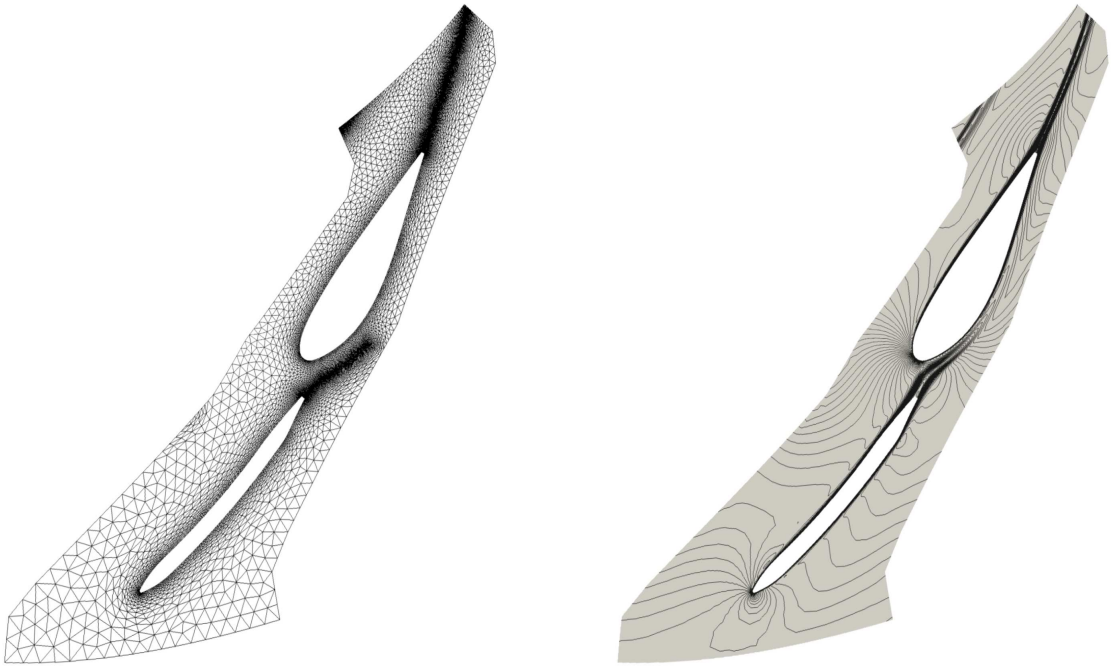


Figure 6.26: Mesh generated by SmartMesh with piecewise linear periodic boundaries and no added skin around blades (left) and isocontours of the velocity field computed by CFX (right).

One can see that there is some disturbance in the velocity field isocontours close to this mesh's stay vane and guide vane when comparing to the H2OMESH solution. It is clear that the lines in this solution do not follow the same pattern close to the two distributor blades. So while it is useful to know that CFX can handle such meshes, the solution loses some accuracy because there is no added structured skin.

6.3.2.6 Value of adapting the mesh

In this thesis it has been stated that intelligent mesh adaptation is essential to generate a mesh that is suitable for obtaining accurate results from a solver. Here we will investigate the isocontours of the solution for a simple Delaunay mesh, a mesh created by SmartMesh with no added structured skin and a mesh created by SmartMesh with an added structured skin from `Layer`. The images will zoom into the area between the stay vane and guide vane where there is a wake to be captured, a thin boundary layer at the leading edge of the guide vane and a thick boundary layer at the end of the guide vane.

Figure 6.27 shows a simple Delaunay mesh generated by Tri△ngle. This mesh is not sufficiently fine in the important regions and thus the corresponding solution has very rough isocontours. Since there is no intelligent mesh adaptation, the wake behind the stay vane is almost completely absent in the calculated solution, which means that this output is inaccurate. Furthermore, notice that he isocontours along the distributor blade boundaries are not smooth at all due to the accompanying mesh being too coarse in these regions.

Figure 6.28 shows a mesh created by SmartMesh with no added structured skin. With mesh generation driven by an intelligent user, the mesh is sufficiently fine in the wake and along the distributor blade boundaries. The accompanying solution correctly captures the wake as seen by the smooth isocontours in this region. However, along the boundaries, the solution's isocontours are still a little bit rough compared to the solution shown in Fig. 6.29. As explained, this is our first and unique trial. CFD and CFX experts should be able to tweak parameters to obtain better solutions.

Figure 6.29 shows a mesh created by SmartMesh with a structured skin added by the `Layer` program around the distributor blades. The structured skin allows the CFX to capture boundary layers much more accurately along the boundaries. This is especially obvious at the rear of the stay vane where the wake begins.

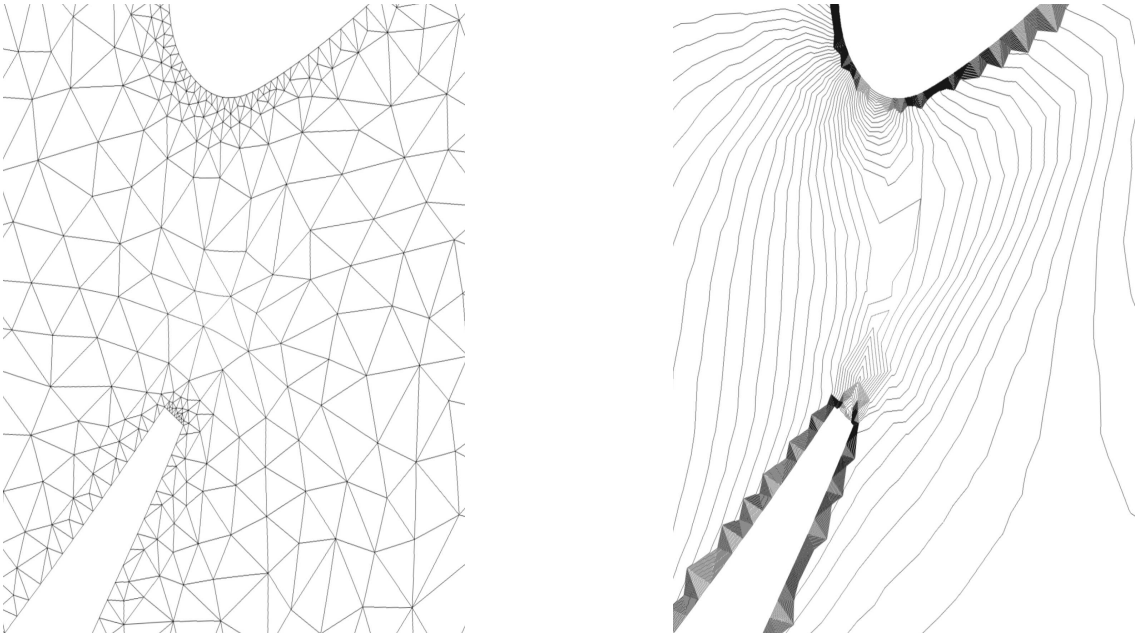


Figure 6.27: Simple Delaunay mesh and isocontours between distributor blades.

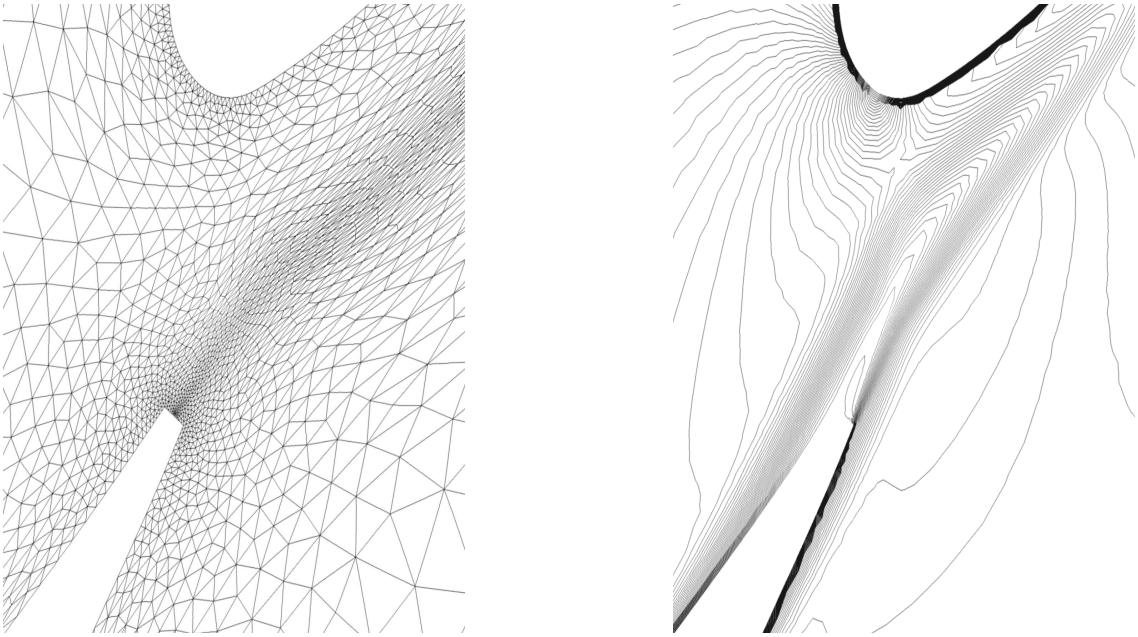


Figure 6.28: Adapted mesh with no structured skin and isocontours between distributor blades.

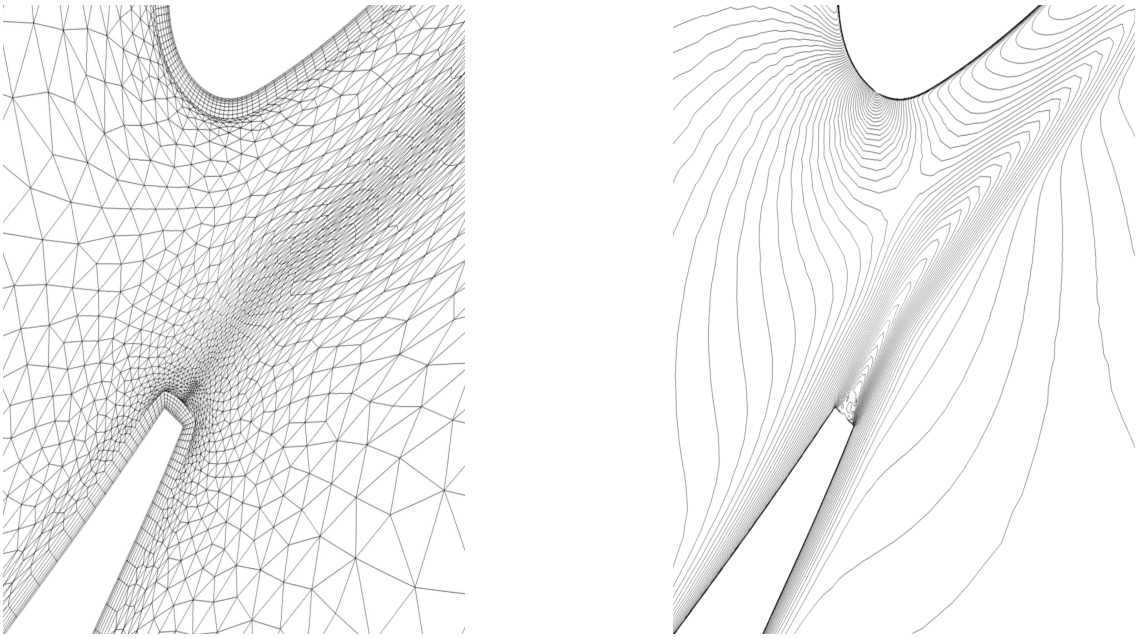


Figure 6.29: Adapted mesh with structured skin and isocontours between distributor blades.

7 Conclusions

> If we knew what it was we were doing, it would not be called research, would it?
>
> Albert Einstein

7.1 General conclusion

Remind that the goal of this thesis, according to Chapter 1, was *to develop, implement and test new methods to add more user control in unstructured triangular mesh generation.* If we are allowed to draw only one conclusion, it will be that the choices made, the software written and the test cases done, show that we reached this goal.

7.2 Analysis of the results

7.2.1 Testing results

The goal of this thesis is to show that it is possible to have a general framework for unstructured triangular mesh generation that can be deployed in any scenario and adhere to the user's requirements for his geometric domain and his application domain. In order to show that SmartMesh achieved this goal, many different problems were used as test cases.

The U-duct test case was presented simply to show that SmartMesh has the tools necessary to produce a mesh similar to the one presented by a different researcher using different tools.

The airplane wing test cases show that SmartMesh can be applied to many different scenarios. Each of the different cases have different properties and different requirements. The position of the shocks are different for each case. Without having the capability of letting the user specify his own algorithms and parameters, including the position of the shocks captured by the distance to imaginary polyline algorithm, accurate meshes could not be created and accurate solutions could not be computed with NSC2$k\varepsilon$. However, the ability to have personalized executions allows SmartMesh to create an accurate mesh and NSC2$k\varepsilon$ to compute an accurate solution for each problem.

Also worth noting is that in some cases the placement of the shocks can be found mathematically or by simple intuition while in other cases this is not as trivial. SmartMesh alone cannot be used to find the placement of the shocks in non trivial cases. The user can discover these shock locations by using one of the two methods.

The first method is an iterative approach. The user can guess where he thinks the shock will be and call the distance to imaginary polyline algorithm with these parameters. This will likely produce a bad result, but when visualizing the bad solution the isocontours will likely give him a better idea of the shock placement. The user can tweak the configuration parameters and try again. This process can be repeated until the solver produces a solution that passes the accuracy evaluation. This process is the posteriori mesh adaptation and is automatized in a loop that call an error estimator to detect features and OO$\mathcal{RT}$ to adapt a mesh accordingly.

The second method is to use algorithms to produce a very fine mesh throughout the entire area where the shock could be located. One way to accomplish this is to use the distance to imaginary polyline algorithm with a very large maximum distance. This will assure that the mesh will be fine over a large area. While using this approach makes it likely to find the location of the shock, the outputted mesh will contain a much larger number of vertices and triangles which means the solver will execute more slowly due to the extra computation required to perform calculations on all

mesh elements.

In short, SmartMesh could be used to figure out where are the unknown phenomenons, but it is not the best tool for this. However, if the location of the phenomenons is known, SmartMesh was proved to be able to drive accurately mesh generation to tackle any phenomenons. In real life, an experimented user, who has worked many years in the same application domain, will already know, before any computation, where the many phenomenons are located. In this case, SmartMesh is a flexible tool to express the a priori knowledge of the experimented users. An example of this is fluid flow in distributor of hydraulic turbine.

The hydraulic turbine distributor test cases revealed some interesting results. First, the simple distributor test cases show that given a geometric model, if the topology remains the same, the geometry can change and the same SmartMesh configuration file can be used. This is an important result because it shows that SmartMesh can be integrated into a shape optimization software without having to change the configuration for each new design tried by the optimization process.

The industrial test cases done by integrating SmartMesh into H2OMESH show that SmartMesh can produce a mesh which can be inputted into a solver and produce a solution that is very close to the one produced by H2OMESH, which is a software used in the industry to provide reliable solutions. In this case, some a priori knowledge is sufficient to create accurate meshes. It is known that there will be boundary layers, no shocks and a wake starting at the trailing edge. However, a priori knowledge of the exact path of the wake far from the trailing edge is probably not possible. The results show that the algorithms and functions available in SmartMesh are sufficient for solving real-life problems.

7.2.2 Observations

While the results show that SmartMesh can successfully be used in any domain, the integration of SmartMesh in a particular domain is a time consuming and difficult

task. One such challenge is the compatibility of file formats. For example, when integrating SmartMesh into H2OMESH, the output file from SmartMesh was incompatible with the other programs called by H2OMESH despite the fact that the output was in the correct format (`pirate`). The issue was that the `pirate` format is very flexible and H2OMESH only supports a subset of cases. With additional modifications to the parameters used in the calls by H2OMESH to other programs, the output file from SmartMesh became compatible.

The test cases done with SmartMesh integrated into H2OMESH show that CFX is a very flexible and robust CFD solver. Previously, H2OMESH testing only produced meshes with exact periodic boundaries and with an added structured skin around the distributor blades. SmartMesh produces non periodic meshes on the periodic boundaries. If the periodic boundaries are curved, then they will not fit. If the periodic boundaries are piecewise linear, then they will fit without the mesh being periodic. Also, a pure triangular mesh without an added structured skin around the blades was tested.

The results show that if the mesh has no added structured skin around the distributor blades or curved (non-periodic) boundaries, there are some disturbances in the velocity field isocontours. However, the mesh produced by SmartMesh having both the added structured skin and periodic boundaries like meshes outputted by H2OMESH produced velocity field isocontours almost identical to those generated by its industrial counterpart.

The disturbances in velocity field isocontours for certain solutions generated by SmartMesh input can be explained by some additional factors. All the test cases were done with exactly the same parameters in CFX. These simulations were not performed by an expert in CFD and turbulence modeling. While it is useful to know that CFX can perform calculations on meshes that do not have periodic boundaries or an added structured skin around the distributor blades, it is not known whether CFX can be configured to perform better under these circumstances. With further

research and intuition, these results could probably be improved.

7.3 Contributions

The mesh sizing problem is not new. However, the approach developed in this thesis presents some interesting new features.

7.3.1 General framework for intelligent unstructured mesh generation

The results in Chapter 6 show that SmartMesh can easily be applied to many different problems. The library is intended to be generic and no number is hard coded inside the library. All algorithms with their parameters can be set by the user through a configuration file. The syntax of the `libconfig` configuration file is, albeit not quite user-friendly, surely not cryptic.

This control allows us to claim that SmartMesh is general. It should be able to create suitable meshes for almost all application domains such as heat transfer, electromagnetism, stress analysis, etc, as it was able to create meshes for CFD.

Unstructured mesh generation is always automatic, but the meshes produced are ordinary and most of the time unusable. The user, with his experience of the application domain obtained after years of work, can now script his knowledge into a configuration file and produce, still automatically, an intelligent mesh suitable for accurate numerical simulation.

7.3.2 Iterative approach

The common sense taught us that the first trial is rarely the best one or the successful one. Every human learned this the hard way when trying to walk, run, bike or ski. The only way to achieve good results is with trial and error. From a mathematical point of view, we can say that a nonlinear problem cannot be solved in one iteration.

Most mesh sizing approaches found in literature are direct methods that create

a one shot mesh. There is no mechanism to improve the result. If you apply the method many times, you will get the same results every time.

The iterative approach is the only known method that can approximate accurately solutions of non linear problems. The approach used in this thesis is iterative by design. A loop composed of SmartMesh and $\mathsf{OO}\mathcal{RT}$ really improves the mesh, one iteration at a time.

7.3.3 Use the current mesh for all purposes

Most mesh sizing approaches use an extra data structure that overlap the geometric domain. This data structure is used to compute geometric features and to store some information. This is a mindless data structure but it has some justification in its context. These approaches are direct, i.e. not iterative, and claim to be able to create the perfect mesh on the first trial. Prior to building that first and final mesh, an initial data structure is needed to perform some computations and to store some values. This extra data structure is inaccurate, not body fitted and requires additional memory and programming time.

In this thesis, the approach is iterative. Prior to having a good mesh, we can live with a bad mesh, such as the one automatically generated by a Delaunay mesh generator. This mesh is far from perfect according to the precise requirements of the experimented user, but at least it discretizes the geometric domain and it is body fitted. This mesh can be used to compute geometric characteristics and distance to the boundaries will be accurate. Also, this mesh can be used to store computed values and does not require an extra data structure that overlaps the geometric domain. Furthermore, when the loop composed of SmartMesh and $\mathsf{OO}\mathcal{RT}$ goes on, this mesh is getting more and more suitable for the computations that are performed on it and for the data that it stores.

7.3.4 Independence of a CAD system

The mesh sizing problem consists essentially of creating meshes that take into account, mainly, the boundaries. The boundaries are given by a CAD system. Then any approach of the mesh sizing problem must depend on a CAD system. There are three dependencies. We must be able to compute the distance to the boundaries, the curvature of the boundaries, and the last, but not the least, to be able to mesh the geometric domain defined by the boundaries.

In the SmartMesh library, there are algorithms that depend on the distance to the boundaries and on the curvature of the boundaries. However, the SmartMesh library is independent of a CAD system. This is achieved in the SmartMesh library by replacing the boundaries of the geometric model by their discretizations provided by the mesh. So, the distance to a boundary, which is a curve in the geometric model, is replaced by the distance to a polyline. In the same manner, the curvature of a boundary, which is the curvature of a curve defined in the geometric model, is replaced by the curvature of a polyline. To replace a boundary in the geometric model by a polyline in the mesh, we need to import the topological part of the geometric model into the SmartMesh library. The topological entities help to find the mesh edges that correspond to topological edges.

While the SmartMesh library is (almost) independent of the CAD system, the SmartMesh program strongly depends on a CAD system. In this thesis, the SmartMesh program used the `pirate` CAD system. But if the user needs to link with CATIA or AutoCAD, for example, a complete new SmartMesh program must be written. Theoretically, the SmartMesh library will not need a single modification.

Finally, last but not least, any mesh sizing approach must be able to mesh the geometric domain defined by the boundaries. This strong dependence to a CAD system is completely external to the SmartMesh library and SmartMesh program. Remember that SmartMesh library and SmartMesh program do not create a mesh. They only create a field of metric tensors. The mesh generator and the mesh adapta-

tion are two programs entirely independent of the SMARTMESH library and program. In this thesis, we used Tri△ngle as mesh generator and OO$\mathcal{RT}$ as mesh adaptation. These two software hold a strong dependence to a CAD system.

7.3.5 Efficient geometric calculations done by neighbour propagation

The computation of the geometric characteristics at a given vertex can be completed in $O(1)$ in time due to the neighbour propagation technique. Each vertex, with guess information from an already processed neighbour vertex, can project itself on some boundary in very few steps. Indeed, the storage of algorithm-specific information at each vertex allows to quickly locate an edge on a boundary that is either the closest location on that boundary or one that is very close to the shortest distance. Therefore it is easy to find the closest edge without having to evaluate all edges of that boundary.

Also, geometric characteristics do not need to be evaluated on all vertices of the mesh, especially if these characteristics only affect a very small region of the mesh. Indeed, beginning at the desired topological regions, we simply propagate to neighbours until we reach the maximum distance threshold while ignoring all vertices outside of this range.

7.4 Future work

7.4.1 Smoothing algorithm

When there are too many differences between two size tensors that are on close vertices, OO$\mathcal{RT}$ has trouble creating the corresponding mesh. A smoothing algorithm could be implemented to remove extreme differences between neighbour size tensors. This is prone to happen at the maximal distance of any algorithm's application if the maximal size of size tensors at that point is a lot smaller than the size tensors beyond this distance. A smoothing algorithm can find these extreme differences and attempt

to make a smoother transition from small to large mesh triangles.

One proposed approach is to use the robust metric intersection algorithm already present in the SmartMesh library in order to perform the smoothing. Since the intersection retains the minimal area contained by both input metrics, this would allow to reduce the size of existing large size tensors.

Currently in SmartMesh we depend on users providing input parameters that make sense within the context of the domain, in order to avoid such situations.

7.4.2 Function types

Currently there are only a couple of different types of defined functions that help determine the values of size tensors based on their environment. But there are dozens of other approaches that can be used. Each new approach has advantages and disadvantages, but if used properly, can help a user to achieve his desired output mesh.

One issue with the implementation of new functions lies in the code itself, as new classes and function calls would have to be added to the library for each new function type. Even if we take advantage of class hierarchies by creating interfaces and implementing them, it is impossible to cover all scenarios. New functions could have a different type of input values and each algorithm has to create this value, as well as treating the output value appropriately.

7.4.3 Minimum and maximum size

When inputting functions into SmartMesh the user must specify minimum and maximum sizes. In essence, the minimum size is applied exactly at the source (boundary, imaginary polyline or vertex) of the algorithm, while the maximum size is applied at the maximum distance. Conceptually, the minimum is lesser than the maximum and this condition is imposed to users. However in practice this does not have to be the case.

It is possible to have situations where triangle size should decrease as we move away from a boundary. It is also possible that the size is constant between the source and the maximum distance. SmartMesh could be adapted to accommodate this change and make these scenarios available to users.

For this change to take effect, it would be very convenient to change some label names in the `libconfig` parser so that users are not mislead into thinking that the existing minimum and maximum conditions are still in place. The MINSIZE label could be changed to SIZE_AT_BORDER or SIZE_AT_TOPOLOGICAL_VERTEX to indicate that the value chosen will be applied at the source of the algorithm. Likewise, MAXSIZE could be changed to SIZE_AT_MAX_DISTANCE to indicate that the value will be applied at the maximum distance away from the source of the algorithm.

With these label changes made, users can select whichever non negative size values they want. The growth or contraction at the rate of the exponent value provided would give users a lot of added flexibility and would help SmartMesh appeal to even more domains and scenarios.

7.4.4 Curvature at topological vertex

Mathematically there is no curvature defined at each end of a curve. However, in SmartMesh there are size tensors at each mesh vertex, therefore it is necessary to determine an orientation and size at those points. It is not ideal to use the three-point radius method defined in Section 3.3.2 because the neighbours of the topological vertex are part of different curves and thus it is possible to have very extreme curvature values at this point.

The current method is to use the average size and orientation of the two neighbour points on the boundary in question. However, this approach is very generic and more could be done to take advantage of the environment.

A new method to handle this special case could lead to a more accurate solution at the point linking the two boundaries.

7.4.5 Explore additional methods to compute normals and curvature

In [15] there are many methods to compute normals and curvature on a discrete parametric curve. SmartMesh only implements one of them to define curvature values. While many of the other methods are very complicated and difficult to implement, some have proved to be more accurate and therefore there could be a benefit to SmartMesh.

Ideally, several normal and curvature methods could be made available and the user could choose which one to use in the configuration file.

7.4.6 Implementation in three dimensions

Implementing SmartMesh in three dimensions is more a software engineering task than a research topic. If we move to three dimensions, it is essentially done to deal with real industrial problems. This implies real and complex geometries done with CATIA or Pro/Engineer. Therefore three-dimensional mesh generation software must deal with CAD systems. However, the SmartMesh library was designed to be almost CAD free. It only needs the topological part of the geometric model. So, it should not be difficult to implement the SmartMesh library in three dimensions.

However, SmartMesh does not create meshes, it only creates a field of metric tensors. There must be a software that reads this field of metric tensors and creates a three-dimensional mesh that fits as close as possible the input field and also discretizes the geometric domain. This means that SmartMesh does not need to deal with a CAD system, but the problem is still there. It is the duty of the mesh generator to deal with the CAD system, and to deal with anisotropic unstructured tetrahedral meshes. OOR$\mathcal{T}$ did that task pretty well in two dimensions with anisotropic unstructured triangular meshes. However, OOR$\mathcal{T}$ is probably not robust enough for the same task in three dimensions.

In short, creating a field of metric tensors is the "easy" part. Creating an

anisotropic unstructured mesh that satisfies this field and also satisfies a CAD system is the bottleneck of the approach proposed in this thesis.

A SmartMesh user's manual

> When all else fails, read the instructions.
>
> Cann's (or Allen's) Axiom

This is a user guide to show users how to create a SMARTMESH configuration file using Libconfig syntax.

A.1 SmartMesh `libconfig` syntax

A.1.1 Regions

Regions are a uniquely named group of entities of the same dimension in the topological model. They are created so that the user can easily reference them when deciding which entities are affected by the desired algorithms.

Syntax NAME = "*any unique name*";

ENTITIES = [*comma-separated list of topological code identifiers*];

The list of regions must be assigned to a label name called CREATE_REGIONS.

A.1.2 Distance functions

Distance functions are uniquely named and represent the distance functions defined in Section 4.3.5.1. These functions are used by various algorithms to compute a size tensor based on a vertex's distance relative to a region of choice.

Syntax NAME = "*any unique name*"; MIN_SIZE = *decimal number*;

MAX_SIZE = *decimal number*;

EXPONENT = *double number*;

The list of distance functions must be assigned to a label name called CREATE_DISTANCE_FUNCTIONS.

A.1.3 Curvature-to-size functions

The curvature-to-size functions are uniquely named and invoke the function definition in Section 4.3.5.3. These functions are used by various algorithms to compute a size tensor for vertices along a target border based on the curvature calculated at that position.

Syntax NAME = "*any unique name*";

MIN_SIZE = *decimal number*;

MAX_SIZE = *decimal number*;

SIZE_OF_ERROR = *double number*;

A.1.4 Projection functions

Projection functions are uniquely named and invoke the function defined in Section 4.3.5.2. These functions are used by various algorithms to compute a size tensor based on a vertex's distance relative to a region of choice.

Syntax NAME = "*any unique name*";

MIN_SIZE = *decimal number*;

MAX_SIZE = *decimal number*;

EXPONENT = *double number*;

A.1.5 Algorithms

The algorithms are executed in the order they appear in the file and invoke the unique names of the regions and functions defined in the file. The list of algorithms must be assigned to the label CREATE_ALGORITHMS.

A.1.5.1 Uniform algorithm

Syntax TYPE = "*UNIFORM*";

SIZE = *double number*;

REGIONS = *list of defined region names*;

A.1.5.2 Distance to point algorithm

Syntax TYPE = "*DISTANCE_TO_POINT*";

FUNCTION = *defined distance function name*;

REGIONS = *list of defined region names*;

A.1.5.3 Distance to border algorithm

Syntax TYPE = "*DISTANCE_TO_BORDER*";

PARALLEL_FUNCTION = *defined distance function name*;

PERPENDICULAR_FUNCTION = *defined distance function name*;

REGIONS = *list of defined region names*;

A.1.5.4 Curvature of border algorithm

Syntax TYPE = "*CURVATURE_OF_BORDER*";

PERPENDICULAR_FUNCTION = "*defined distance function name*";

CURVATURE_TO_SIZE_FUNCTION = "*defined curvature-to-size function name*";

PROJECTION_FUNCTION = "*defined projection function name*";

REGIONS = *list of defined region names*;

A.1.5.5 Distance between two borders algorithm

Syntax TYPE = "*DISTANCE_BETWEEN_TWO_BORDERS*";

PARALLEL_FUNCTION = *defined distance function name*;

PERPENDICULAR_FUNCTION = *defined distance function name*;

REGIONS = "*list of defined region names for first border*";

REGIONS2 = "*list of defined region names for second border*";

A.1.5.6 Distance to imaginary polyline algorithm

Syntax TYPE = "*DISTANCE_TO_IMAGINARY_POLYLINE*";

PARALLEL_FUNCTION = *defined distance function name*;

PERPENDICULAR_FUNCTION = *defined distance function name*;

PROJECTION_FUNCTION = *defined projection function name*;

REGIONS = *list of defined region names*;

A.1.6 Configuration file example

Here is an example of a configuration file. Note that this file goes through cpp and so can contain cpp instructions.

```
#define GUIDEVANE 11, 12, 1, 2, 3, 4, 5, 6, 7


// REGIONS note:  NAME must be unique
CREATE_REGIONS =
(


   { NAME = "Inlet";        TOPOLOGICAL_ENTITIES = [ 301 ]; },
   { NAME = "Outlet";       TOPOLOGICAL_ENTITIES = [ 305 ]; },
   { NAME = "Periodic";     TOPOLOGICAL_ENTITIES = [ 302, 303, 304, 306,
                                                     307, 308 ]; },
```

```
   { NAME = "StayVane";     TOPOLOGICAL_ENTITIES = [ 101, 102, 103, 104,
                                                     105, 106, 107]; },
   { NAME = "GuideVane";    TOPOLOGICAL_ENTITIES = [ 201, 202, 203]; },
   { NAME = "Corners";      TOPOLOGICAL_ENTITIES = [ GUIDEVANE ]; },
   { NAME = "Fluid";        TOPOLOGICAL_ENTITIES = [ 1000 ]; }
);

CREATE_POLYLINES =
(
   { NAME = "POLYLINE1";
     X_COORDS = [ -50.0, -48.0, -46.0, -44.0 ];
     Y_COORDS = [ -12.8, -13.5, -13.5, -13.0 ]; }
);

CREATE_DISTANCE_FUNCTIONS =
(
   {
     NAME = "FStayVanePerp";  MINSIZE = 0.1;
     MAXSIZE = 1.0; MAXDISTANCE = 5.0;
     EXPONENT = 1.0;
   },

   {
     NAME = "FStayVanePara";   MINSIZE = 1.0;
     MAXSIZE = 2.0; MAXDISTANCE = 5.0;
     EXPONENT = 1.0;
   },
```

```
{
  NAME = "FGuideVanePerp";  MINSIZE = 0.1;
  MAXSIZE = 1.0; MAXDISTANCE = 5.0;
  EXPONENT = 1.0;
},

{
  NAME = "FGuideVanePara";  MINSIZE = 1.0;
  MAXSIZE = 2.0; MAXDISTANCE = 5.0;
  EXPONENT = 1.0;
},

{
  NAME = "Fhaut";    MINSIZE = 0.04;
  MAXSIZE = 0.05;   MAXDISTANCE = 0.5;
  EXPONENT = 1.0;
},

{
  NAME = "F3";    MINSIZE = 0.05;
  MAXSIZE = 0.25;   MAXDISTANCE = 0.5;
  EXPONENT = 3.0;
},

{
  NAME = "Fcorners";   MINSIZE = 0.02;
  MAXSIZE = 0.5;   MAXDISTANCE = 0.75;
  EXPONENT = 2.0;
```

```
   },

   {
     NAME = "FLinePerp";   MINSIZE = 0.03;
     MAXSIZE = 0.5;   MAXDISTANCE = 2.0;
     EXPONENT = 1.0;
   },

   {
     NAME = "FLinePara";   MINSIZE = 0.03;
     MAXSIZE = 1.0;   MAXDISTANCE = 2.0;
     EXPONENT = 1.0;
   }
);

CREATE_PROJECTION_FUNCTIONS =
(
   { NAME = "P1"; MAXSIZE = 2.0; MAXDISTANCE = 1.0; EXPONENT = 1.0; },
   { NAME = "P2"; MAXSIZE = 2.0; MAXDISTANCE = 2.0; EXPONENT = 1.0; }
);

CREATE_CURVATURE_TO_SIZE_FUNCTIONS =
(
   { NAME = "C1"; MINSIZE = 0.1; MAXSIZE = 1.0; SIZE_OF_ERROR = 0.1; }
);

CREATE_ALGORITHMS =
(
```

```
    { TYPE = "UNIFORM"; SIZE = 2.0; REGIONS = ( "Fluid" ); },

    { TYPE = "DISTANCE_TO_POINT"; FUNCTION = "Fcorners";
      REGIONS = ( "Corners" ); },

    { TYPE = "DISTANCE_TO_BORDER"; PERPENDICULAR_FUNCTION = "FStayVanePerp";
      PARALLEL_FUNCTION = "FStayVanePara"; REGIONS = ( "StayVane" ); },

    { TYPE = "DISTANCE_TO_BORDER"; PERPENDICULAR_FUNCTION = "FGuideVanePerp";
      PARALLEL_FUNCTION = "FGuideVanePara"; REGIONS = ( "GuideVane" ); },

    { TYPE = "DISTANCE_TO_IMAGINARY_POLYLINE";
      PERPENDICULAR_FUNCTION = "FLinePerp";
      POLYLINE_DISTANCE_FUNCTION = "FLinePara";
      PARALLEL_PROJECTION_FUNCTION = "P1";
      COORDINATES = "POLYLINE1"; REGIONS = ( "Fluid" ); }
);
```

B Metric intersection

> It is easier to write an incorrect program than understand a correct one.
>
> Alan J. Perlis

Here is the Maple code for the robust metric intersection method defined in Section 3.2.2. This code was used as source for the metric intersection method implemented in SmartMesh. Note that Maple provides many built-in mathematical functions that are not available in C++ therefore some of the commands shown here had to be implemented manually in SmartMesh.

You will notice the `EigenVV2D` command in this code. This is an in-house function that, given a symmetric 2×2 input matrix, returns the two eigenvectors along with each eigenvectors's corresponding eigenvalue using the Jacobi method.

```
e1  := G:-get_Eigenvalues()[1];
e2  := G:-get_Eigenvalues()[2];
pt1 := multiply( matrix([[e1*cos(.75),e2*sin(.75)]]),
       transpose(G:-get_Eigenvectors()) );
pt2 := multiply( matrix([[e1*cos(3.14),e2*sin(3.14)]]),
       transpose(G:-get_Eigenvectors()) );
pt3 := multiply( matrix([[e1*cos(1.5),e2*sin(1.5)]]),
       transpose(G:-get_Eigenvectors()) );


Rot := transpose( F:-get_Eigenvectors() );
```

```
e1 := F:-get_Eigenvalues()[1];
e2 := F:-get_Eigenvalues()[2];
invRot := F:-get_Eigenvectors();
scale := matrix( [ [ 1/e1, 0 ], [ 0, 1/e2 ] ] );
invScale := matrix( [ [ e1, 0 ], [ 0, e2 ] ] );

p1 := multiply( Rot, transpose(pt1) );
p1 := multiply( scale, p1 );
p2 := multiply( Rot, transpose(pt2) );
p2 := multiply( scale, p2 );
p3 := multiply( Rot, transpose(pt3) );
p3 := multiply( scale, p3 );
f1 := eval(a*x^2 +2*b*x*y + c*y^2 = 1, {x=p1[1,1], y=p1[2,1]} );
f2 := eval(a*x^2 +2*b*x*y + c*y^2 = 1, {x=p2[1,1], y=p2[2,1]} );
f3 := eval(a*x^2 +2*b*x*y + c*y^2 = 1, {x=p3[1,1], y=p3[2,1]} );
constants := solve( {f1, f2, f3}, {a, b, c} );
constants := op(constants);
for i in constants do
    if op(i)[1] = a then a := op(i)[2] end if;
    if op(i)[1] = b then b := op(i)[2] end if;
    if op(i)[1] = c then c := op(i)[2] end if;
end do;

# Rewrite the previous lines uising EigenVV2D
OutputList := EigenVV2D( a, b, c );
evalues    := [ abs(OutputList[1][1]), abs(OutputList[1][2]) ];
R          := OutputList[2];
```

```
d1 := 1/sqrt(evalues[1]);
d2 := 1/sqrt(evalues[2]);

if operation = "intersection" then
   if d1 > 1 then d1 := 1 end if;
   if d2 > 1 then d2 := 1 end if;
else
   if d1 < 1 then d1 := 1 end if;
   if d2 < 1 then d2 := 1 end if;
end if;

pt1 := multiply( matrix( [[ d1*cos(.75), d2*sin(.75) ]] ), transpose(R) );
pt2 := multiply( matrix( [[ d1*cos(3.14), d2*sin(3.14) ]] ), transpose(R) );
pt3 := multiply( matrix( [[ d1*cos(1.5), d2*sin(1.5) ]] ), transpose(R) );
p1  := multiply( invScale, transpose(pt1) );
p1  := multiply( invRot, p1 );
p2  := multiply( invScale, transpose(pt2) );
p2  := multiply( invRot, p2 );
p3  := multiply( invScale, transpose(pt3) );
p3  := multiply( invRot, p3 );
a   := evaln(a);
b   := evaln(b);
c   := evaln(c);
f1  := eval(a*x^2 +2*b*x*y + c*y^2 = 1, {x=p1[1,1], y=p1[2,1]} );
f2  := eval(a*x^2 +2*b*x*y + c*y^2 = 1, {x=p2[1,1], y=p2[2,1]} );
f3  := eval(a*x^2 +2*b*x*y + c*y^2 = 1, {x=p3[1,1], y=p3[2,1]} );

constants := solve( {f1,f2,f3}, {a,b,c} );
```

```
constants := op(constants);

# Solutions return unsorted, sort to identify the correct values for
# each constant a_i.
for i in constants do
    if op(i)[1] = a then a := op(i)[2] end if;
    if op(i)[1] = b then b := op(i)[2] end if;
    if op(i)[1] = c then c := op(i)[2] end if;
end do;

# Rewrite the previous lines uising EigenVV2D
OutputList := EigenVV2D( a, b, c );
evalues    := [ abs(OutputList[1][1]), abs(OutputList[1][2]) ];
R          := OutputList[2];
temp       := multiply( R, matrix( [[ 1/sqrt(evalues[1]), 0 ],
              [ 0, 1/sqrt(evalues[2]) ]] ) );
newInt     := multiply( temp, transpose(R) );

# It may be a Size or a SquareSize, or a Density object that was
# returned, but we create a SquareDensity object because that
# kind of object does nothing with the eigenvalues.
# But the type may be wrong... We must change the type of the
# output to be the same as the type of the two inputs.

IntersectedMetric := Metric2D( matrix([[newInt[1,1],
newInt[1,2]],[newInt[1,2],newInt[2,2]]]), "SquareDensity");
IntersectedMetric:-set_Type( F:-get_MetricType() );
```

C Javadoc documentation of SmartMesh

> Give a man a fish and he will eat for a day; Teach a man to fish and he will eat for a lifetime. The moral? READ THE MANUAL!
>
> Sign on a computer system consultant's desk

Please see attached disc for a detailed documentation of the SmartMesh code base. It is available in HTML and PDF format.

Bibliography

[1] D. Aït-Ali-Yahia, G. Baruzzi, W. G. Habashi, M. Fortin, J. Dompierre, and M.-G. Vallet. Anisotropic mesh adaptation: Towards user-independent, mesh-independent and solver-independent CFD. Part II: Structured grids. *International Journal for Numerical Methods in Fluids*, 39(8):657–673, July 2002.

[2] F. Alauzet. Size gradation control of anisotropic meshes. *Finite Elements in Analysis and Design*, 46:181–202, 2010.

[3] F. Alauzet and P. J. Frey. Estimateur d'erreur géométrique et métriques anisotropes pour l'adaptation de maillage. Partie I: aspects théoriques. Technical Report 4759, Institut National de Recherche en Informatique et en Automatique, France, March 2003.

[4] F. Alauzet and P. J. Frey. Estimateur d'erreur géométrique et métriques anisotropes pour l'adaptation de maillage. Partie II: exemples d'applications. Technical Report 4789, Institut National de Recherche en Informatique et en Automatique, France, March 2003.

[5] F. Alauzet and A. Loiselle. High-order sonic boom modeling based on adaptive methods. *Journal of Computational Physics*, 229:561–593, 2010.

[6] F. Alauzet, P. J. Frey, P. L. George, and B. Mohammadi. 3D transient fixed point mesh adaptation for time-dependent problems: Application to CFD simulations. *Journal of Computational Physics*, 222(2):592–

623, 2007. ISSN 0021-9991. doi: DOI: 10.1016/j.jcp.2006.08.012. URL http://www.sciencedirect.com/science/article/B6WHY-4M69P8W-1/2/a6bd4dd08ba6a8c

[7] Éric Béchet. *Résolution d'un problème aux limites à frontières libres au moyen d'un algorithme de remaillage adaptatif et anisotrope*. PhD thesis, École Polytechnique de Montréal, 2002.

[8] M.-O. Bristeau, R. Glowinski, J. Périaux, and H. Viviand, editors. *Numerical Simulation of Compressible Navier-Stokes Flows*. Notes in Numerical Fluid Mechanics, Vol. 18, A GAMM-Workshop, Friedr. Vieweg & Sohn, Braunschweig/Wiesbaden, 1987.

[9] J. R. Cebral and R. Löhner. Flow visualization on unstructured grids using geometrical cuts, vortex detection and shock surfaces. In *39th Aerospace Sciences Meeting and Exhibit*, number AIAA–2001–0915, Reno, NV, January 1995. AIAA.

[10] A. Cuhna, S. Canann, and S. Saigal. Automatic boundary sizing for 2D and 3D meshes. In *AMD-Vol. 220 Trends in Unstructured Mesh Generation*, pages 65–72, Evanston. IL, July 1997. Joint ASME/ASCE/SES Summer Meeting.

[11] J.-C. Cuillière. An adaptive method for the automatic triangulation of 3D parametric surfaces. *Computer-Aided Design*, 30(2):139–149, 1998.

[12] A. Cunha, S. A. Canann, and S. Saigal. Automatic boundary sizing for 2D and 3D meshes. In *AMD-Vol. 220 Trends in Unstructured Mesh Generation*, pages 65–72. ASME, July 1997.

[13] B. Delaunay. Sur la sphère vide. *Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk*, 7:793–800, 1934.

[14] J. Dompierre, F. Guibault, J.-F. Dubé, H. Iepan, and T. C. Vu. Hybrid O-mesh generation for hydraulic vane optimization. In *10th ISGG Conference on Numerical Grid Generation*, Forth, Crete, Greece, September 2007.

[15] K. Fontaine. Numerical methods for first and second derivatives of two-dimensional discrete parametric curves. Technical report, Laurentian University, Sudbury, ON, August 2010.

[16] V. François and J.-C. Cuillière. 3D automatic remeshing applied to model modification. *Computer-Aided Design*, 32(7):433–444, 2000.

[17] P. J. Frey and P.-L. George. *Mesh Generation. Application to Finite Elements*. Hermès, Paris, 2000.

[18] P. J. Frey and L. Maréchal. Fast adaptive quadtree mesh generation. In *Seventh International Meshing Roundtable*, pages 211–224, Dearborn, Michigan, October 1998. Sandia National Laboratories.

[19] C. Gruau. *Génération de métriques pour adaptation anisotrope de maillages: applications à la mise en forme des matériaux*. PhD thesis Mécanique numérique, ENSMP - CEMEF Centre de Mise en Forme des Matériaux, ENSMP, 2004.

[20] C. Gruau and T. Coupez. 3D tetrahedral, unstructured and anisotropic mesh generation with adaptation to natural and multidomain metric. *Computer Methods in Applied Mechanics and Engineering*, 194(48–49):4951–76, 2005.

[21] F. Guibault, Y. Zhang, J. Dompierre, and T. C. Vu. Robust and automatic CAD-based structured mesh generation for hydraulic turbine component optimization. In *Proceedings of the 23rd IAHR Symposium*, Yokohama, Japan, October 2006.

[22] D. J. Jones. Test cases for inviscid flow field methods: Reference test cases and contributions. Technical Report AR-211, AGARD, NATO, 1985.

[23] P. Labbé, J. Dompierre, M.-G. Vallet, F. Guibault, and J.-Y. Trépanier. A universal measure of the conformity of a mesh with respect to an anisotropic metric field. *International Journal for Numerical Methods in Engineering*, 61: 2675–2695, 2004.

[24] C. L. Lawson. Generation of a triangular grid with application to countour plotting. Technical Report JPL–299, California Institute of Technology, 1972.

[25] C. K. Lee. Automatic adaptive mesh generation using metric advancing front approach. *Engineering Computations*, 16(2):230–263, 1999.

[26] C. K. Lee. On curvature element-size control in metric surface mesh generation. *International Journal for Numerical Methods in Engineering*, 50:787–807, 2001.

[27] F. Loiselle, A. Dervieux, and F. Alauzet. Fully anisotropic goal-oriented mesh adaptation for 3D steady Euler equations. *Journal of Computational Physics*, 229:2866–2897, 2010.

[28] S. McKenzie, J. Dompierre, A. Turcotte, and E. Meng. On metric tensor representation, intersection, and union. In *11th ISGG Conference on Numerical Grid Generation*, Montréal, QC, May 2009.

[29] B. Mohammadi. Fluid dynamics computation with NSC2KE, an user-guide, release 1.0. Technical Report RT-0164, Institut National de Recherche en Informatique et en Automatique, May 1994.

[30] T. H. Pulliam and J. T. Barton. Euler computation of AGARD working group 07 airfoil test cases. In *AIAA 23rd Aerospace Sciences Meeting*, number AIAA–85–0018, Reno, NV, January 1985.

[31] M. Purdy. On metric tensor merging operations. Technical report, Laurentian University, Sudbury, March 2010.

[32] É. Seveno. *Génération automatique de maillages tridimensionnels isotropes par une méthode frontale*. PhD thesis, Université Pierre et Marie Curie, Paris VI, March 1998.

[33] J. R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In M. C. Lin and D. Manocha, editors, *Applied Computational*

Geometry: Towards Geometric Engineering, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996.

[34] Y. Sirois, J. Dompierre, M.-G. Vallet, and F. Guibault. Hybrid mesh smoothing based on Riemannian metric non-conformity minimization. *Finite Elements in Analysis and Design*, 46:47–60, January 2010.

[35] K.-F. Tchon, M. Khachan, F. Guibault, and R. Camarero. Three-dimensional anisotropic geometric metrics based on local domain curvature and thickness. *Computer-Aided Design*, 37(2):173–187, 2005.

[36] M.-G. Vallet. *Génération de maillages éléments finis anisotropes et adaptatifs*. PhD thesis, Université Pierre et Marie Curie, Paris VI, France, 1992.

[37] G. P. Warren, W. K. Anderson, J. L. Thomas, and S. L. Krist. Grid convergence for adaptive methods. In *AIAA 29nd Aerospace Sciences Meeting*, number AIAA–91–1592–CP, Reno, NV, January 1991.