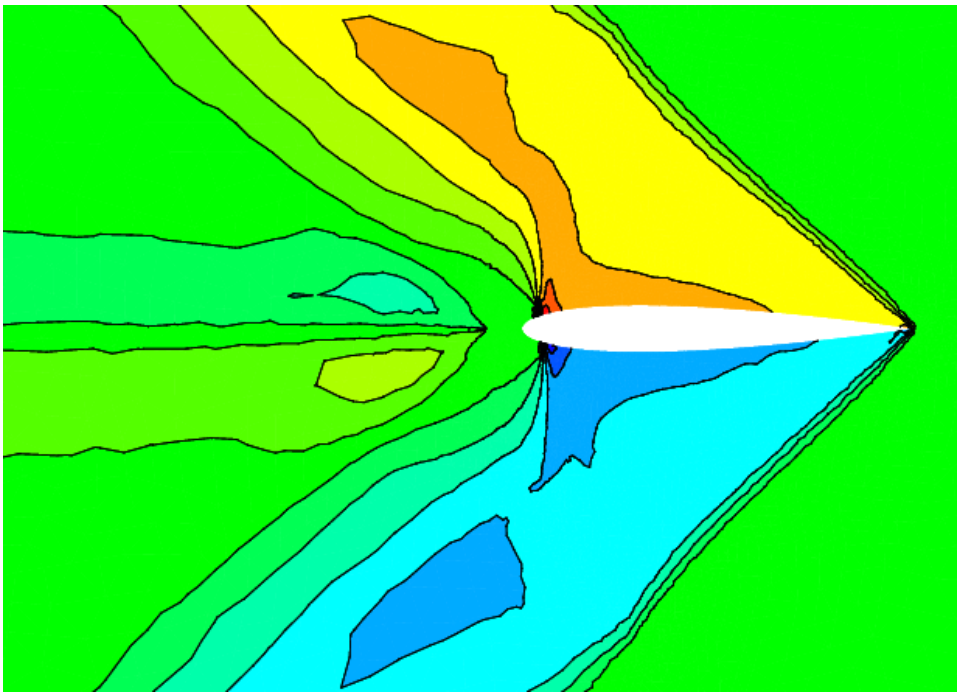
Master of Science Thesis

Goal oriented adaptation of unstructured meshes

Application to finite volume methods

Giovanni Todarello

February 28, 2014



Aerodynamics

Faculty of Aerospace Engineering

TU Delft

Delft University of Technology

Goal oriented adaptation of unstructured meshes

Application to finite volume methods

Master of Science Thesis

For obtaining the degree of Master of Science in
Aerospace Engineering at Delft University of Technology

Giovanni Todarello

February 28, 2014

Faculty of Aerospace Engineering · Delft University of Technology

Delft University of Technology

DELFT UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF AERODYNAMICS

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance the thesis entitled **"Goal oriented adaptation of unstructured meshes"** by **Giovanni Todarello** in fulfillment of the requirements for the degree of **Master of Science**.

Dated: February 28, 2014

Graduation commitee:

Dr. ir. M.I. Gerritsma

Dr. R.P. Dwight

Dr. J. Peter (ONERA, France)

Ir. I. Azijli

TABLE OF CONTENTS

List of Figures **vii**

List of Tables **ix**

List of symbols and acronyms **xi**
List of Symbols xi
List of Acronyms xii

Acknowledgements **xv**

Abstract **xvii**

1 Literature review **1**
1.1 Introduction 1
1.2 Goal oriented mesh adaptation for finite volume schemes . . . 2

2 Gradient computation methods **7**
2.1 The *elsA Opt* module 7
2.2 Finite differences 9
2.3 Discrete direct differentiation method 9
2.4 Discrete adjoint method 10
2.4.1 Parameter mode 10
2.4.2 Mesh mode 11
2.5 Solution of the linear systems 11
2.6 Recursive projection method 12

3 Code development **15**
3.1 Framework of the implementation 15
3.1.1 MUSCL 2nd order scheme 19
3.2 Implementation of discrete gradient computation methods . . 20

3.2.1 Direct differentiation method 20
3.2.2 Discrete adjoint method (parameter mode) 26
3.2.3 Discrete adjoint method (mesh mode) 28

4 Code verification **33**
4.1 Test case NACA0012 33
4.1.1 Mesh 33
4.1.2 Flow 34
4.1.3 Parameter α 35
4.2 Verification procedure 37
4.2.1 Direct differentiation method 37
4.2.2 Adjoint method (parameter mode) 43
4.2.3 Adjoint method (mesh mode) 44

5 Mesh adaptation **49**
5.1 Adaptation procedure 49
5.2 Results NACA0012 - Lift and drag 56
5.2.1 $M = 0.5$ $AoA = 0°$ 57
5.2.2 $M = 0.85$ $AoA = 2°$ 60
5.2.3 $M = 1.50$ $AoA = 1°$ 64

6 Conclusions **69**

Bibliography **73**

A Goal function partial derivatives **77**

B Python package for mesh adaptation **81**

C Adjoint method subroutine **87**

LIST OF FIGURES

2.1 Convergence of adjoint system of equations for computation of drag derivatives using RPM (NACA0012 - 4841 nodes triangular mesh). 13

3.1 Normal vector orientation and interface states. 18
3.2 Block of cells involved in the computation of the flux Jacobian at the highlighted interface 21
3.3 Structure of the code 22
3.4 Block of cells involved in the computation of $\frac{\partial R_C}{\partial W}\frac{dW}{d\alpha}$. 24
3.5 Structure of the code 27
3.6 Only internal interface states marked with * are involved in the definition of ω in cell A. 29
3.7 Structure of the code 31

4.1 Detail of the mesh in proximity of the airfoil. 33
4.2 View of the mesh. 34
4.3 Density contour plot at steady state. 35
4.4 Mach contour plot at steady state. 35
4.5 Mesh rotation vanishing in the vicinity of the boundary (for visualization purposes the value of $d\alpha$ in this picture is much larger than the $d\alpha$ used). 36
4.6 Detail of the $\pm\alpha$ shifted meshes near the trailing edge ($d\alpha = 5 \cdot 10^{-5}$). 37
4.7 $\frac{\partial\left(\frac{\partial u}{\partial y}\right)}{\partial W}\frac{dW}{d\alpha}$ computed with DDM and FD respectively 39
4.8 $\frac{\partial\left(\frac{\partial p}{\partial x}\right)}{\partial W}\frac{dW}{d\alpha}$ computed with DDM and FD respectively. 39
4.9 Density sensitivity computed with DDM and FD respectively. 40
4.10 x-momentum sensitivity computed with DDM and FD respectively 40

4.11 Sensitivity of the 1st component of the flux balance with respect to the flow, computed with DDM and FD respectively. 41
4.12 Sensitivity of the 2nd component of the flux balance with respect to the flow, computed with DDM and FD respectively. 41
4.13 Comparison between partial sensitivity with respect to the mesh of the flux balance and total sensitivity of the flux balance (1st component). 42
4.14 Comparison between partial sensitivity with respect to the mesh of the flux balance and total sensitivity of the flux balance (2nd component). 42
4.15 First component of the adjoint vector related to drag. 45
4.16 First component of the adjoint vector related to lift. 45
4.17 Node displacement in shock region (respectively plus and minus dX). 46
4.18 Plot of $dJdX$ vector for drag. 47
4.19 Plot of $dJdX$ vector for lift. 47

5.1 Adaptation procedure. 50
5.2 All the quantities involved in the adaptation procedure. . . . 55
5.3 The initial mesh 56
5.4 Subsonic case - 1st component of adjoint vector relative to lift. 57
5.5 Subsonic case - $\frac{dJ}{dX}$ vector field. 57
5.6 Subsonic case - 1st adaptation step. 58
5.7 Subsonic case - 2nd adaptation step. 58
5.8 Subsonic case - Plot of minus the logarithm of the estimator. 59
5.9 Transonic case - 1st component of adjoint vector relative to lift. 60
5.10 Transonic case - $\frac{dL}{dX}$ vector field. 60
5.11 Transonic case - 1st adaptation step. 61
5.12 Transonic case - 2nd adaptation step. 61
5.13 Transonic case - 3rd adaptation step. 62
5.14 Transonic case - Plot of minus the logarithm of the estimator. 63
5.15 Supersonic case - 1st component of the adjoint vector relative to lift. 64
5.16 Supersonic case - $\frac{dL}{dX}$ vector field. 64
5.17 Supersonic case - 1st adaptation step. 65
5.18 Supersonic case - 2nd adaptation step. 65
5.19 Supersonic case - 3rd adaptation step. 66
5.20 Supersonic case - Plot of minus the logarithm of the estimator. 67

LIST OF TABLES

4.1 Direct differentiation - Drag sensitivity to parameter α. . . . 43
4.2 Direct differentiation - Lift sensitivity to parameter α. 43
4.3 Adjoint (parameter mode) - Drag sensitivity to parameter α. 44
4.4 Adjoint (parameter mode) - Lift sensitivity to parameter α. . 44
4.5 Adjoint (mesh mode) - Drag sensitivity to parameter α. . . . 48
4.6 Adjoint (mesh mode) - Lift sensitivity to parameter α. 48

5.1 Subsonic case - Number of nodes at each step. 59
5.2 Subsonic case - Average values of the estimator at each step. 59
5.3 Transonic case - Number of nodes at each step. 63
5.4 Transonic case - Average values of the estimator at each step. 63
5.5 Supersonic case - Number of nodes at each step. 67
5.6 Supersonic case - Average values of the estimator at each step. 67

LIST OF SYMBOLS AND ACRONYMS

List of Symbols

$\boldsymbol{\alpha}$ - Vector of design parameters.

$\boldsymbol{\beta}$ - Cell-centered contribution to Van Albada limiter.

$\boldsymbol{\epsilon}$ - Tolerance for mesh adaptation.

$\boldsymbol{\eta}$ - Mesh adaptation estimator .

$\boldsymbol{\gamma}$ - Heat capacity ratio.

$\boldsymbol{\lambda}$ - Adjoint vector.

$\boldsymbol{\mu}_{init}$ - Initial metric.

$\boldsymbol{\mu}_{target}$ - Target metric.

$\boldsymbol{\omega}$ - Auxiliary variable in adjoint method code (parameter mode).

$\boldsymbol{\phi}$ - Van Albada limiter.

$\boldsymbol{\psi}$ - Percentage of spectral radius below which Harten correction is applied.

$\boldsymbol{\rho}$ - Density.

$\boldsymbol{\theta}$ - Mesh adaptation estimator (including characteristic length).

$\boldsymbol{\zeta}$ - Upwind contribution to Van Albada limiter.

$\mathbf{D}$ - Drag.

$\mathbf{E}$ - Energy per unit mass.

$\mathbf{F}$ - Flux (interface quantity).

$\mathbf{J}$ - Vector of goal functions of interest.

$\mathbf{L}$ - Lift.

$\mathbf{P^+}$ - Interface state relative to right cell.

$\mathbf{P^-}$ - Interface state relative to left cell.

$\mathbf{P_b}$ - Primitive variable at the borders of the domain.

$\mathbf{P}$ - Primitive variable.

$\mathbf{R}$ - Flux balance without division by the volume of the cell (cell centered quantity).

$\mathbf{S}$ - Surface.

$\mathbf{V}$ - Vector.

$\mathbf{W_b}$ - Conservative variables at the borders.

$\mathbf{W}$ - Conservative variables.

$\mathbf{X}$ - Mesh nodes coordinates.

$\tilde{\mathbf{A}}$ - Roe average matrix.

$\mathbf{f}$ - Split factor.

$\mathbf{k^{(2)}}, \mathbf{k^{(4)}}$ - Dissipation coefficients in Jameson scheme.

$\mathbf{n_\alpha}$ - Number of design parameters.

$\mathbf{n_f}$ - Number of goal functions of interest.

$\mathbf{n_n}$ - Number of nodes.

$\mathbf{p}$ - Pressure.

$\mathbf{s}$ - Entropy.

$\mathbf{u}$ - Velocity component in x direction.

$\mathbf{v}$ - Velocity component in y direction.

$\mathbf{w}$ - Velocity component in z direction.

List of Acronyms

elsA - Ensemble Logiciel de Simulation en Aérodinamique.

AoA - Angle of Attack.

CFD - Computational Fluid Dynamics.

CGNS - CFD General Notation System.

DDM - Direct Differentiation Method.

DSNA - Département Simulation Numérique des écoulements et Aéroacoustique.

FD - Finite Difference.

FDM - Flux Difference Method.

FVS - Flux Vector Splitting.

INRIA - Institut National de Recherche en Informatique et en Automatique.

MUSCL - Monotone Upstream-centered Schemes for Conservation Laws.

NACA - National Advisory Committee for Aeronautics.

ONERA - Office National d'Études et de Recherches Aérospatiales.

RPM - Recursive Projection Method.

ACKNOWLEDGEMENTS

The presesent thesis has been the last task of the Master Programme in Aerodynamics at TUDelft and at the same time the most intense, interesting and formative. I want to thank, in this brief section, all the people who contributed to the accomplishment of this work and to my path towards graduation.

First of all I want to thank infinitely Jacques Peter, my supervisor from the DSNA department of ONERA, France, with whom I had the chance to work on a daily basis. He gave continuous support throughout the whole thesis period at ONERA, with precious suggestions to my work. He has always been able to address me towards the most effective solution to problems and without his guidance it would have not been possible to obtain the same result. I also want to thank Sébastien Bourasseau, a PhD student of the DSNA department of ONERA, for his help and suggestions in the code development work. Moreover, I want to thank the whole DSNA department of ONERA in Chatillon (France) for making me feel at home and for keeping the work atmosphere always pleasant and enjoyable.

I want to thank my supervisor from TUDelft, Richard Dwight, for giving me the possibility to take part to such an interesting project, for the suggestions and corrections to my thesis that he has always been available to give and for his help in arranging the thesis defence. I also want to thank Dr. ir. M.I. Gerritsma and Ir. I. Azijli for taking part to the graduation commitee.

I want to thank all my friends and all the amazing people that I met during my Master Programme at TUDelft and during the international experiences in France and Germany.

Most of all I want to thank my parents, for the huge support given to

me during my student career and for giving me the possibility to accomplish my goals. Without them this achievement could have not been possible and it's to them that this work is dedicated.

Giovanni Todarello

February 28, 2014
Pisa, Italy

ABSTRACT

In the present thesis report the author synthetizes 9 months of work at the DSNA department of ONERA in Chatillon, France. The topic of the thesis is goal oriented mesh adaptation with particular application to unstructured grids and to finite volume methods. The motivation of the present work is the application to unstructured meshes of a novel indicator for mesh adaptation, based on the total derivative of the goal function with respect to mesh nodes coordinates, introduced by Peter et al. in [6], [7] and that has been tested until now on structured grids only.
In chapter 1 a brief literature survey is presented, with the aim of introducing the theoretical background of the work and the state of the art of goal oriented mesh adaptation techniques.
In chapter 2 the author gives a description of the gradient computation module of the CFD software *elsA*, developed by ONERA, that has been the main tool used for flow simulations and mesh adaptation and the core of the code development work.
The description of the technical work carried out starts in chapter 3, in which the author describes and explains the code implementation, that has been necessary to compute the local indicator for mesh adaptation.
In chapter 4, the complete code verification chain is described, and the results of the procedure are summarized in terms of accuracy of the gradients computed with respect to the results obtain with finite differences.
Finally, in chapter 5, the author presents the mesh adaptation chain developed and the results obtained after the application of the same adaptation chain on three test-cases, subsonic, transonic and supersonic respectively.

CHAPTER 1
LITERATURE REVIEW

1.1 Introduction

Goal oriented mesh refinement techniques are of great importance in numerical simulations for engineering applications. In many practical cases it is sufficient to have a good estimate of an integral function of interest rather than resolving with accuracy all the details. In particular in fluid dynamics, the quantities of interest are often lift and drag and grid adaptation techniques become important, resulting in considerable computational cost saving.
There are mainly two fundamental types of grid adaptation methods: feature based methods and adjoint based methods. A clear and exhaustive comparison of the two types has been presented by Balasubramanian and Newman [1]. Feature based methods aim to refine the grid in physically significant zones like shock waves and boundary layers, in order to improve the accuracy of the flow in these areas [2, 3, 4]. The most common technique of this type is gradient based grid adaptation, in which the idea is to refine the grid more severely in zones with a high gradient of the solution. However, this technique may lead to erroneous results as shown in [5], in which the shock position found by locally refining the mesh based on the pressure gradient was different by the position found by refining the mesh globally. Adjoint based grid adaptation techniques, based on the definition of the dual problem, will be discussed more in detail in the following. The grid adaptation technique adopted in the thesis belongs to this class of methods. The objective of the thesis is to analyze the properties of the derivative of the goal function with respect to mesh coordinates used as a local indicator for goal oriented adaptation of unstructured grids in the framework of Euler equations discretized using finite volume schemes. At present, this indicator has been tested already on structured meshes [6, 7]. The idea and

motivation of the thesis is to extend its application to unstructured grids, where the adaptation capability can be exploited to full extent without the limitations deriving from the structured grid constraint.

1.2 Goal oriented mesh adaptation for finite volume schemes

Adjoint based methods were developed during the 90's in the framework of finite element methods and are based on the duality property of the adjoint solution [8, 9, 10, 11, 12]. This class of grid adaptation methods is always preceded by an estimation of the local error on which the refinement strategy will be based. The first important generalization of adjoint based error estimation techniques from the limited field of finite element methods to a broader framework was introduced by Pierce and Giles [13, 14].
Considering the well-posed linear differential problem:

$$Lu = f, \tag{1.1}$$

defined on a domain Ω and complete with boundary conditions, where f is a function of the Hilbert space $L_2(\Omega)$ with inner product denoted by $(.,.)$, the corresponding adjoint operator L^* is defined by the relation:

$$(L^*\lambda, u) = (\lambda, Lu), \tag{1.2}$$

in which λ is the solution of the equation:

$$L^*\lambda = g, \tag{1.3}$$

and it's called adjoint vector. Given the integral quantity $J = (g, u)$, it can be expressed equivalently as the inner product between λ and f:

$$(g, u) = (L^*\lambda, u) = (\lambda, Lu) = (\lambda, f). \tag{1.4}$$

The error that is made by computing J with an approximation u_h of the linear differential problem is:

$$(g, u) - (g, u_h) = (\lambda, f - Lu_h). \tag{1.5}$$

This is a crucial relation, which is the basis of all adjoint based error estimation and grid adaptation techniques. It states that the error in the computation of the integral function J (could be for example lift), due to the numerical approximation $u_h \approx u$ (could be for example a numerical approximation of the flow field), can be expressed as a weighted sum of the residual components in the domain (the weights being the adjoint vector components). However this expression is not so useful in this form, since

the exact adjoint solution of the vector has the same computational cost of the exact solution of the primal problem. It is more convenient to express the error in terms of an approximate adjoint vector λ_h as:

$$(g,u) - (g,u_h) = \underbrace{(\lambda_h, f - Lu_h)}_{first\ order\ error} + \underbrace{(\lambda - \lambda_h, f - Lu_h)}_{second\ order\ error}. \tag{1.6}$$

The output value of the goal function J can be conveniently corrected:

$$J_{corr} = J + (\lambda_h, f - Lu_h). \tag{1.7}$$

The corrected value will converge, as the grid spacing tends to zero, with double the order of convergence of the solution error. Pierce and Giles provide results in [13] for the case of the 1D and 2D Poisson equation, both for finite differences and for finite volumes discretizations.
Venditti and Darmofal proposed an extension of this method to non linear functions with several applications for finite volume schemes, respectively quasi-one dimensional flows [15], inviscid flows [16] and viscous flows [17]. The Venditti and Darmofal method requires the definition of two meshes: a coarse, affordable mesh, that will be referred to with the subscript H and a fine one, which is too expensive for the solution of the problem, for which the subscript h will be used. The integral quantity of interest J, which is a function of the field U, can be written using Taylor expansion as:

$$J_h(U_h) \approx J_h(U_h^H) + \frac{\partial J_h}{\partial U_h}\Big|_{U_h^H}(U_h - U_h^H), \tag{1.8}$$

in which $J_h(U_h^H)$ represents the value of J obtained by interpolating on the fine grid the field U_H computed on the coarse grid. The finite volume equations can be written in terms of residuals of the variables as:

$$R_h(U_h) = 0 = R_h(U_h^H) + \left[\frac{\partial R_h}{\partial U_h}\Big|_{U_h^H}\right](U_h - U_h^H), \tag{1.9}$$

and substituting for the term $(U_h - U_h^H)$ in equation (1.8):

$$J_h(U_h) \approx J_h(U_h^H) - \frac{\partial J_h}{\partial U_h}\Big|_{U_h^H}\left[\frac{\partial R_h}{\partial U_h}\Big|_{U_h^H}\right]^{-1} R_h(U_h^H). \tag{1.10}$$

This equation can be rewritten in terms of the adjoint vector λ:

$$J_h(U_h) \approx J_h(U_h^H) - \lambda_h|_{U_h^H} R_h(U_h^H), \tag{1.11}$$

in which λ is the solution of the adjoint problem:

$$\left[\frac{\partial R_h}{\partial U_h}\Big|_{U_h^H}\right]^T \lambda_h|_{U_h^H} = \left[\frac{\partial J_h}{\partial U_h}\Big|_{U_h^H}\right]^T. \tag{1.12}$$

Equation (1.11) is similar to (1.5) and the information is exactly the same: the error is a weighted sum of residuals and the adjoint vector components represent the weights. However the solution of λ on the fine grid is too expensive. A more useful expression can be found by solving the adjoint problem (1.12) on the coarse grid and interpolating onto the fine grid:

$$J_h(U_h) \approx J_h(U_h^H) - \lambda_h^H R_h(U_h^H) - \left(\lambda_h|_{U_h^H} - \lambda_h^H\right) R_h(U_h^H). \tag{1.13}$$

The term $\lambda_h^H R_h(U_h^H)$ is called by Venditti and Darmofal *computable correction*, since it can be easily computed by solving the problem on the coarse grid. The term $\left(\lambda_h|_{U_h^H} - \lambda_h^H\right) R_h(U_h^H)$ is called *error in computable correction* and all the grid adaptive strategy outlined is based on reducing this term.
Dwight [18, 19] proposed a new method for the specific case of finite volume methods in which the flux is computed with the Jameson scheme [20]. The dissipation coefficients of the Jameson scheme, $k^{(2)}$ and $k^{(4)}$, whose values determine the level of accuracy and stability of the scheme, are used to obtain an estimation of the error in the computation of the goal function J:

$$k^{(2)} \frac{dJ}{dk^{(2)}} + k^{(4)} \frac{dJ}{dk^{(4)}}, \tag{1.14}$$

and the corrected value of the goal function becomes:

$$J - k^{(2)} \frac{dJ}{dk^{(2)}} - k^{(4)} \frac{dJ}{dk^{(4)}}. \tag{1.15}$$

The grid adaptation strategy is based on a local indicator that is a measure of local artificial dissipation. To this purpose, the dissipation coefficients are defined independently on each cell and the local indicator for mesh adaptation is:

$$k_{(i)}^{(2)} \frac{dJ}{dk_{(i)}^{(2)}} + k_{(i)}^{(4)} \frac{dJ}{dk_{(i)}^{(4)}}, \tag{1.16}$$

This quantity is a measure of the local sensitivity of the goal function J to the coefficients of artifical dissipation. On a converged solution, the numerical dissipation should vanish and the same should happen for the sensitivity of the goal function with respect to the coefficients. The higher the value of the spurious sensitivity, the higher the local error. The adjoint method is used to compute the derivatives $\frac{dJ}{dk_i^{(2)}}$ and $\frac{dJ}{dk_i^{(4)}}$ in an efficient way. Considering the state equation $R(w,k) = 0$, the Lagrangian quantity L can be defined as:

$$L(w,k,\lambda) = J(w) + \lambda^T R(w,k), \tag{1.17}$$

and its derivative with respect to the dissipation coefficient k is equal to the derivative of the goal function J with respect to k:

$$\frac{dL}{dk} = \frac{dJ}{dk} = \left(\frac{\partial J}{\partial w} + \lambda^T \frac{\partial R}{\partial w}\right) \frac{dw}{dk} + \lambda^T \frac{\partial R}{\partial k}, \tag{1.18}$$

and defining the adjoint vector λ through the relation:

$$\frac{\partial R^T}{\partial w}\lambda = -\frac{\partial J^T}{\partial w}, \tag{1.19}$$

the derivative of the goal function with respect to k can be rewritten as:

$$\frac{dJ}{dk} = \lambda^T \frac{\partial R}{\partial k}. \tag{1.20}$$

The local indicator proposed by Peter et al. [6, 7] is based on the derivative of the goal function with respect to mesh nodes coordinates $\frac{dJ}{dX}$. This quantity is widely used in the context of sensitivity analysis [21] for shape optimization to compute the gradient of the goal function with respect to design parameters with the lowest memory effort, as proposed in [22]. The reason why $\frac{dJ}{dX}$ is also used for mesh adaptation is that it is an indication of how the goal function is sensitive to a small change in mesh coordinates. Relatively large values of $\frac{dJ}{dX}$ at grid location X, compared to other zones of the mesh, suggest that the goal function is particularly sensitive to the mesh in X and that refinement is necessary. The adjoint method is used to compute the quantity:

$$\frac{dJ}{dX} = \frac{\partial J}{\partial X} + \lambda^T \frac{\partial R}{\partial X}, \tag{1.21}$$

after solving the adjoint equation (1.19).
The quantity computed through formula (1.21) is not directly used, since it will present components that are orthogonal to the walls, e.g. orthogonal to the airfoil. These components are not suitable for mesh optimization while they are very important in the case of shape optimization. A projection of the $\frac{dJ}{dX}$ vector field is then necessary along the solid walls contours, and the field $P\left(\frac{dJ}{dX}\right)$ is obtained.
The results obtained by Peter et al. are very promising, even if the analysis has been carried out on structured meshes only. They tested the behaviour of two local indicators:

$$\eta = \left|\left|P\left(\frac{dJ}{dX}\right)\right|\right|,$$

$$\theta = \left|\left|P\left(\frac{dJ}{dX}\right)\right|\right| \cdot r,$$

the first one being the norm of $P\left(\frac{dJ}{dX}\right)$, while the second one is multiplied by a characteristic length r, equal to half the radius of the inscribed circle at each node. In the latter, the influence of the mesh size is considered. This is because it might not be possible to identify an admissible node displacement (considering the neighbouring nodes) that affects the value of J considerably when the mesh is too fine. In other words, the mesh adaptation capability is larger when the grid is coarse, even if the value of the first indicator is the same and this larger capability is taken into account in the second indicator through the characteristic length r. Peter et al. proved that the average value of the second indicator considered decreases after local mesh adaptation, while this is not always true for the average value of the first indicator. Moreover, a heuristic grid adaptation criterion has been developed, called *line addition method* [7]. This method, based on the addition of lines of nodes in areas with large values of the indicator, has been applied and shows good results for the adapted grids, on which the value of the goal function gets significantly closer to the extrapolated correct value.
The work from Peter et al. is the basis of the research that has been carried out at ONERA. The topic has been the investigation of the behaviour of the $\frac{dJ}{dX}$ local indicator for goal oriented mesh adaptation, with application to unstructured grids. The extension to unstructured grids will constitute a significant advantage in terms of adaptation capability, since the restriction to preserve a structured mesh will not be present. The topic of grid adaptation on unstructured meshes is addressed in details in [23, 24]. A complete description and comparison of a-posteriori error estimation methods can be found in [25].

CHAPTER 2
GRADIENT COMPUTATION METHODS

2.1 The *elsA Opt* module

The framework of the technical work carried out, has been the CFD software *elsA* [26], developed by ONERA. One of the many functionalities of the code *elsA* is the *Opt* module for the computation of gradients of functional outputs such as lift and drag. The capability to compute gradients is fundamental for a complete CFD chain in order to perform sensitivity analysis, [21], and aerodynamic shape optimization. In this chapter, a summary of the gradient computation methods implemented in the *elsA Opt* module will be given. However, since the main topic of the present thesis is goal oriented mesh optimization, it is important to clarify why the *Opt* module has been used. The main reason for using the *Opt* module in the present work, is that the discrete adjoint method for gradient computation, allows the user to compute derivatives of the function J with respect to a large numbers of design parameters and also with respect to the mesh nodes coordinates, $\frac{dJ}{dX}$, at an acceptable cost. As already discussed in chapter 1, this vector field is mainly used in sensitivity analysis, in order to avoid the storage of the mesh sensitivity, as explained in [22]. However, in the present thesis, the vector field $\frac{dJ}{dX}$ will be used only as a local indicator for mesh optimization. Moreover, the *Opt* module has been very important in the framework of the implementation for debugging purposes. As a matter of fact, it would have been impossible, or at least extremely difficult and time consuming, to verify the accuracy of the code for the computation of $\frac{dJ}{dX}$ without the possibility to compare with the other gradient computation methods described in the following.
The most common methods for the computation of function gradients with respect to design parameters are:

- Finite differences.

- Direct differentiation method.
- Adjoint method.

About the last two, it is important to highlight that both a continuous and a discrete form can be obtained, depending on the order in which the linearization (for direct differentiation) or adjoint operation and the discretization of the equations are done. While for the direct differentiation method only the discrete form has been actually considered in literature, for the adjoint method both forms have been used and compared, as discussed in detail in [21], in which the pros and cons of the two approaches are listed. The conclusion is that the discrete adjoint method is preferable mainly because boundary conditions are intrinsically considered in this form and the consistency with finite differences of the actual code is ensured. In the following, only the discrete form of the direct differentiation and adjoint methods will be considered.
Not only *elsA* but also external modules are used for the computation of gradients. In particular it will be necessary to have an external module for the computation of partial derivatives of the integral quantities of interest with respect to the mesh nodes coordinates and to the flow field. In the present thesis a set of Python scripts has been developed to this scope, an extract of which is given in appendix A. More in general, dedicated software are used at ONERA to this purpose, namely FFD72 for civil aircraft application, [27], and X-OPT for turbomachinery applications.
In the following, the cell-centered flow variables will be referred to with W and the mesh nodes coordinates with X. The flow variables at the boundaries of the domain are denoted by W_b and they are functions of the flow in the cells close to the boundary and of the mesh coordinates at the boundary:

$$W_b = W_b(W, X). \tag{2.1}$$

The equations of fluid dynamics, as they are coded in *elsA*, can be written in the general form:

$$R(W, X) = 0, \tag{2.2}$$

where R is the flux balance in each cell (summation of interface fluxes). It is important to highlight that the coding of the term R in *elsA* does not include the division by the volume of the cell. This remark will be valid throughout the whole thesis when referring to the term R.
The vector of design parameters will be denoted by α and the size of this vector (number of design parameters) is n_α. The vector of goal functions will be denoted by J and the size of this vector (number of goal functions) is n_f. The number of nodes will be denoted by n_n.

2.2 Finite differences

The oldest method for gradient computation is finite differences. The only advantage of this method is that it does not require any additional code implementation, since the only quantities needed are the values of the functions of interest computed on a pair of shifted meshes ($\pm d\alpha$). The formula for gradient computation by finite differences is the following:

$$\frac{dJ_k}{d\alpha_i} = \frac{J_k(W(\alpha + d\alpha_i), X(\alpha + d\alpha_i)) - J_k(W(\alpha - d\alpha_i), X(\alpha - d\alpha_i))}{2d\alpha_i}, \quad (2.3)$$

in which $J_k(W(\alpha \pm d\alpha_i), X(\alpha \pm d\alpha_i))$ represent the values of the k-th goal function computed on the meshes shifted of the quantities $\pm d\alpha_i$ respectively. The shifted meshes are not directly available as inputs, while the mesh sensitivities are. For this reason the shifted meshes have to be computed as:

$$X(\alpha \pm d\alpha_i) = X(\alpha) \pm \frac{dX}{d\alpha_i} \cdot |d\alpha_i|. \quad (2.4)$$

The finite difference method requires $2n_\alpha$ flow computations for second order accuracy, which is in general unacceptable, given the large number of parameters involved in aerodynamic design. Moreover, determining the appropriate value for $|d\alpha_i|$ is not always trivial. Too small values may lead to large rounding errors, while large values would deteriorate the accuracy of the gradient computed through formula (2.3), [21].

2.3 Discrete direct differentiation method

The discrete direct differentiation method aims at computing the gradient of the goal function by directly differentiating the discrete equations of the scheme, (2.2). The formula that can be derived by derivating with respect to the i-th component of the design parameters vector is:

$$\frac{\partial R}{\partial W}\frac{dW}{d\alpha_i} + \frac{\partial R}{\partial X}\frac{dX}{d\alpha_i} = 0 \qquad i \in [1, n_\alpha]. \quad (2.5)$$

The sensitivity of the k-th goal function J_k with respect to α_i can be expressed as:

$$\frac{dJ_k}{d\alpha_i} = \frac{\partial J_k}{\partial X}\frac{dX}{d\alpha_i} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dX}\frac{dX}{d\alpha_i} + (\frac{\partial J_k}{\partial W} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dW})\frac{dW}{d\alpha_i}. \quad (2.6)$$

Substituting in (2.6) the $\frac{dW}{d\alpha_i}$ vector obtained from (2.5), the sensitivity can be rewritten as:

$$\frac{dJ_k}{d\alpha_i} = \frac{\partial J_k}{\partial X}\frac{dX}{d\alpha_i} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dX}\frac{dX}{d\alpha_i} - (\frac{\partial J_k}{\partial W} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dW})\frac{\partial R}{\partial W}^{-1}\frac{\partial R}{\partial X}\frac{dX}{d\alpha_i}. \quad (2.7)$$

2.4 Discrete adjoint method

2.4.1 Parameter mode

There are different ways of obtaining the adjoint system of equations. The simplest one is to transpose equation (2.7):

$$\frac{dJ_k}{d\alpha_i}^T = \left(\frac{\partial J_k}{\partial X}\frac{dX}{d\alpha_i}\right)^T + \left(\frac{\partial J_k}{\partial W_b}\frac{dW_b}{dX}\frac{dX}{d\alpha_i}\right)^T - \left(\frac{\partial R}{\partial X}\frac{dX}{d\alpha_i}\right)^T \left(\frac{\partial R}{\partial W}\right)^{-T} \left(\frac{\partial J_k}{\partial W} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dW}\right)^T. \tag{2.8}$$

The definition of the adjoint vector λ_k is given by the following system of equations:

$$\left(\frac{\partial R}{\partial W}\right)^T \lambda_k = -\left(\frac{\partial J_k}{\partial W_b}\frac{dW_b}{dW} + \frac{\partial J_k}{\partial W}\right)^T \qquad k \in [1, n_f], \tag{2.9}$$

and the goal function sensitivity can be rewritten as:

$$\frac{dJ_k}{d\alpha_i} = \frac{\partial J_k}{\partial X}\frac{dX}{d\alpha_i} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dX}\frac{dX}{d\alpha_i} + \lambda_k^T \frac{\partial R}{\partial X}\frac{dX}{d\alpha_i}. \tag{2.10}$$

A different way of obtaining the discrete adjoint equation is to add a null term to equation (2.6), which is the product of the adjoint vector by (2.5):

$$\frac{dJ_k}{d\alpha_i} = \frac{\partial J_k}{\partial X}\frac{dX}{d\alpha_i} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dX}\frac{dX}{d\alpha_i} + \left(\frac{\partial J_k}{\partial W} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dW}\right)\frac{dW}{d\alpha_i} + \lambda_k^T\left(\frac{\partial R}{\partial W}\frac{dW}{d\alpha_i} + \frac{\partial R}{\partial X}\frac{dX}{d\alpha_i}\right), \tag{2.11}$$

that can be rewritten as:

$$\frac{dJ_k}{d\alpha_i} = \frac{\partial J_k}{\partial X}\frac{dX}{d\alpha_i} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dX}\frac{dX}{d\alpha_i} + \left(\frac{\partial J_k}{\partial W} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dW} + \lambda_k^T\frac{\partial R}{\partial W}\frac{dW}{d\alpha_i}\right)\frac{dW}{d\alpha_i} + \lambda_k^T\left(\frac{\partial R}{\partial X}\frac{dX}{d\alpha_i}\right). \tag{2.12}$$

If the adjoint vector is defined as in equation (2.9) the term multiplying the flow sensitivity cancels out and the goal function sensitivity is again as in (2.10).
The adjoint method requires the solution of n_f systems of equations of the type (2.9). This means that there is one adjoint vector per goal function, and it makes sense to talk about adjoint vector related, for example, to lift or drag. This also implies that every error estimation and grid adaptation strategy based on the adjoint method is function specific.
In the case of direct differentiation instead, it will be necessary to solve n_α systems of equations of type (2.5), one per each design parameter α_i. Since in general in aerodynamic design problems $n_\alpha >> n_f$, the adjoint method is more convenient than the direct differentiation.

2.4.2 Mesh mode

The discrete adjoint method described in the previous section requires the storage of the mesh sensitivity $\frac{dX}{d\alpha}$, that is a large term of size $3n_n \times n_\alpha$. An interesting alternative is to compute instead only the total derivative of the goal function with respect to the mesh nodes coordinates $\frac{dJ}{dX}$:

$$\frac{dJ_k}{dX} = \frac{\partial J_k}{\partial X} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dX} + \lambda_k^T \frac{\partial R}{\partial X}. \tag{2.13}$$

The idea is to store $\frac{dX}{d\alpha}$ on an another computer with higher memory resources and to compute the product $\frac{dJ}{dX}\frac{dX}{d\alpha}$ afterwards.
Another difference is that, in the mesh mode approach, the term $\frac{\partial R}{\partial X}$ has to be directly computed, while in the standard adjoint implementation described in the previous section the term $\frac{\partial R}{\partial X}\frac{dX}{d\alpha}$ is computed by finite differences:

$$\frac{\partial R}{\partial X}\frac{dX}{d\alpha_i} \approx \frac{R\left[X\left(\alpha + d\alpha_i\right), W(\alpha)\right] - R\left[X\left(\alpha - d\alpha_i\right), W(\alpha)\right]}{2d\alpha_i}. \tag{2.14}$$

It is worth noticing that the same idea cannot be applied to the direct differentiation method. The corresponding expression would be:

$$\frac{\partial R}{\partial W}\frac{dW}{dX} + \frac{\partial R}{\partial X} = 0, \tag{2.15}$$

and the term $\frac{dW}{dX}$ is in general too large to be stored.

2.5 Solution of the linear systems

Both in the case of the direct differentiation method and in the case of the adjoint method the solution of linear systems of equations involving the Jacobian matrix $\frac{\partial R}{\partial W}$ is required, respectively equations (2.5) and (2.9). The Jacobian $\frac{\partial R}{\partial W}$ is a large sparse matrix, so the direct inversion is not a suitable approach and some iterative procedure is necessary. In *elsA*, a Newton relaxation method is used.
In the case of direct differentiation, the Newton method can be written as:

$$\frac{\partial R}{\partial W}^{(APP)}\left(\left(\frac{dW}{d\alpha}\right)^{(l+1)} - \left(\frac{dW}{d\alpha}\right)^{(l)}\right) = -\left(\frac{\partial R}{\partial W}^{(EXA)}\frac{dW}{d\alpha}^{(l)} + \left(\frac{\partial R}{\partial X}\frac{dX}{d\alpha}\right)\right), \tag{2.16}$$

and for the adjoint method:

$$\frac{\partial R}{\partial W}^{T(APP)}\left(\lambda^{(l+1)} - \lambda^{(l)}\right) = -\left(\frac{\partial R}{\partial W}^{T(EXA)}\lambda^{(l)} + \left(\frac{\partial J}{\partial W_b}\frac{dW_b}{dW} + \frac{\partial J}{\partial W}\right)^T\right).$$

(2.17)

The matrix $\frac{\partial R}{\partial W}^{(APP)}$ is an approximate Jacobian, while $\frac{\partial R}{\partial W}^{(EXA)}$ is the exact one, that was simply noted $\frac{\partial R}{\partial W}$ in the previous sections. The subscript l refers to the iteration index.

2.6 Recursive projection method

Equations (2.16) and (2.17) can be rewritten in a fixed point iteration form as:

$$\lambda^{(l+1)} = F\lambda^{(l)} + y. \tag{2.18}$$

For example considering the adjoint method:

$$F = \left[\left(\frac{\partial R}{\partial W}^{T(APP)} \right)^{-1} \left(\frac{\partial R}{\partial W}^{T(APP)} - \frac{\partial R}{\partial W}^{T(EXA)} \right) \right],$$

$$y = - \left[\left(\frac{\partial R}{\partial W}^{T(APP)} \right)^{-1} \left(\frac{\partial J}{\partial W_b} \frac{dW_b}{dW} + \frac{\partial J}{\partial W} \right)^{T} \right],$$

in which the matrix F is called iteration matrix.
The convergence of method (2.18) depends on the spectral radius of the iteration matrix. In particular the method is convergent if the spectral radius of matrix F is lower than 1:

$$\rho(F) < 1.$$

The recursive projection method of Shroff and Keller described in detail in [28], is a stabilization procedure that aims at eliminating from the solution the unstable eigenmodes of the matrix F. The implementation of this method, that was until now present for structured grids only, in the *elsA* code, has been necessary for the practical solution of systems of equations (2.5) and (2.9) in the present thesis, since convergence could not be obtained by Newton relaxation.

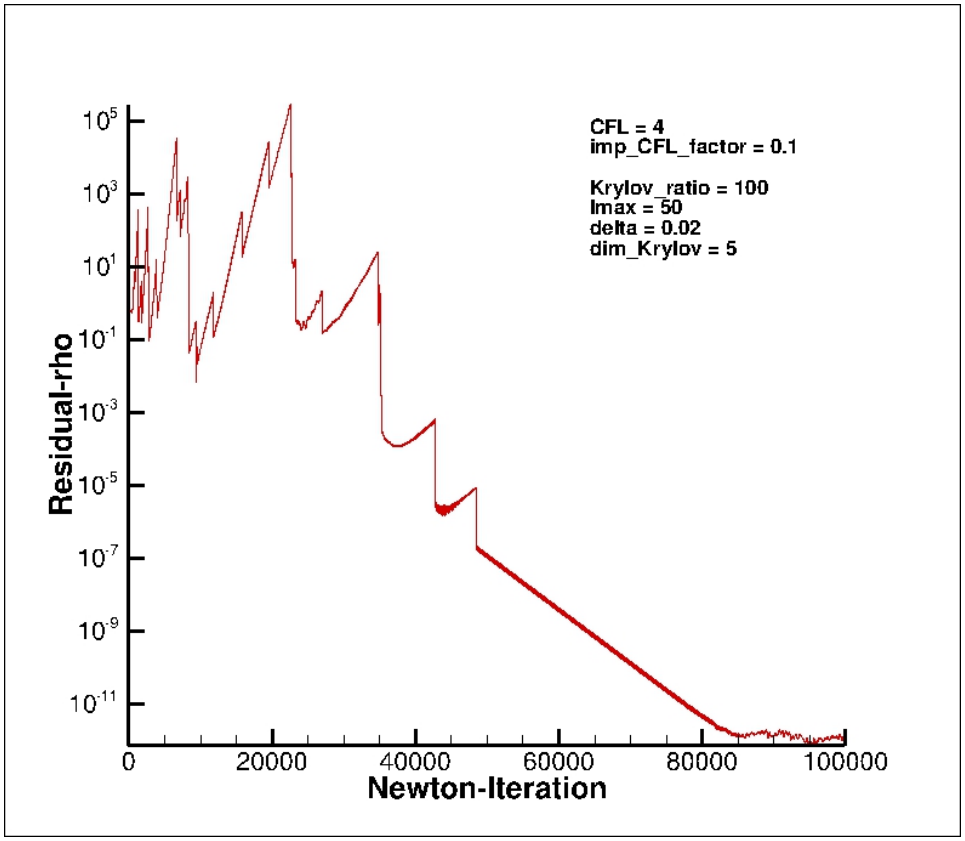


Figure 2.1: Convergence of adjoint system of equations for computation of drag derivatives using RPM (NACA0012 - 4841 nodes triangular mesh).

CHAPTER 3

CODE DEVELOPMENT

3.1 Framework of the implementation

The formulas presented in chapter 2, and in particular the discrete equations for fluid dynamics (2.2), have been written until now in a very general form. The only restriction that has been done is related to the code *elsA*, that is based on the finite volume method. In this section, the topic will be the characterization of the equations presented in chapter 2 to a more specific case, that has been the framework of the code implementation.
Only unstructured meshes have been considered in the implementation of the code, as well as in the code verification and in the mesh adaptation procedure described in chapter 4 and 5 respectively. The advantage of using unstructured meshes in the context of mesh adaptation is significant, since the possibility to add or remove arbitrary nodes is much larger, the only restriction being the capability of the code used to deal with strongly anisotropic meshes and with the presence of small cells next to large ones.
In *elsA*, at every level of coding, the connectivity information of an unstructured mesh is available in terms of left and right cell at each interface. This is an important remark, that explains why all the loops that will be described in the following sections, relative to the code implemented, have been carried out on the interfaces of the cells and never on the cells.
The normal vector at each interface is always oriented from left to right as illustrated in figure 3.1. It must be noted that the terms left and right do not have a topological meaning in this case, but they're just labels assigned to the two adjacent cells at each interface, so that the orientation of the normal vector is always properly defined.
The set of equations that will be considered in the following are the Euler equations for inviscid flows. In integral conservative form, on a fixed volume

V surrounded by surface S, the equations can be written as:

$$\frac{\partial}{\partial t}\int_V U\, dV + \int_S \vec{F}\cdot\vec{n}\, dS = 0, \tag{3.1}$$

where the vector of conservative variables U and the flux $\vec{F}\cdot\vec{n}$ are respectively equal to :

$$U = \begin{bmatrix} \rho \\ \rho\vec{u} \\ \rho E \end{bmatrix} \qquad \vec{F}\cdot\vec{n} = \begin{bmatrix} \rho\,(\vec{u}\cdot\vec{n}) \\ \rho\vec{u}\,(\vec{u}\cdot\vec{n}) + p\vec{n} \\ \rho E\,(\vec{u}\cdot\vec{n}) + p\,(\vec{u}\cdot\vec{n}) \end{bmatrix},$$

where ρ is the density, p is the pressure, $\vec{u}$ is the velocity vector and E is the total energy per unit mass.
In order to close the system of 5 equations and 6 unknowns, a thermodynamic relation is needed. In the present thesis, the assumption is that the flow considered is an ideal gas, for which the following relation holds:

$$\rho E = \frac{p}{\gamma - 1} + \frac{\rho}{2}|\vec{u}|^2, \tag{3.2}$$

where γ is the heat capacity ratio.
Moreover, it is necessary to exclude the possibility of the appearance of expansion shocks in the solution, by considering the second law of thermodynamics:

$$\frac{\partial s}{\partial t} + (\vec{u}\cdot\vec{\nabla})\, s \;\geq\; 0, \tag{3.3}$$

where s represents the entropy.
From equation (3.1) it is possible to obtain the differential conservative form of Euler equation by applying the divergence theorem:

$$\frac{\partial U}{\partial t} + \vec{\nabla}\cdot\vec{F} = 0, \tag{3.4}$$

and in linear form:

$$\frac{\partial U}{\partial t} + \frac{\partial F_x}{\partial U}\frac{\partial U}{\partial x} + \frac{\partial F_y}{\partial U}\frac{\partial U}{\partial y} + \frac{\partial F_z}{\partial U}\frac{\partial U}{\partial z} = 0. \tag{3.5}$$

The Euler equations have been discretized using a second order upwind finite volume scheme. The most common upwind techniques are the FVS, flux vector splitting, introduced by Van Leer in 1973 [29] and the FDM, flux difference method, first proposed by Godunov [30], based on the solution of a Riemann problem at each interface. A detailed explanation of the two methods is given in [31, 32]. In the present thesis the second method has been adopted, by means of the Roe approximate Riemann solver [33].
The Roe solver is based on the definition of an average Jacobian matrix $\tilde{A}$ that satisfies the following properties:

- The eigenvectors are linearly independent.
- It satisfies the condition $\tilde{A}(U_{i+1} - U_i) = F_{i+1} - F_i$.
- As $U_{i+1} \to U_i$ the average Roe matrix tends to the exact Jacobian $\tilde{A} \to \frac{\partial F}{\partial U}$.

The matrix $\tilde{A}$ is obtained by substituting in the exact Jacobian expression some average quantities between left and right interface states. Interface states will be referred to, in the following, using a $\pm$ index, with the convention that the normal vector at each interface is always directed from "-" to "+" as illustrated in figure 3.1. Since the orientation of the normal vector in *elsA* is always from the cell labelled as "left" to the cell labelled as "right", it follows that the "-" interface states are relative to left cells and "+" interface states are relative to right cells.
For the definition of the averages that are needed to build matrix $\tilde{A}$, the auxiliary variable R is introduced:

$$R = \sqrt{\frac{\rho^+}{\rho^-}},$$

and the Roe averages are:

$$\bar{\rho} = R\rho^- \qquad \bar{u} = \frac{u^+R + u^-}{1+R} \qquad \bar{v} = \frac{v^+R + v^-}{1+R}$$

$$\bar{w} = \frac{w^+R + w^-}{1+R} \qquad \bar{e} = \frac{1}{2}\sqrt{\bar{u}^2 + \bar{v}^2 + \bar{w}^2} \qquad \bar{h} = \frac{h^+R + h^-}{1+R}$$

$$\bar{c} = \sqrt{(\gamma - 1)(\bar{h} - \bar{e})}.$$

The eigenvalues of the Roe's matrix are then:

$$\bar{\lambda}_{(1)} = \bar{\lambda}_{(2)} = \bar{\lambda}_{(3)} = \bar{u}_n,$$
$$\bar{\lambda}_{(4)} = \bar{u}_n + \bar{c},$$
$$\bar{\lambda}_{(5)} = \bar{u}_n - \bar{c},$$

where $\bar{u}_n$ is the component of the average velocity vector $\vec{\bar{u}}$:

$$\vec{\bar{u}} = \begin{pmatrix} \bar{u} \\ \bar{v} \\ \bar{w} \end{pmatrix},$$

in the direction of the surface normal $\vec{n}$:

$$\bar{u}_n = \vec{\bar{u}} \cdot \vec{n}.$$

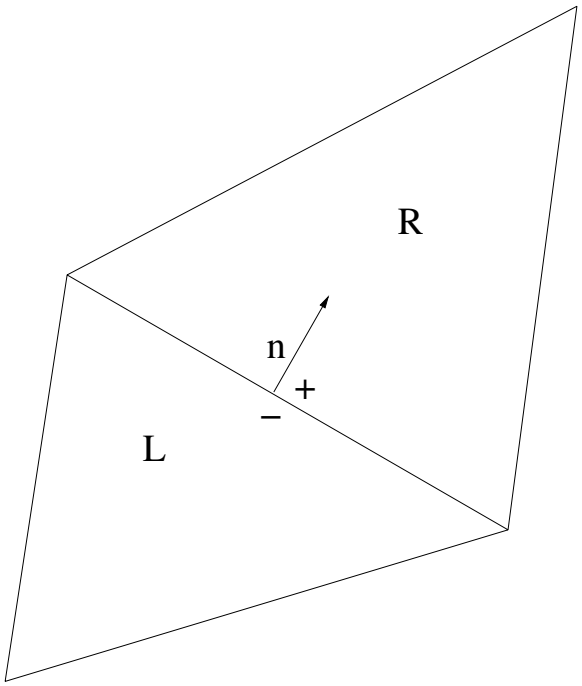


Figure 3.1: Normal vector orientation and interface states.

The Harten correction is applied in order to prevent non-physical shocks from occurring. The correction implemented in the code is based on the value ψ, which represents a percentage of the Roe matrix spectral radius:

$$|\bar{\lambda_i}| = \begin{cases} |\bar{\lambda_i}| & \text{if } |\bar{\lambda_i}| > \psi\,, \\ \frac{\bar{\lambda_i}^2+\delta^2}{2\delta} & \text{if } |\bar{\lambda_i}| < \psi\,. \end{cases} \tag{3.6}$$

The Roe flux can be written in different equivalent forms, in terms of the right eigenvectors $r_{(j)}$ of the Roe matrix $\tilde{A}$ and the corresponding eigenvalues λ_j (wave speeds) and characteristic strengths Γ_j, [34]:

$$F^{ROE} = \left(\vec{F^-} \cdot \vec{n} + \sum_j^{(-)} \lambda_{(j)} \Gamma_{(j)} r_{(j)} \right) S, \tag{3.7}$$

$$F^{ROE} = \left(\vec{F^+} \cdot \vec{n} - \sum_j^{(+)} \lambda_{(j)} \Gamma_{(j)} r_{(j)} \right) S, \tag{3.8}$$

$$F^{ROE} = \frac{1}{2} \left[(\vec{F^+} + \vec{F^-}) \cdot \vec{n} - \sum_j |\lambda_{(j)}| \Gamma_{(j)} r_{(j)} \right] S, \tag{3.9}$$

in which $\sum^{(-)}$ and $\sum^{(+)}$ indicate summation over negative and positive wave speeds respectively.
In the code *elsA*, the fluxes are always defined in terms of primitive variables. For this reason, in the following, all the interface variables, directly involved in the computation of the fluxes, will be expressed in primitive variables denoted with P.

3.1.1 MUSCL 2nd order scheme

The Roe flux scheme considered has been extended to 2nd order using MUSCL extrapolation formula, [35, 36, 37, 38, 39]. It consists in computing left and right interface states by means of a limiting function. In the present thesis, the Van Albada limiter,[40], has been considered, so that the $\pm$ interface states can be written, in primitive variables, as:

$$P^{-} = P_l + \frac{1}{2}\phi_l, \tag{3.10}$$

$$P^{+} = P_r - \frac{1}{2}\phi_r, \tag{3.11}$$

where $\phi_{l,r}$ represent the Van Albada limiters in left and right cell respectively.
The Van Albada limiter is a function of a cell centered contribution β and an upwind contribution ζ.

$$\phi_l(\beta_l, \zeta_l) = \frac{\beta_l^2\zeta_l + \zeta_l^2\beta_l}{\beta_l^2 + \zeta_l^2}, \tag{3.12}$$

$$\phi_r(\beta_r, \zeta_r) = \frac{\beta_r^2\zeta_r + \zeta_r^2\beta_r}{\beta_r^2 + \zeta_r^2}. \tag{3.13}$$

The cell centered contribution, which is the same for left and right cell, is just the difference between the value of the primitive variable in the left and right cell:

$$\beta_l = \beta_r = P_r - P_l. \tag{3.14}$$

The upwind contribution of the left cell (relative to the "-" interface state) and of the right cell (relative to "+" interface state) to the limiter, is the scalar product of the gradient times the vector $\Delta\vec{x}_c$ from the left to the right cell center.

$$\zeta_l = \frac{\partial P_l}{\partial x}\Delta x_c + \frac{\partial P_l}{\partial y}\Delta y_c + \frac{\partial P_l}{\partial z}\Delta z_c, \tag{3.15}$$

$$\zeta_r = \frac{\partial P_r}{\partial x}\Delta x_c + \frac{\partial P_r}{\partial y}\Delta y_c + \frac{\partial P_r}{\partial z}\Delta z_c. \tag{3.16}$$

The gradient of each primitive variable P (and in general the gradient of a scalar quantity) is computed in *elsA*, for unstructured meshes, with the following formula valid in each interior cell:

$$\frac{dP}{dx_i} = \frac{1}{V}\sum_j \frac{(P + P_j)}{2} S_j n_{i,j}, \tag{3.17}$$

where the index i refers to the gradient component and the index j refers to each of the interfaces of the cell considered. The quantity P_j in the formula above is the value of the variable P in the j-th neighbouring cell. The correction for border cells at the boundary interfaces consists in substituing the contribution of the average between the border cell value and the ghost cell value with the boundary interface value P_b, that depends on the type of boundary condition:

$$\left[P_b - \frac{1}{2}(P + P_{ghost})\right] S \cdot n.$$

By looking at figure 3.2, that represents a block of a simple 2D unstructured triangular mesh, we see that the interface state " - "will depend on the values in cells L, R, J1 and J2, while interface state " + " will depend on the values in L, R, H1 and H2. More in general the state " - " depends on the value of P in the left cell and all its neighbours, and state " + " depends on the value of P in the right cell and neighbours. Since the flux is function of P^+ and P^- the whole block represented in figure 3.2 will be involved in the value of the flux at the highlighted interface.
On the border interfaces, left and right states are the same and are equal to the value of the primitive variables at the border:

$$P^+ = P^- = P_b.$$

3.2 Implementation of discrete gradient computation methods

3.2.1 Direct differentiation method

As already discussed in detail in the section related to gradient computation techniques, the direct differentiation method aims at computing the gradient of the goal functions J_k:

$$\frac{dJ_k}{d\alpha_i} = \frac{\partial J_k}{\partial X}\frac{dX}{d\alpha_i} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dX}\frac{dX}{d\alpha_i} + \left(\frac{\partial J_k}{\partial W} + \frac{\partial J_k}{\partial W_b}\frac{\partial W_b}{\partial W}\right)\frac{dW}{d\alpha_i}, \tag{3.18}$$

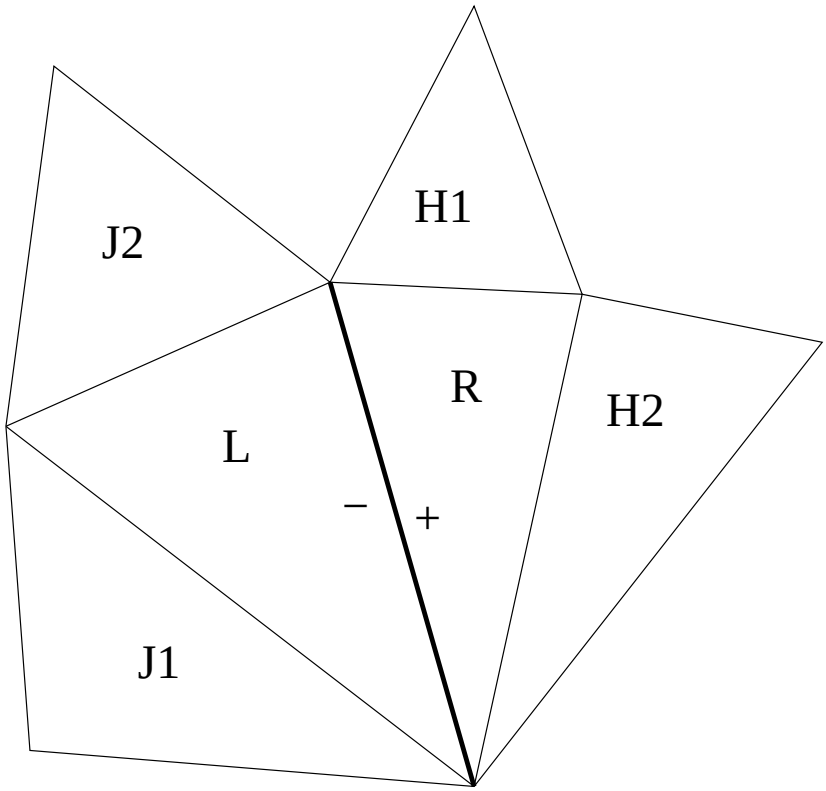


Figure 3.2: Block of cells involved in the computation of the flux Jacobian at the highlighted interface

based on the direct solution of the systems of equation:

$$\frac{\partial R}{\partial W}\frac{dW}{d\alpha_i} = -\frac{\partial R}{\partial X}\frac{dX}{d\alpha_i} \qquad i \in [1, n_\alpha]. \tag{3.19}$$

In practice the systems will be solved using a Newton relaxation method:

$$\frac{\partial R}{\partial W}^{(APP)}\left(\frac{dW}{d\alpha_i}^{(l+1)} - \frac{dW}{d\alpha_i}^{(l)}\right) = -\left(\frac{\partial R}{\partial W}^{(EXA)}\frac{dW}{d\alpha_i}^{(l)} + \frac{\partial R}{\partial X}\frac{dX}{d\alpha_i}\right). \tag{3.20}$$

In figure 3.3 a flowchart is presented summarizing the procedure followed in *elsA* for the computation of the gradient by direct differentiation. The block filled in red is the one that has been implemented, adding a new class to the rest of the already existing code.
The main class is *OptLinParam*, an istance of which is created once the user declares that he is going to use this method for gradient computation. After that, the three classes below are created. The one on the left, *OptLinGatherGradients*, is responsible for gathering the different terms of equation (3.18). All the partial derivatives of the goal function with respect to the

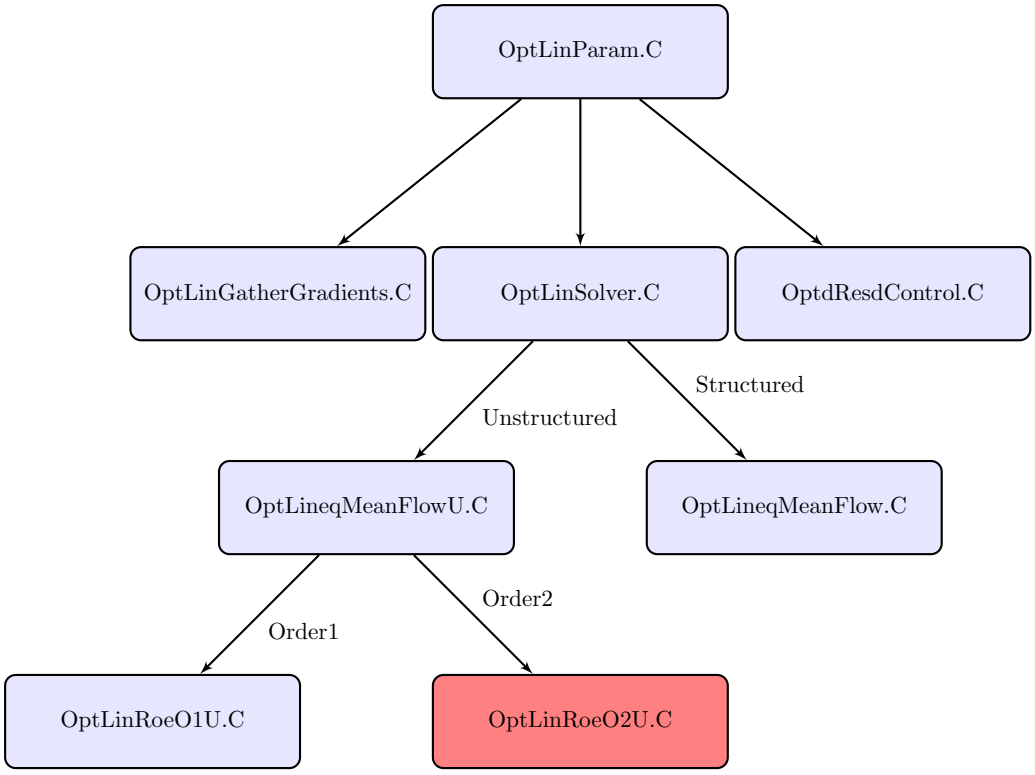


Figure 3.3: Structure of the code

flow and mesh coordinates are not computed using *elsA*, but through external modules. As already mentioned, in this thesis a set of Python scripts has been developed to this purpose, and a brief extract is reported in appendix A. The mesh sensitivity is given as input to *elsA* before starting the computation. The derivatives of the boundary values W_b with respect to the mesh $\frac{dW_b}{dX}$ and the flow $\frac{dW_b}{dW}$ are computed in a different module of *elsA* and are dependent on the type of boundary condition. In the end, the only term to be computed inside the module *Opt* of *elsA* is the flow sensitivity $\frac{dW}{d\alpha}$ through the solution of system (3.20). The class *OptLindResdControl* is responsible for the computation of the term $\frac{\partial R}{\partial X}\frac{dX}{d\alpha}$ in equation (3.20) by finite differences:

$$\frac{\partial R}{\partial X}\frac{dX}{d\alpha_i} \approx \frac{R[X(\alpha + d\alpha_i), W(\alpha)] - R[X(\alpha - d\alpha_i), W(\alpha)]}{2d\alpha_i}, \tag{3.21}$$

which means that the initial mesh is shifted according to the quantities $d\alpha$ and $-d\alpha$ and the flux balance is computed respectively on the two meshes. Since only the mesh sensitivity $\frac{dX}{d\alpha}$ is given as input to the code, while the two shifted meshes are not, their coordinates have to be computed as:

$$X(\alpha \pm d\alpha_i) = X(\alpha) \pm \frac{dX}{d\alpha_i} \cdot |d\alpha_i|. \tag{3.22}$$

The focus of the implementation has been though the *OptLinSolver* class, that manages the operations for the solution of the system (3.20). The matrix $\frac{\partial R^{(APP)}}{\partial W}$ is already coded in *elsA* and there is no need to make any adaptation for the second order flux. The term of equation (3.20) that has been coded for second order flux is instead:

$$\left(\frac{\partial R}{\partial W}^{(EXA)}\frac{dW}{d\alpha}^{(l)}\right), \tag{3.23}$$

in which the matrix $\frac{\partial R}{\partial W}^{(EXA)}$ is the exact Jacobian of the flux balance (cell centered variable) with respect to aerodynamic variables and l is the iteration index of the Newton method. This operation is carried out after the creation of the objects *OptLineqMeanFlow* or *OptLineqMeanFlowU* depending on the type of domain considered (structured or unstructured). In the present thesis only unstructured domains have been considered. The last object to be created is the flux object that is specific for the type of flux used and in this case it will be the 2nd order Roe flux scheme previously described in this chapter. At this level of the code the new class *OptLinRoeO2U* has been embedded as shown in figure 3.3.
In order to explain the implementation of the term (3.23), it is convenient to take in consideration one single row of the flux balance Jacobian matrix, corresponding to the row vector of derivatives of the flux balance in cell C

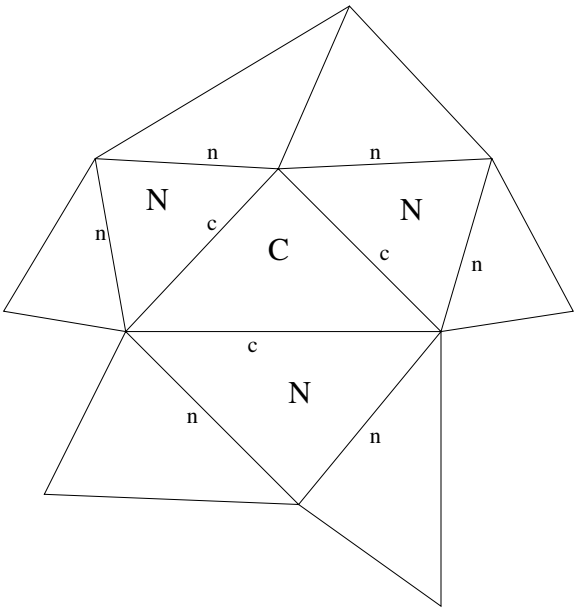


Figure 3.4: Block of cells involved in the computation of $\frac{\partial R_C}{\partial W}\frac{dW}{d\alpha}$.

represented in figure 3.4. This row vector will have 10 non null components corresponding to the indices of the cells in figure 3.4, representing the block of cells involved in the computation of $\frac{\partial R_C}{\partial W}\frac{dW}{d\alpha}$. Since the connectivity information in *elsA* is available in terms of left and right cell indices at each interface, it is convenient to implement a code based on a loop on the interfaces. To this purpose the product $\frac{\partial R_C}{\partial W}\frac{dW}{d\alpha}$ can be rewritten in terms of interface variables, namely the flux derivatives:

$$\frac{\partial R_C}{\partial W}\frac{dW}{d\alpha} = \sum_c \frac{\partial F_c}{\partial W}\frac{dW}{d\alpha}, \tag{3.24}$$

where the indices C and c refer to figure 3.4.
The implementation will consist in the computation of the Jacobian of the flux (interface variable) at each interface:

$$\left(\frac{\partial F_c}{\partial W}\frac{dW}{d\alpha}\right) = \frac{\partial F_c}{\partial P^-}\left[\sum_j \frac{\partial P^-}{\partial P_j}\frac{dP_j}{dW_j}\frac{dW_j}{d\alpha}\right] + \frac{\partial F_c}{\partial P^+}\left[\sum_h \frac{\partial P^+}{\partial P_h}\frac{dP_h}{dW_h}\frac{dW_h}{d\alpha}\right]. \tag{3.25}$$

The index j refers to the subset of cells that have an influence on the interface value P^-, that are the left cell and all its neighbouring cells. Similarly the index h refers to the subset of cells that influence the value P^+, that are the right cell and all its neighbouring cells.
This approach is absolutely general and is also valid for structured meshes.

The difference between structured and unstructured meshes is the explicit expression of the interface states, since for unstructured meshes, the gradient of the primitive variables (and in particular the value of the gradient in left and right cell) is involved in the definition of the interface variables through the Van Albada limiting function (see previous section).
The complexity of the implementation consists in the fact that, while looping on each interface, it is only possible to identify indices and quantities related to left and rigth cell and there is no further connectivity information regarding the block of surrounding cells. Considering figure 3.2 for example, and reminding the formulas (3.10) and (3.12), the derivative of the P^- interface state with respect to the cell centered variable P_{J1} can be written as:

$$\frac{\partial P^-}{\partial P_{J1}} = \frac{1}{2}\left[\frac{\partial \phi_L}{\partial \zeta_L}\frac{d\zeta_L}{dP_{J1}} + \frac{\partial \phi_L}{\partial \beta_L}\frac{d\beta_L}{dP_{J1}}\right],$$

and reminding formula (3.15) and (3.17)

$$\frac{\partial P^-}{\partial P_{J1}} = \frac{1}{2}\frac{\partial \phi_L}{\partial \zeta_L}\left[\frac{S_{J1}}{2V_L}\left(n_{(x,J1)}\Delta x_c + n_{(y,J1)}\Delta y_c + n_{(z,J1)}\Delta z_c\right)\right].$$

In this case, while looping on the interface highlighted in figure 3.2, the values S_{J1}, $n_{(x,J1)}$, $n_{(y,J1)}$, $n_{(z,J1)}$ would not be available.
The best way to overcome this obstacle is to do a two-steps implementation, by looping on the interfaces two times. During the first loop, the goal will be to store a cell centered quantity that contains the information of the contribution of the surrounding cells. Only after doing this, it will be possible to compute the exact Jacobian with a second loop on interfaces.
The terms of equation (3.25) can be rearranged after defining the partial sensitivity (only flow variables dependence) of primitive variables gradients with respect to the design parameter:

$$\frac{\partial(\nabla P)}{\partial W}\frac{dW}{d\alpha} = \frac{1}{V}\sum_j\left[\frac{1}{2}\left(\frac{dP}{dW}\frac{dW}{d\alpha} + \frac{dP_j}{dW_j}\frac{dW_j}{d\alpha}\right)S_j n_j\right],$$

that should not be confused with the total sensitivity (flow variables and mesh dependence):

$$\frac{d(\nabla P)}{d\alpha} = \frac{\partial(\nabla P)}{\partial W}\frac{dW}{d\alpha} + \frac{\partial(\nabla P)}{\partial X}\frac{dX}{d\alpha}. \tag{3.26}$$

Now formula (3.25) can be rewritten in a more general form as follows:

$$\frac{\partial F_c}{\partial W}\frac{dW}{d\alpha} = \frac{\partial F_c}{\partial P^\pm}\frac{\partial P^\pm}{\partial(\nabla P)_{L,R}}\left[\frac{\partial(\nabla P)_{L,R}}{\partial W}\frac{dW}{d\alpha}\right] + \frac{\partial F_c}{\partial P^\pm}\frac{\partial P^\pm}{\partial P_{L,R}}\frac{dP_{L,R}}{dW_{L,R}}\frac{dW_{L,R}}{d\alpha}. \tag{3.27}$$

This expression is useful because it includes only terms related to the left and right cell. Indeed the contribution of the surrounding cells is hidden inside the terms $\frac{\partial(\nabla P)}{\partial W}\frac{dW}{d\alpha}$, that will be computed during the first loop. Once this terms are stored, a second loop on the interfaces is carried out in order to finalize the computation of (3.27). The cost paid for the much easier implementation is the memory needed to store the terms $\frac{\partial(\nabla P)}{\partial W}\frac{dW}{d\alpha}$, corresponding to 15 scalar fields.

3.2.2 Discrete adjoint method (parameter mode)

The discrete adjoint method is based on the definition of the adjoint vector:

$$\frac{\partial R}{\partial W}^T \lambda_k = -\left(\frac{\partial J_k}{\partial W_b}\frac{dW_b}{dW} + \frac{\partial J_k}{\partial W}\right)^T \qquad k \in [1, n_f], \tag{3.28}$$

and, as already illustrated in chapter 1, after the solution of this systems of equation we can rewrite the goal functions sensitivities as:

$$\frac{dJ_k}{d\alpha_i} = \frac{\partial J_k}{\partial X}\frac{dX}{d\alpha_i} + \frac{\partial J_k}{\partial W_b}\frac{dW_b}{dX}\frac{dX}{d\alpha_i} + \lambda_k^T\frac{\partial R}{\partial X}\frac{dX}{d\alpha_i}. \tag{3.29}$$

Similarly to the direct differentiation method, the systems of equation (3.28) can be solved using the Newton relaxation method. In this case:

$$\frac{\partial R}{\partial W}^{T(APP)}\left(\lambda_k^{(l+1)} - \lambda_k^{(l)}\right) = -\left[\frac{\partial R}{\partial W}^{T(EXA)}\lambda_k^{(l)} + \left(\frac{\partial J_k}{\partial W_b}\frac{dW_b}{dW} + \frac{\partial J_k}{\partial W}\right)\right]. \tag{3.30}$$

In figure 3.5 the structure of the code of the adjoint method (parameter mode) is presented. The main object *OptAdjParam* is provided with two methods. The method *gather*, in which the terms of equation (3.29) are gathered. In order to do this, it is necessary to create an object of type *OptdResdControl* for the computation of $\frac{\partial R}{\partial X}\frac{dX}{d\alpha}$ by finite differences. The method *solve*, for the solution of system (3.30), that provides the adjoint vector λ. The scheme is very similar to the one presented for the direct differentiation method, with the only difference that in this case the object of type *OptAdjdFunctiondAero* is created to gather the aerodynamic partial derivatives of the goal function appearing in the right hand side of (3.30). The focus of my work has been instead the creation of the new class *OptAdjRoeO2U* for the computation of the term:

$$\frac{\partial R}{\partial W}^{T(EXA)}\lambda^{(l)} = \lambda^{T(l)}\frac{\partial R}{\partial W}^{(EXA)}. \tag{3.31}$$

In this case, in order to explain the implementation, it is convenient to consider one single column of the flux balance Jacobian matrix, representing the

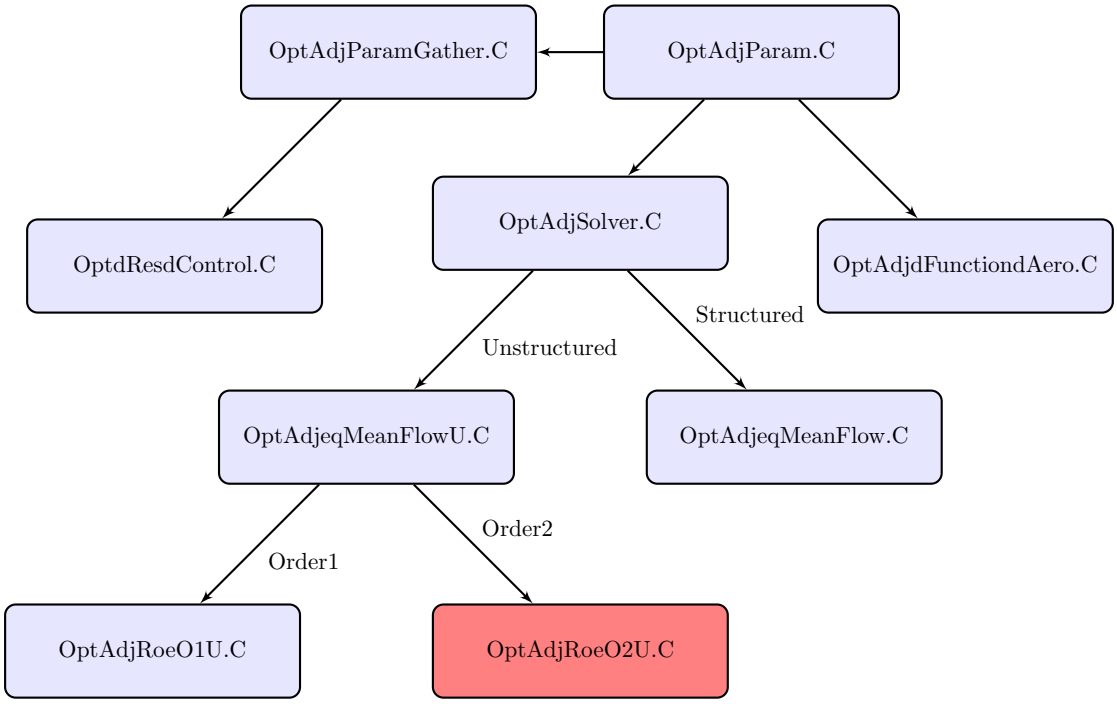


Figure 3.5: Structure of the code

column vector of derivatives of the flux balance with respect to the flow variables of cell C. This vector will have 10 non null components, corresponding to the flux balances of each cell of the block represented in figure 3.4. For this reason the product $\lambda^T \frac{\partial R}{\partial W_C}$ will involve the adjoint vector in each cell of the block. Also in this case, it is convenient to express this quantity in term of fluxes rather than flux balances, since the loop has to be carried out on interfaces where fluxes are available rather than flux balances:

$$\lambda^T \frac{\partial R}{\partial W_C} = \sum_{c,n} \Delta\lambda_{c,n} \frac{\partial F_{c,n}}{\partial W_C}, \tag{3.32}$$

where the indices c and n refer to figure 3.4 and $\Delta\lambda$ is the difference of the adjoint components of left and right cells at each interface (always left minus right, see figure 3.1). Also in this case an implementation based on one single loop on the interfaces is not possible. With two loops on the interfaces it is again possible, similarly to the case of the direct differentiation code, to store a cell centered quantity containing the information of the contribution of the furthest cells. In this case the quantity that has been stored does not have a clear physical interpretation as in the case of the gradient sensitivity for the direct differentiation method. It can be expressed as:

$$\omega_A = \sum_i \Delta\lambda_i \frac{\partial F_i}{\partial P^*} \frac{\partial P^*}{\partial (\nabla P)_A}, \tag{3.33}$$

where the superscript * refers to internal interface states of each cell, as illustrated in figure 3.6 for cell A.
After the definition of ω, equation (3.32) can be rewritten as:

$$\lambda^T \frac{\partial R}{\partial W_C} = \sum_c \left(\Delta\lambda_c \frac{\partial F_c}{\partial W_C} + \omega_N \cdot \vec{S}_c \right), \tag{3.34}$$

that can be easily computed during the second loop on interfaces. A part of Fortran code implemented for the computation of the term $\lambda^T \frac{\partial R}{\partial X}$ is given in appendix C.

3.2.3 Discrete adjoint method (mesh mode)

The last part of code implemented in the framework of the *elsA Opt* module has been the adjoint method in mesh mode, for the computation of $\frac{dJ}{dX}$, the total derivative of the goal function with respect to the mesh nodes coordinates:

$$\frac{dJ_k}{dX} = \frac{\partial J_k}{\partial X} + \frac{\partial J_k}{\partial W_b} \frac{dW_b}{dX} + \lambda_k^T \frac{\partial R}{\partial X}. \tag{3.35}$$

The structure of the code is presented in figure 3.7. The computation of the adjoint vector λ is carried out in the same subroutines used for the standard

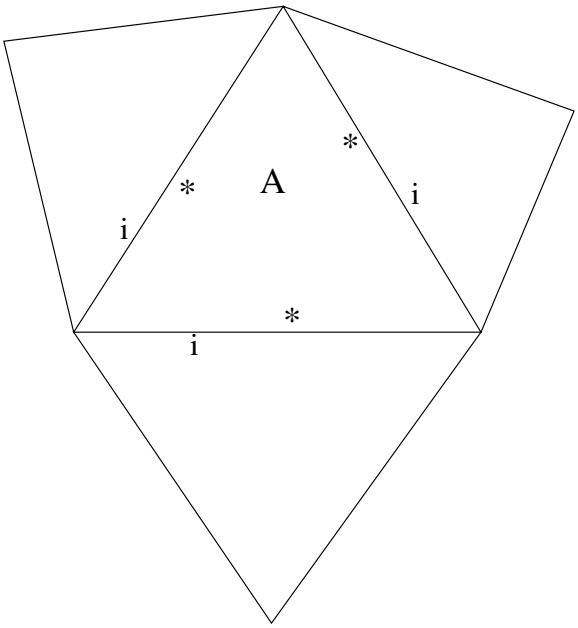


Figure 3.6: Only internal interface states marked with * are involved in the definition of ω in cell A.

adjoint method (parameter mode), described in the previous section. The object *OptAdjMeshGather* will require in this case the computation of the derivative of the flux balance with respect to mesh nodes coordinates, $\frac{\partial R}{\partial X}$, that is dependent on the order of the scheme. For this reason the new class *OptAdjMeshRoeO2U* has been created for second order Roe flux.
The Fortran subroutines called by this new class compute the partial derivative of the flux balance $\frac{\partial R}{\partial X}$ as a sum of flux derivatives on the interfaces of each cell:

$$\frac{\partial R}{\partial X} = \sum_j \frac{\partial F_j}{\partial X}. \tag{3.36}$$

Each term $\frac{\partial F_j}{\partial X}$ is a sum of different contributions: first order terms, that include the direct dependence of the flux balance on the surface vector and second order terms, that include the dependence of the interface states on ζ, the upwind contribution to the Van Albada limiter, see equations (3.15) and (3.16), that in turn depends on the mesh nodes coordinates. This is clear considering the dependence of the gradient of the primitive variables, see equation (3.17), on the volume and the surfaces of the cell, that are directly related to the coordinates of the nodes. The complete expression of $\frac{\partial F}{\partial X}$ can

be written as:

$$\frac{\partial F}{\partial X} = \underbrace{\frac{\partial F}{\partial \vec{S}}\frac{d\vec{S}}{dX}}_{1st\ order\ terms} + \underbrace{\frac{\partial F}{\partial P^{\pm}}\frac{\partial P^{\pm}}{\partial \zeta}\left[\frac{\partial \zeta}{\partial V}\frac{dV}{dX} + \frac{\partial \zeta}{\partial \vec{S}}\frac{d\vec{S}}{dX} + \frac{\partial \zeta}{\partial \Delta\vec{x_c}}\frac{d\Delta\vec{x_c}}{dX}\right]}_{2nd\ order\ terms}. \quad (3.37)$$

For cells that are adjacent to the boundary there is one more contribution, because in this case the expression of the gradient in the cell will include the boundary state W_b, that is in general dependent on the orientation of the normal vector at the boundary interface. This contribution will be:

$$\frac{\partial F}{\partial P^{\pm}}\frac{\partial P^{\pm}}{\partial \zeta}\frac{\partial \zeta}{\partial W_b}\frac{dW_b}{dX}. \quad (3.38)$$

For the computation of the second order terms of equation (3.37), a first loop on interfaces is needed in order to store an array containing the indices of the nodes and the indices of the interfaces belonging to each cell of the mesh. The second step is the computation of the elementary matrices $\frac{dV}{dX}$, $\frac{d\vec{S}}{dX}$ and $\frac{d\Delta\vec{x_c}}{dX}$. Finally, the actual computation of the terms of equation (3.37) is carried out.
Given the complexity of the computation of the second order terms of equation (3.37), the code developed for the adjoint method in mesh mode is able to handle 2D meshes only. On the other hand, the codes developed for the direct differentiation method and for the adjoint method in parameter mode are as general as possible and able to cope with 3D meshes as well.

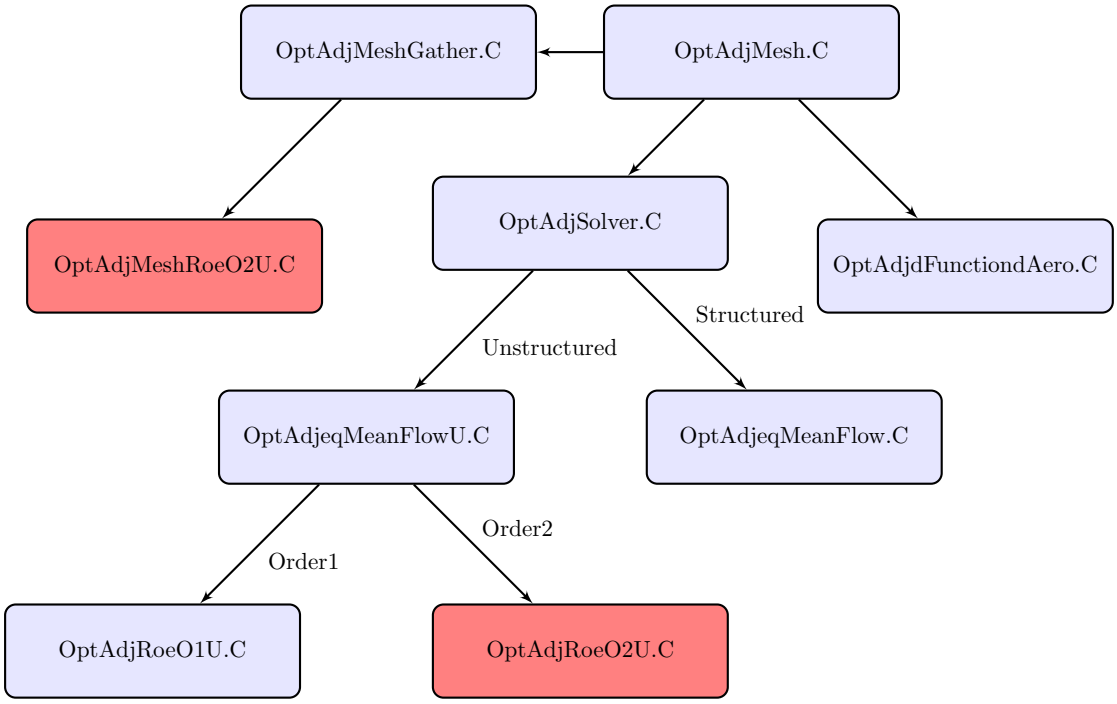


Figure 3.7: Structure of the code

CHAPTER 4

CODE VERIFICATION

4.1 Test case NACA0012

4.1.1 Mesh

The test case that has been used for the verification of the code is a NACA0012. The characteristics of the mesh are the following:

- 4787 nodes
- 9027 triangular elements
- Boundaries are located at 8 chord lengths upwind of the airfoil and 10 chord lengths downwind of the airfoil.

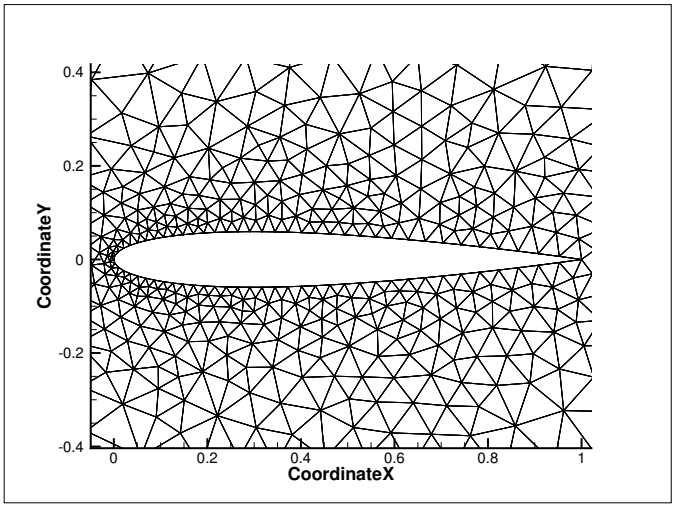


Figure 4.1: Detail of the mesh in proximity of the airfoil.

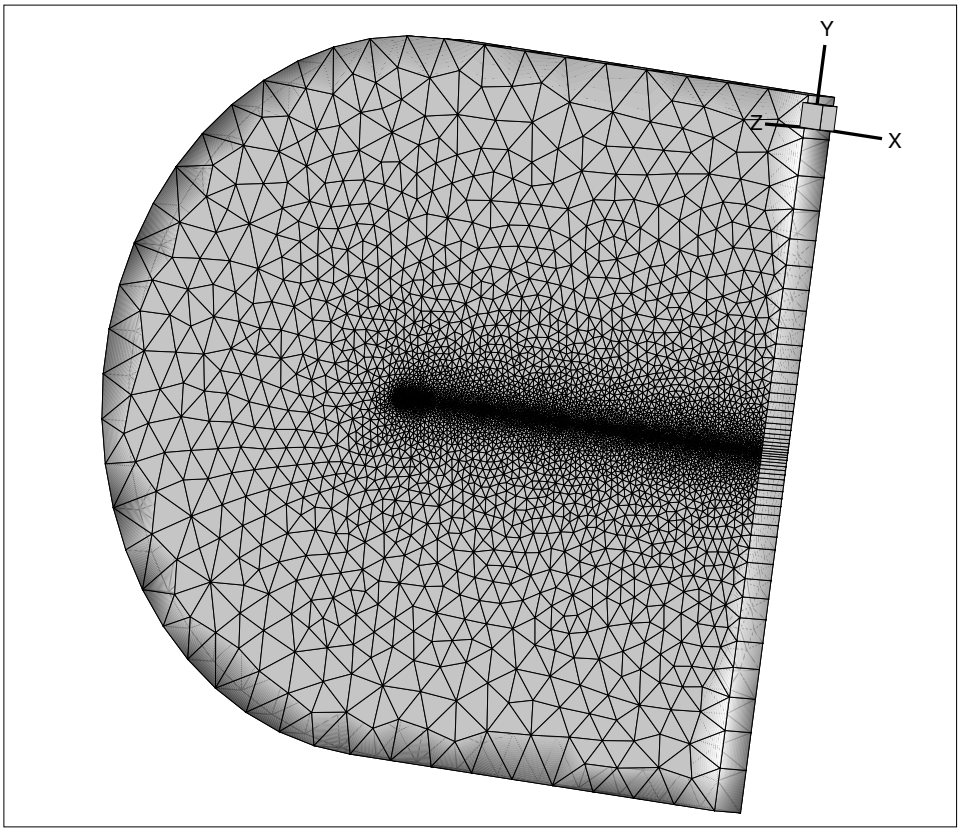
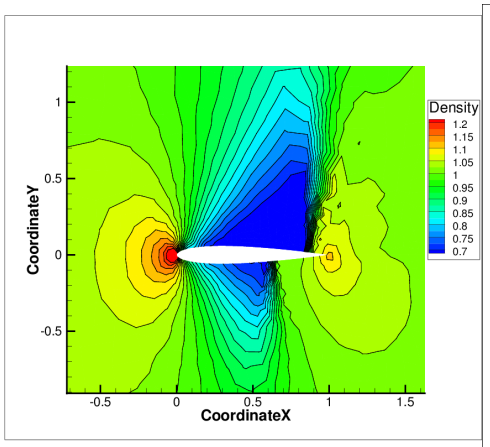


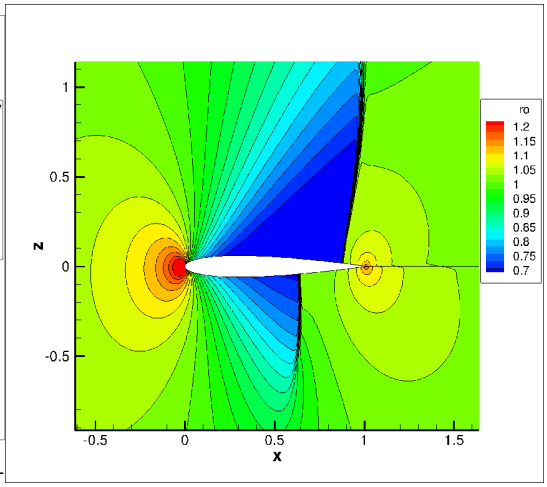
Figure 4.2: View of the mesh.

4.1.2 Flow

The flow conditions that have been considered for the verification test are $Mach = 0.85$ and $AoA = 1°$.
The steady state solution has been found with the code *elsA* using an implicit backward Euler scheme. The type of flux is the second order MUSCL extension of Roe flux described in the previous chapter.
The mesh considered is suitable for the verification of the code implemented, due to the low number of nodes. However it is not fine enough to compute in an accurate way the transonic flow considered. For this reason the flow has been previously computed and checked using a 513×513 nodes structured mesh. After this step, the steady state computation has been carried out on the unstructured mesh in order to proceed with the code verification.
The characteristics of the flow are presented in figures 4.3 to 4.4.

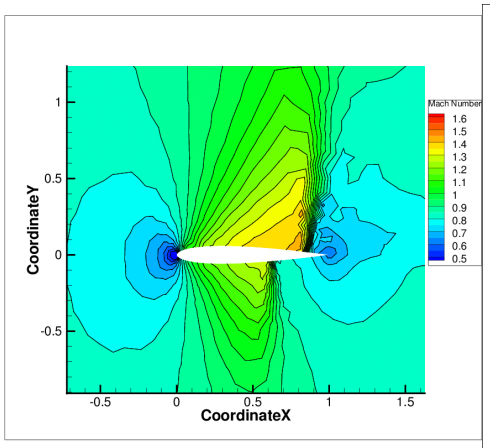


(a) Unstructured mesh for code verification (4787 nodes).

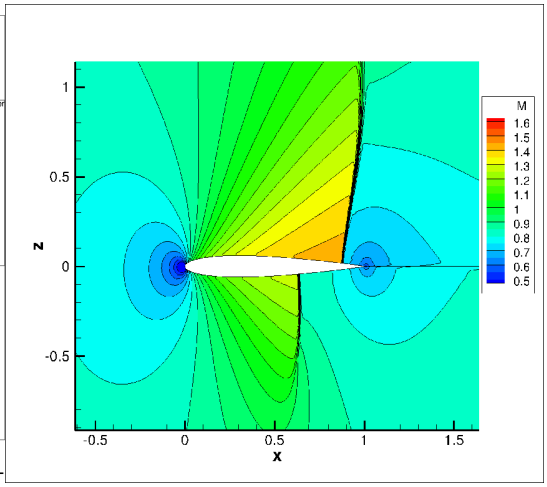


(b) Structured mesh (513×513 nodes).

Figure 4.3: Density contour plot at steady state.



(a) Unstructured mesh for code verification (4787 nodes).



(b) Structured mesh (513×513 nodes).

Figure 4.4: Mach contour plot at steady state.

4.1.3 Parameter α

The parameter α with respect to which the gradients are computed has been chosen as a simple rotation around the leading edge. In order to keep the outer boundary unaltered, the imposed transformation has to gradually vanish while approaching it, as shown in figure 4.5
Being ρ the distance to the leading edge, the transformation can be written:

$$x(\alpha) = \begin{cases} x\cos(\alpha) + y\sin(\alpha) & \text{if } \rho \leq 4\,, \\ (x\cos(\alpha) + y\sin(\alpha) - x)(\frac{1}{4}\rho^3 - \frac{15}{4}\rho^2 + \frac{1}{18}\rho) & \text{if } 4 \leq \rho \geq 6\,, \\ x & \text{if } \rho \geq 6\,. \end{cases}$$

$$y(\alpha) = \begin{cases} -x\sin(\alpha) + y\cos(\alpha) & \text{if } \rho \leq 4\,, \\ (-x\sin(\alpha) + y\cos(\alpha) - y)(\frac{1}{4}\rho^3 - \frac{15}{4}\rho^2 + \frac{1}{18}\rho) & \text{if } 4 \leq \rho \geq 6\,, \\ y & \text{if } \rho \geq 6\,. \end{cases}$$

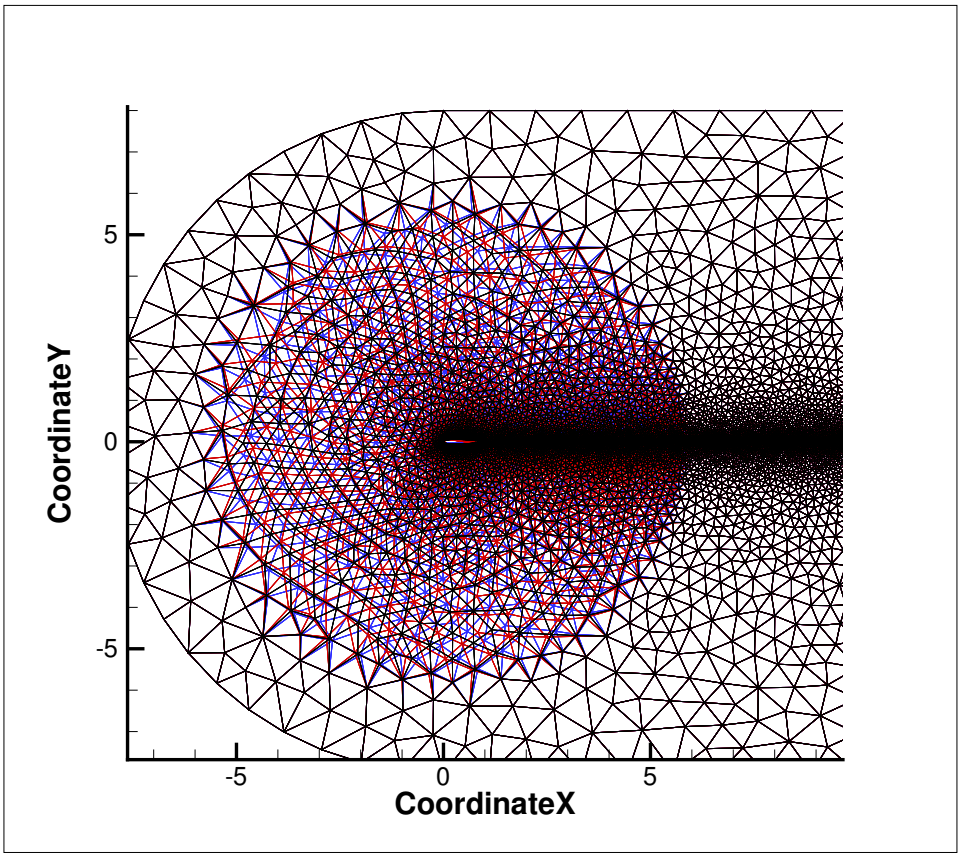


Figure 4.5: Mesh rotation vanishing in the vicinity of the boundary (for visualization purposes the value of $d\alpha$ in this picture is much larger than the $d\alpha$ used).

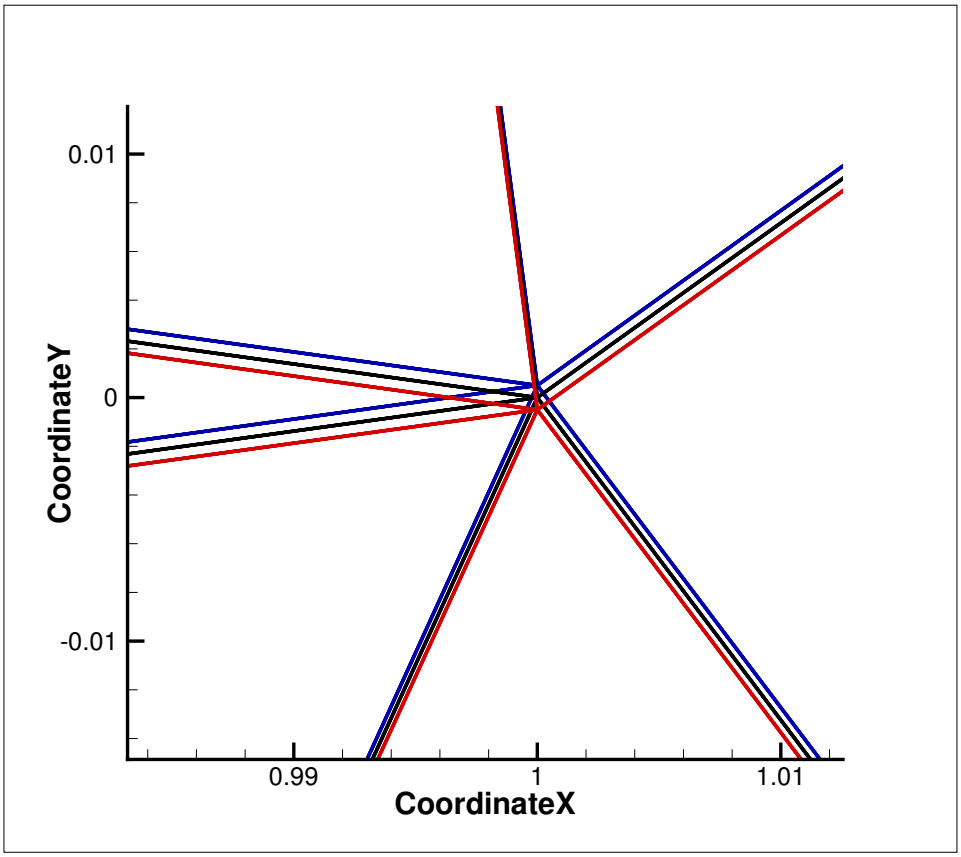


Figure 4.6: Detail of the $\pm\alpha$ shifted meshes near the trailing edge ($d\alpha = 5 \cdot 10^{-5}$).

The shifted meshes will be used to compute finite differences values and also to compute the mesh sensitivity to the parameter α, $\frac{dX}{d\alpha}$, that is one of the needed file inputs for gradient computation in *elsA*.

4.2 Verification procedure

4.2.1 Direct differentiation method

The validity and the precision of the output of the direct differentiation method has been checked by means of comparison with finite differences. Several quantities of interests have been checked including:

- Aerodynamic part of primitive variable gradients sensitivities
- Sensitivities of conservative variables
- Aerodynamic part of flux and total flux sensitivity

- Sensitivity of goal functions

As shown in chapter 3, the aerodynamic part of the flux derivative can be written as:

$$\frac{\partial F}{\partial W}\frac{dW}{d\alpha} = \frac{\partial F}{\partial P^{\pm}}\frac{\partial P^{\pm}}{\partial (\nabla P)_{L,R}}\left[\frac{\partial (\nabla P)_{L,R}}{\partial W}\frac{dW}{d\alpha}\right] + \frac{\partial F}{\partial P^{\pm}}\frac{\partial P^{\pm}}{\partial P_{L,R}}\frac{dP_{L,R}}{dW_{L,R}}\frac{dW_{L,R}}{d\alpha}. \tag{4.1}$$

The array $\frac{\partial(\nabla P)}{\partial W}\frac{dW}{d\alpha}$ is stored as a cell-centered quantity during a first loop on the interfaces of the mesh. The advantage of this type implementation, apart from those explained in the previous chapter, is that this quantity can be checked using finite differences in the code verification stage. It can be seen, in each cell, as the change of the gradient of primitive variables produced by the change of the flow due to the shifting of the mesh, while the mesh itself remains unaltered:

$$\frac{\partial(\nabla P)}{\partial W}\frac{dW}{d\alpha} = \frac{\nabla P\left[X(\alpha), W(\alpha + d\alpha)\right] - \nabla P\left[X(\alpha), W(\alpha - d\alpha)\right]}{2d\alpha}. \tag{4.2}$$

The expression (4.2) indicates that, in order to compute this quantity by finite differences, it will be sufficient to compute the gradient of the solution on the nominal mesh $X(\alpha)$, considering the steady state solutions obtained on the shifted meshes. On the other hand, from the gradient computation routine, it will be possible to have this quantity directly, that is computed and stored during the first loop on interfaces, as explained in chapter 3.
The comparison is plotted in figures 4.7 and 4.8 for two different gradient components.
The second step of the verification takes place once the system of equations:

$$\frac{\partial R}{\partial W}^{(APP)}\left(\frac{dW}{d\alpha}^{(l+1)} - \frac{dW}{d\alpha}^{(l)}\right) = -\left(\frac{\partial R}{\partial W}^{(EXA)}\frac{dW}{d\alpha}^{(l)} + \frac{\partial R}{\partial X}\frac{dX}{d\alpha}\right), \tag{4.3}$$

is definitively solved. At this point, it will be possible to compare the mesh sensitivity computed in the direct differentiation routine with the finite differences obtained from the shifted meshes:

$$\frac{dW}{d\alpha} = \frac{W(\alpha + d\alpha) - W(\alpha - d\alpha)}{2d\alpha}. \tag{4.4}$$

The results are shown in figures 4.9 and 4.10.
The last check has been carried out on flux balance derivatives. In particular, the term $\frac{\partial R}{\partial W}\frac{dW}{d\alpha}$ computed in the direct differentiation code has been compared also in this case with finite differences. The right quantity to compare with is the flux balance computed on the nominal mesh, but taking the flow solutions computed on the shifted meshes:

$$\frac{\partial R}{\partial W}\frac{dW}{d\alpha} = \frac{R\left[X(\alpha), W(\alpha + d\alpha)\right] - R\left[X(\alpha), W(\alpha - d\alpha)\right]}{2d\alpha}. \tag{4.5}$$

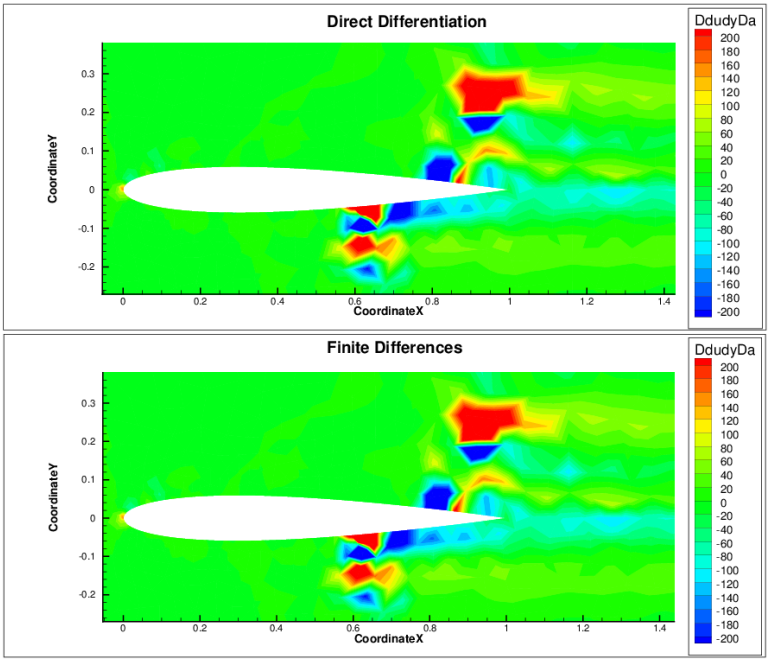


Figure 4.7: $\frac{\partial\left(\frac{\partial u}{\partial y}\right)}{\partial W}\frac{dW}{d\alpha}$ computed with DDM and FD respectively .

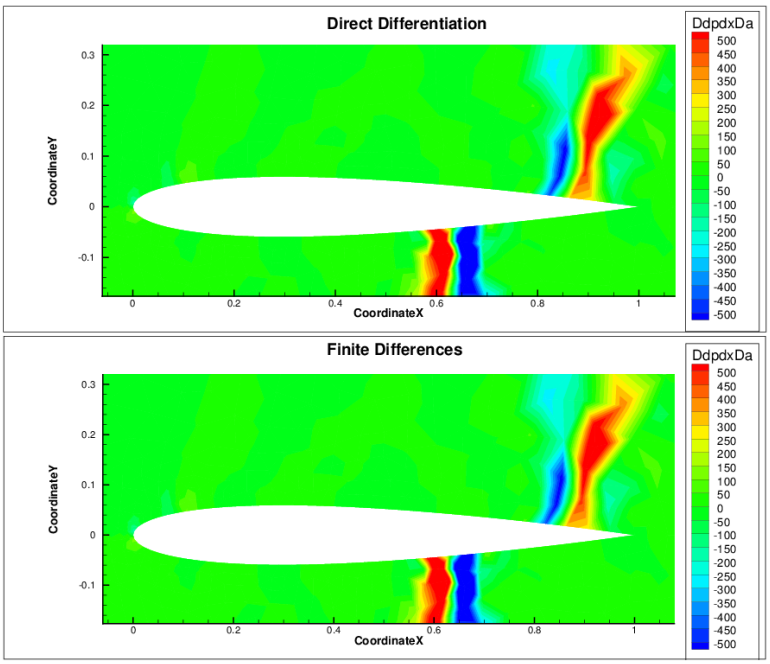


Figure 4.8: $\frac{\partial\left(\frac{\partial p}{\partial x}\right)}{\partial W}\frac{dW}{d\alpha}$ computed with DDM and FD respectively.

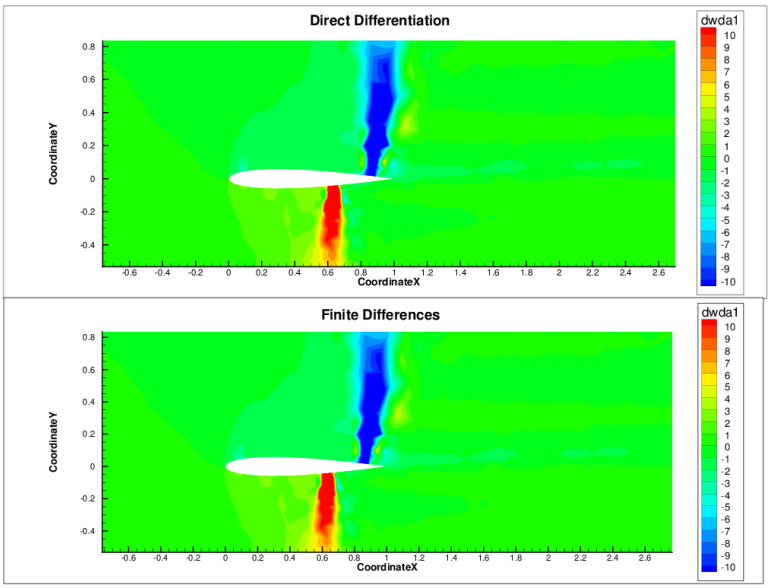


Figure 4.9: Density sensitivity computed with DDM and FD respectively.

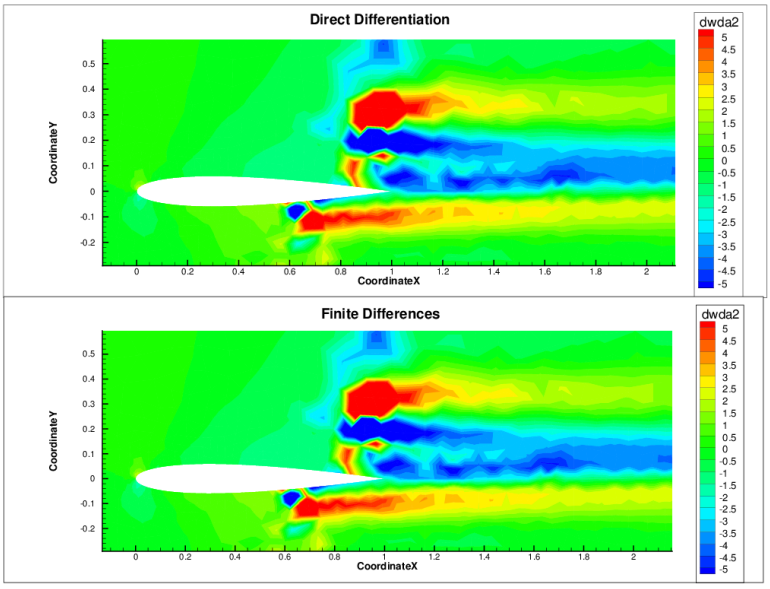


Figure 4.10: x-momentum sensitivity computed with DDM and FD respectively .

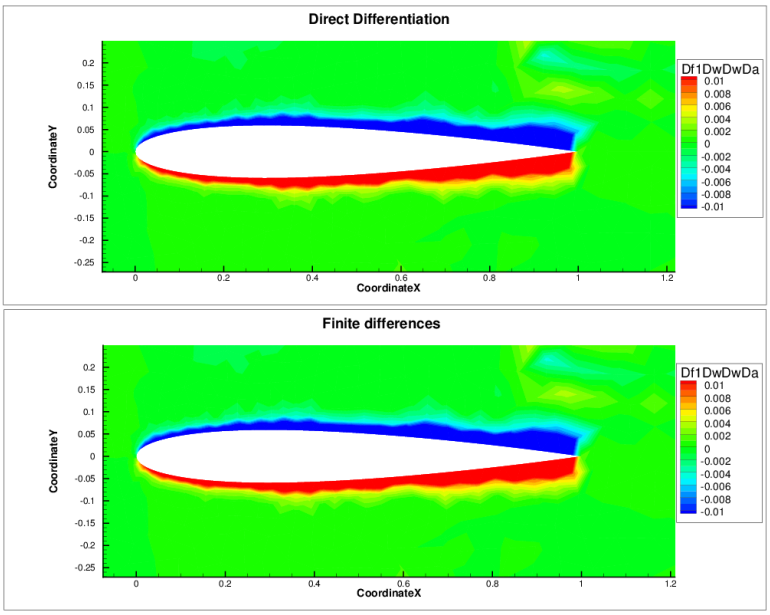


Figure 4.11: Sensitivity of the 1st component of the flux balance with respect to the flow, computed with DDM and FD respectively.

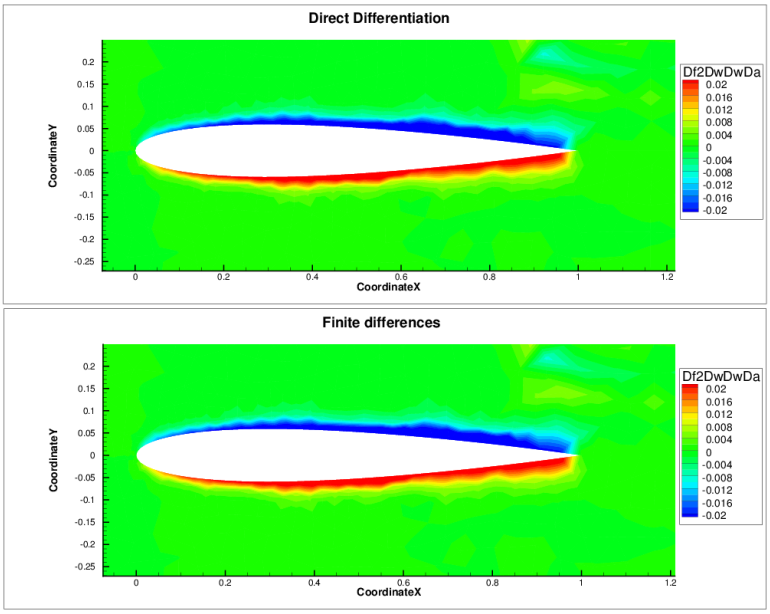


Figure 4.12: Sensitivity of the 2nd component of the flux balance with respect to the flow, computed with DDM and FD respectively.

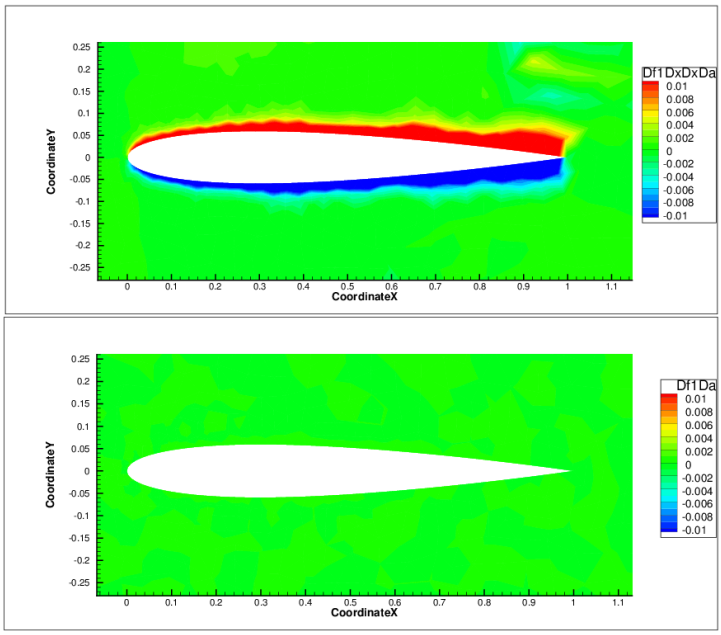


Figure 4.13: Comparison between partial sensitivity with respect to the mesh of the flux balance and total sensitivity of the flux balance (1st component).

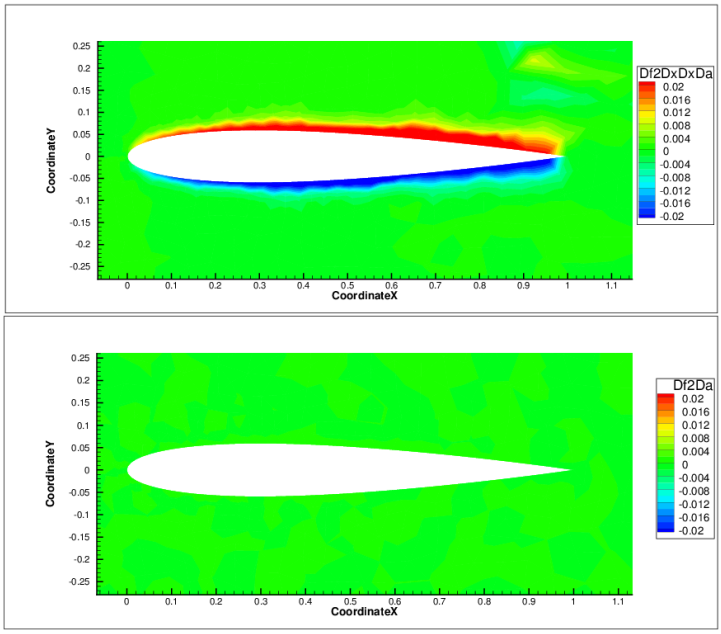


Figure 4.14: Comparison between partial sensitivity with respect to the mesh of the flux balance and total sensitivity of the flux balance (2nd component).

The results are shown if figures 4.11 and 4.12.
Moreover, the total derivative of the flux balance with respect to the parameter α has been proven to be equal to zero, as it must be according to equation (3.19). By comparing it with the partial sensitivity of the flux balance with respect to the mesh only, $\frac{\partial R}{\partial X}\frac{dX}{d\alpha}$, a decrease of 4 orders of magnitudes has been observed, as illustrated in figures 4.13 and 4.14.
The final verification is the value of the sensitivity of lift and drag with respect to the parameter α. The result and the comparison with finite differences are shown in tables 4.1 and 4.2, in which the values of the partial sensitivities (only mesh or flow dependency respectively) and of the total sensitivity are presented. However, it is important to notice that the code developed is only responsible for the computation of the partial sensitivities with respect to the flow, $\frac{\partial D}{\partial W}\frac{dW}{d\alpha}$ and $\frac{\partial L}{\partial W}\frac{dW}{d\alpha}$ in tables 4.1 and 4.2.

Table 4.1: Direct differentiation - Drag sensitivity to parameter α.

	Finite differences	Direct differentiation
$\frac{\partial D}{\partial X}\frac{dX}{d\alpha}$	1.760e-01	1.760e-01
$\frac{\partial D}{\partial W}\frac{dW}{d\alpha}$	2.933e-01	2.932e-01
$\frac{dD}{d\alpha}$	4.693e-01	4.692e-01

Table 4.2: Direct differentiation - Lift sensitivity to parameter α.

	Finite differences	Direct differentiation
$\frac{\partial L}{\partial X}\frac{dX}{d\alpha}$	-3.008e-02	-2.918e-02
$\frac{\partial L}{\partial W}\frac{dW}{d\alpha}$	7.307e+00	7.307e+00
$\frac{dL}{d\alpha}$	7.278e+00	7.278e+00

4.2.2 Adjoint method (parameter mode)

The verification of the discrete adjoint method has been based on the comparison with the direct differentiation method, after it had already been validated.
Since the code developed for adjoint method performs the computation of the term:

$$\lambda^T \frac{\partial R}{\partial W},$$

this will be the object of the verification. An efficient way of doing it, is the cross check of the following equality:

$$\bar{\lambda}^T \cdot \left(\frac{\partial R}{\partial W}\overline{\frac{dW}{d\alpha}}\right)_{ddm} = \left(\bar{\lambda}^T \frac{\partial R}{\partial W}\right)_{adj} \cdot \overline{\frac{dW}{d\alpha}}, \tag{4.6}$$

in which $\bar{\lambda}$ and $\overline{\frac{dW}{d\alpha}}$ are two arbitrarily chosen vectors. On the left hand side, the term between brackets is computed using the previously verified direct differentiation method routine and is then scalarly multiplied by the arbitrary vector $\bar{\lambda}$. On the right hand side, instead, the term in brackets is computed in the adjoint method routine and must give exactly the same result when is multiplied by $\overline{\frac{dW}{d\alpha}}$.
This cross check has been carried out for different choices of the two arbitrary vectors. For example, in the case in which $\overline{\frac{dW}{d\alpha}}$ is taken as the real solution of system (3.20) and $\bar{\lambda}$ as the real solution of system (3.30) relatively to drag (or similarly for lift), the difference between the two computed terms is about 10^{-4} times their order of magnitude.
The final and definitive check has been done on the sensitivity value. The results are shown in table 4.3 and 4.4. Also in this case, the code developed is only responsible for the computation of the partial sensitivity with respect to the flow, $\frac{\partial D}{\partial W}\frac{dW}{d\alpha}$ and $\frac{\partial L}{\partial W}\frac{dW}{d\alpha}$.

Table 4.3: Adjoint (parameter mode) - Drag sensitivity to parameter α.

	Finite differences	Direct differentiation	Adjoint
$\frac{\partial D}{\partial X}\frac{dX}{d\alpha}$	1.760e-01	1.760e-01	1.760e-01
$\frac{\partial D}{\partial W}\frac{dW}{d\alpha}$	2.933e-01	2.932e-01	2.932e-01
$\frac{dD}{d\alpha}$	4.693e-01	4.692e-01	4.692e-01

Table 4.4: Adjoint (parameter mode) - Lift sensitivity to parameter α.

	Finite differences	Direct differentiation	Adjoint
$\frac{\partial L}{\partial X}\frac{dX}{d\alpha}$	-3.008e-02	-2.918e-02	-2.918e-02
$\frac{\partial L}{\partial W}\frac{dW}{d\alpha}$	7.307e+00	7.307e+00	7.307e+00
$\frac{dL}{d\alpha}$	7.278e+00	7.278e+00	7.278e+00

4.2.3 Adjoint method (mesh mode)

The verification of the $\frac{dJ}{dX}$ method was based on a comparison with the discrete adjoint code. Since the code for the computation of the adjoint vector λ is shared by the two methods, the only possible source of error would be the matrix $\frac{\partial R}{\partial X}$. A fast and efficient way of verifying the code is to compare the result of the scalar product $\lambda \cdot \frac{\partial R}{\partial X}\frac{dX}{d\alpha}$ obtained with the adjoint parameter mode method and the adjoint mesh mode method. In the former, the term $\frac{\partial R}{\partial X}\frac{dX}{d\alpha}$ is computed using finite differences, while in the second case the term $\lambda\frac{\partial R}{\partial X}$ is actually built inside the routine and successively multiplied by the vector $\frac{dX}{d\alpha}$. For any arbitrary choice of the vector $\bar{\lambda}$ (not only for the

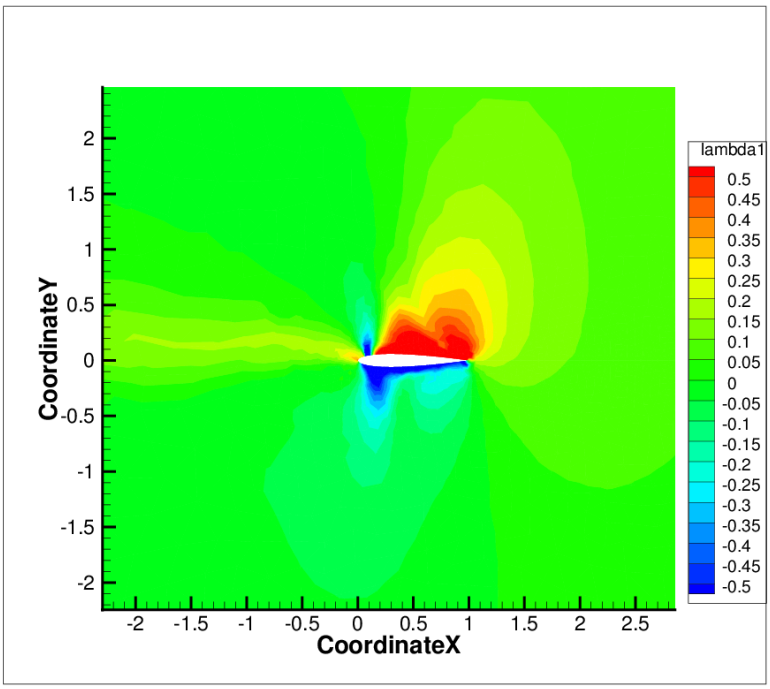


Figure 4.15: First component of the adjoint vector related to drag.

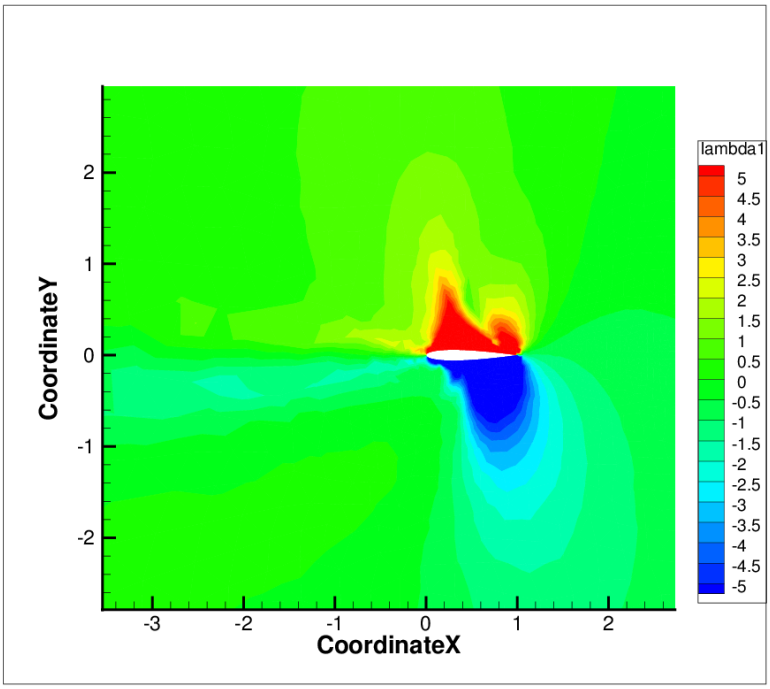


Figure 4.16: First component of the adjoint vector related to lift.

real solution of system (3.30)), the following relation must be satisfied:

$$\bar{\lambda}^T \cdot \left(\frac{R\left[X(\alpha + d\alpha)\right] - R\left[X(\alpha - d\alpha)\right]}{2d\alpha} \right) = \left(\bar{\lambda}^T \frac{\partial R}{\partial X} \right) \cdot \frac{dX}{d\alpha}. \tag{4.7}$$

This relation has been checked for several choices of $\bar{\lambda}$. For example in the case in which $\bar{\lambda}$ is the real solution of system (3.30) relatively to drag (or similarly for lift), the difference between the two computed terms is about 10^{-4} times their order of magnitude.
The definitive check on the accuracy of the $\frac{dJ}{dX}$ vector computed consists in moving some of the nodes of the mesh and computing by finite differences the change in the goal function J and finally comparing it with the output of the code.

$$\frac{dJ}{dX} \approx \frac{J(X + dX) - J(X - dX)}{2dX}. \tag{4.8}$$

This check has been carried out with positive results for several nodes of the mesh, in different mesh locations, including the boundaries. For example in the case of the displacement of a node in the shock area, as shown in figure 4.17, the difference between left and right side of equation (4.8) is about 10^{-4} times their order of magnitude.

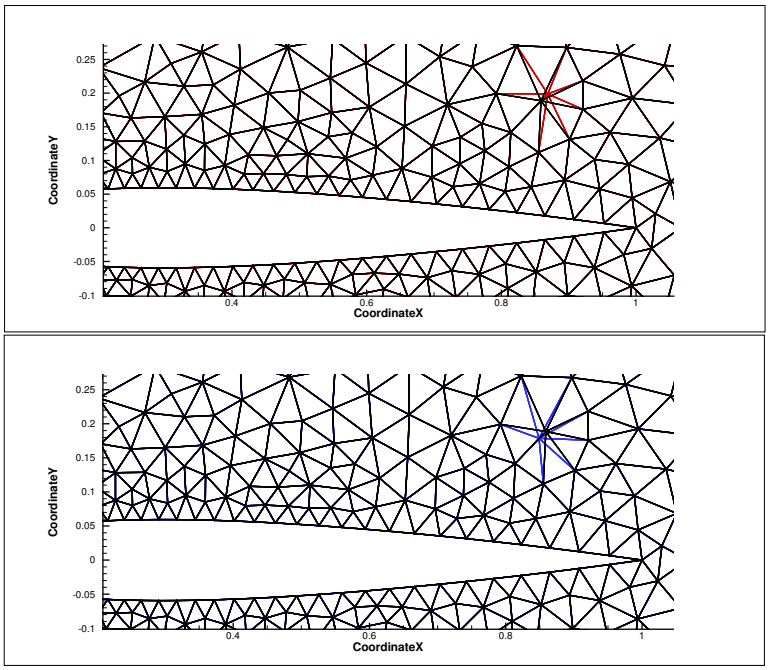


Figure 4.17: Node displacement in shock region (respectively plus and minus dX).

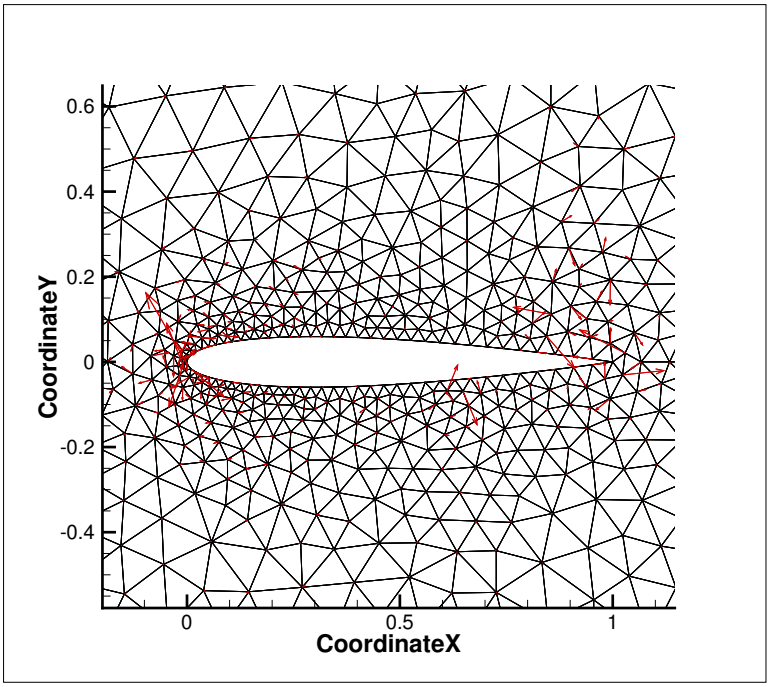


Figure 4.18: Plot of $dJdX$ vector for drag.

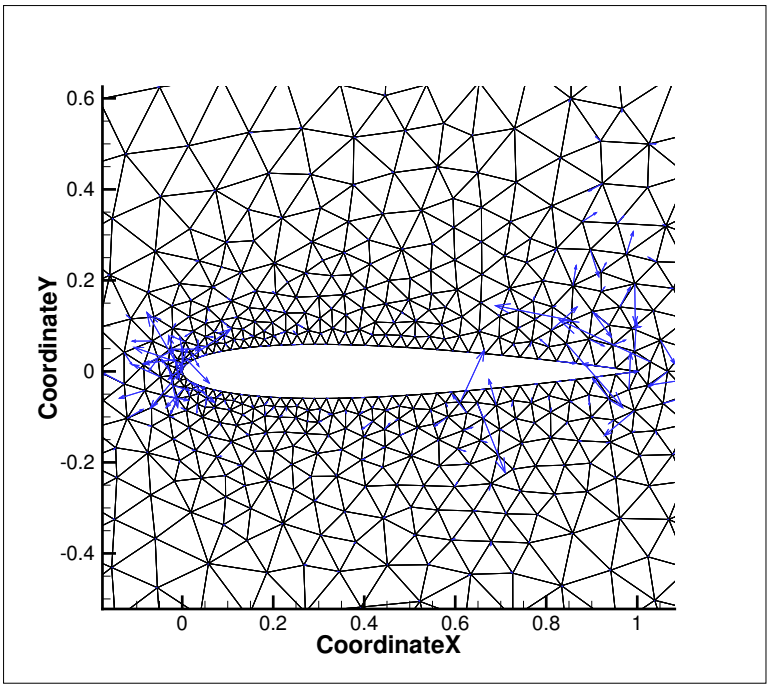


Figure 4.19: Plot of $dJdX$ vector for lift.

The last verification has been done on the values of sensitivity of lift and drag. In table 4.5 and 4.6 the final results of the whole verification process are summarized.

Table 4.5: Adjoint (mesh mode) - Drag sensitivity to parameter α.

	FD	DD	Adjoint	$\frac{dJ}{dX} \cdot \frac{dX}{d\alpha}$
$\frac{\partial D}{\partial X}\frac{dX}{d\alpha}$	1.760e-01	1.760e-01	1.760e-01	1.760e-01
$\frac{\partial D}{\partial W}\frac{dW}{d\alpha}$	2.933e-01	2.932e-01	2.932e-01	2.932e-01
$\frac{dD}{d\alpha}$	4.693e-01	4.692e-01	4.692e-01	4.692e-01

Table 4.6: Adjoint (mesh mode) - Lift sensitivity to parameter α.

	FD	DD	Adjoint	$\frac{dJ}{dX} \cdot \frac{dX}{d\alpha}$
$\frac{\partial L}{\partial X}\frac{dX}{d\alpha}$	-3.008e-02	-2.918e-02	-2.918e-02	-2.918e-02
$\frac{\partial L}{\partial W}\frac{dW}{d\alpha}$	7.307e+00	7.307e+00	7.307e+00	7.307e+00
$\frac{dL}{d\alpha}$	7.278e+00	7.278e+00	7.278e+00	7.278e+00

An important final remark on the complete verification procedure is that, both in the case of the direct differentiation method and in the case of the adjoint method, the Newton relaxation method has not been successfull in bringing the systems of equations of type (3.20) and (3.30) to convergence. For this reason, it has been necessary to code and extension to unstructured meshes of the recursive projection algorithm [28] presented at the end of chapter 2, in order to eliminate the unstable eigenmodes of the iteration matrix.

CHAPTER 5
MESH ADAPTATION

5.1 Adaptation procedure

The vector field $\frac{dJ}{dX}$, that can be computed using the code developed, can be used as a local estimator in a goal oriented mesh adaptation strategy. As a matter of fact, large components of $\frac{dJ}{dX}$ in a specific mesh node indicate that the value of the goal function J is particularly sensitive to the node displacement at that location. This simple observation suggests that refining the mesh in areas with large $\frac{dJ}{dX}$ components and large cells sizes (large possible displacements of the nodes), would produce an adapted mesh on which the computation of the goal function J would be more accurate. This assumption has been proven to be correct by Peter et al. in [6, 7], in which the investigation has been limited to structured meshes. In this chapter, the steps followed in order to extend this results to unstructured meshes will be carefully described.
Each step of the adaptation procedure, with the creation of an adapted mesh from an initial one, can be split in four different stages, as illustrated in figure 5.1, in which blue boxes refer to operations carried out through *elsA*, while red boxes indicate external modules. The first stage is the steady state computation in *elsA*. Once the flow is converged, the aim will be the computation of the partial derivatives of the goal function with respect to the mesh coordinates and the flow. This stage requires the creation of an external module, that in my case has been a set of Python scripts, a part of which is reported in appendix A. The partial derivatives computed, are necessary inputs for the computation of the $\frac{dJ}{dX}$ vector in *elsA*. Finally, a new external module has been created, a part of which is reported in appendix B, for the computation of the local estimator and for the creation of the adapted mesh based on the value of the local estimator itself. At this point, one step of the adaptation procedure is complete and it is possible to go on,

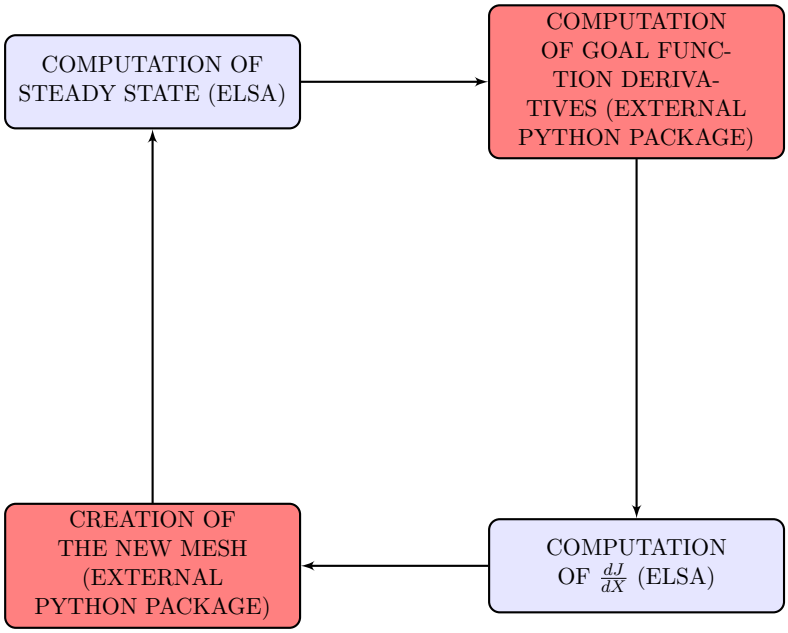


Figure 5.1: Adaptation procedure.

starting again with the steady state computation.
The creation of the new mesh by means of the $\frac{dJ}{dX}$ vector can be further analyzed. The first part is the projection of the $\frac{dJ}{dX}$ vector field at the boundaries of the domain. This procedure has also been followed by Peter et al. in [6, 7]. The objective is to eliminate the components of $\frac{dJ}{dX}$ that cannot be used in the framework of mesh optimization without modification of the shape of the object. The new vector field $P\left(\frac{dJ}{dX}\right)$ is defined as follows:

$$P\left(\frac{dJ}{dX}\right) = \begin{cases} \frac{dJ}{dX} \cdot \vec{n} & \text{at the boundaries} \ , \\ 0 & \text{at corners} \, , \\ \frac{dJ}{dX} & \text{elsewhere} \, . \end{cases}$$

where $\vec{n}$ is the normal vector at each boundary interface.
At this point, the local estimator can be computed. In the present thesis, the same estimator used by Peter et al. in [6, 7] has been used, that is a node quantity equal to the product of $\left|\left|P\left(\frac{dJ}{dX}\right)\right|\right|$ by some characteristic length:

$$\theta = \left|\left|P\left(\frac{dJ}{dX}\right)\right|\right| \cdot \frac{d_{min}}{2}, \tag{5.1}$$

where the characteristic length d_{min} has been computed at each node, as the minimum distance to the neighbouring nodes.
For the creation of the new mesh, the open source software MMG2D, [41], developed at INRIA Bordeaux, has been used. MMG2D is a software for

mesh adaptation that works with 2D meshes, while MMG3D is the version for 3D meshes.
It requires two inputs:

- The coordinates of the nodes of the mesh.
- The desired metric, that is a node quantity, based on which the new adapted mesh will be built.

The desired metric that is given as input can be either a scalar or a vector quantity. In the present thesis only the scalar metric input has been used. The exact definition of metric, or in other words the exact way in which the metric input file is interpreted by MMG2D, is explained in the software manual [41]. However, a very good approximation of the metric of the mesh is the average value between the minimum and maximum distance to the neighbouring nodes. This means that, if this quantity is computed for the initial mesh and given as input to MMG2D as target metric for the new mesh to be built, the output mesh will be almost identical to the initial one, which corresponds to no change or very little change. This node quantity will be named μ_{init} in the following and is equal to:

$$\mu_{init} = \frac{d_{min} + d_{max}}{2}. \tag{5.2}$$

The objective will be to change this initial metric μ_{init} in such a way to keep into account the local estimator θ previously computed. This leads to the computation of the desired metric μ_{target}, that will be the metric of the output mesh created by MMG2D. The criterion used for the adaptation is based on a threshold value of the estimator. In all nodes in which the estimator is higher than the prefixed tolerance, the metric μ_{init} will be changed to the desired metric μ_{target}:

$$\mu_{target} = \frac{\mu_{init}}{f}, \tag{5.3}$$

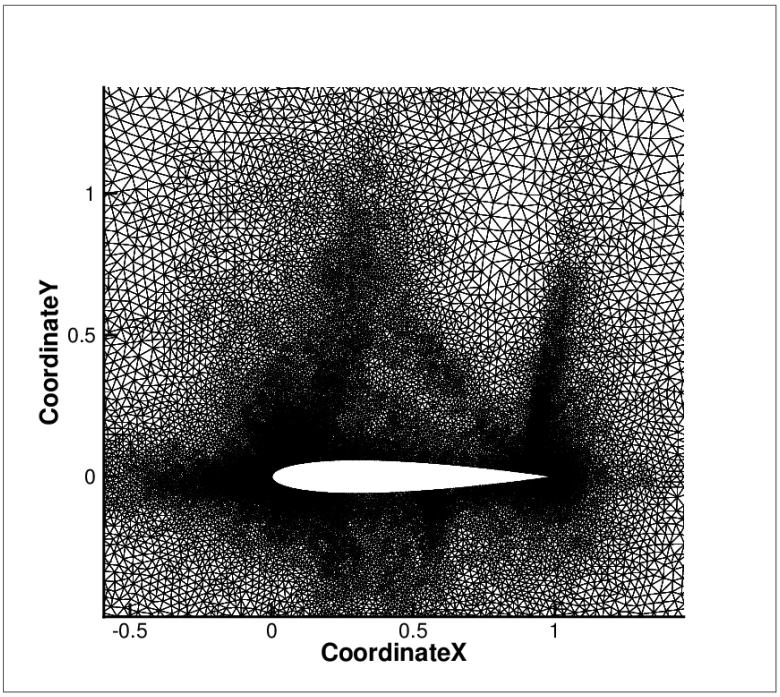
where f is the split factor, that is a node quantity indicating how much the metric has to be reduced or in other words how much the mesh has to be refined in each node. The split factor f has been defined as:

$$f = \sqrt{2^{[log_{10}(\theta) - log_{10}(\epsilon)]}}, \tag{5.4}$$

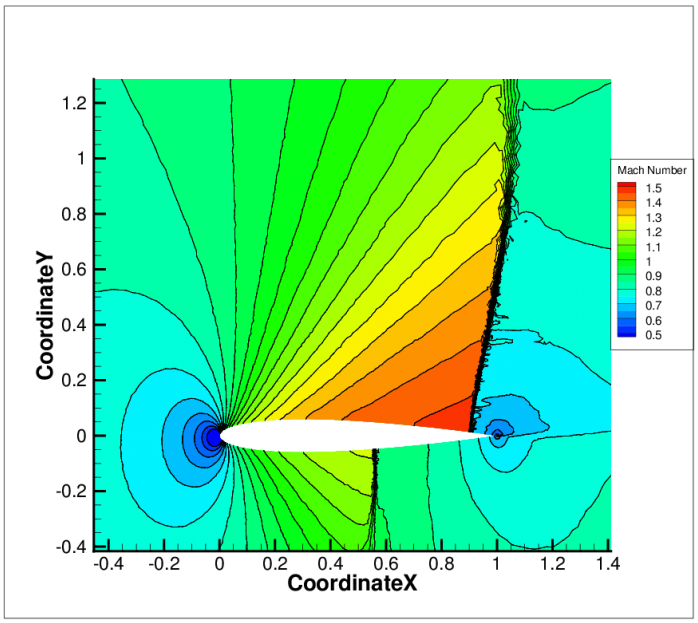
where ϵ is the value of the threshold fixed on the value of the estimator. From equation (5.4) it can be deduced that, if the estimator θ is one order of magnitude higher than the tolerance, the metric of the adapted mesh will be reduced by a factor $f = \sqrt{2}$ and if it would be two orders of magnitude higher than the tolerance, f would be equal to 2 and so on.
The definition of the desired metric described, can be summarized with the following piece of pseudo-code:

```
IF (estimator > tol):
    splitf = 2^[sqrt(log10(estimator) - log10(tol))]
    metric_new = metric_old/splitf
```

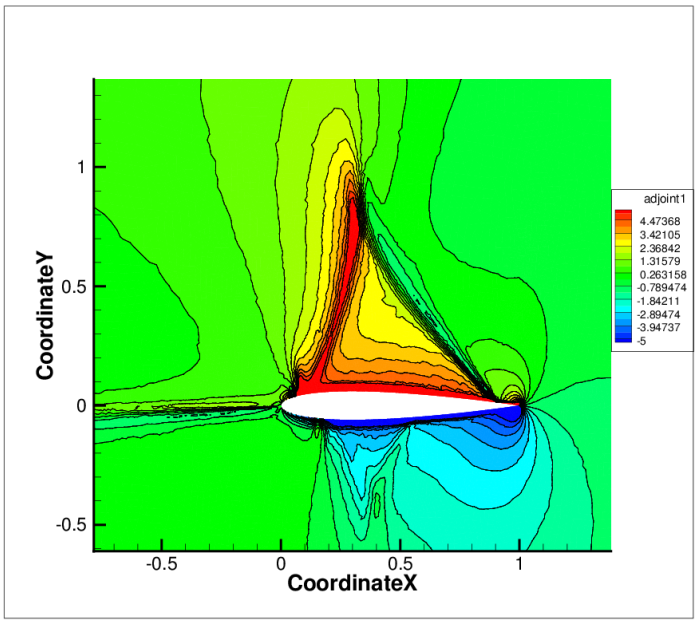
Once the target metric is computed, all the necessary inputs to MMG2D are ready and the output mesh can be obtained. At this point there are still two issues to be solved before going to the next step of the adaptation. First of all, the new nodes created at the boundary, are positioned by MMG2D on the straight line between the two nearest neighbouring nodes. This is improved by projecting the new nodes on the real profile of the boundary. The second point is the conversion of the mesh from the MMG2D format to the CGNS format, necessary for the next step steady state computation in *elsA*. After this, the cycle illustrated in figure 5.1 is ready to start again.
In figures 5.2a to 5.2g all the quantities involved in one step of the adaptation procedure are presented.
Throughout the whole mesh adaptation procedure, as for the code verification part, the recursive projection algorithm [28], briefly described in chapter 2, has been necessary in order to guarantee convergence of the systems of equations (3.20) and (3.30).



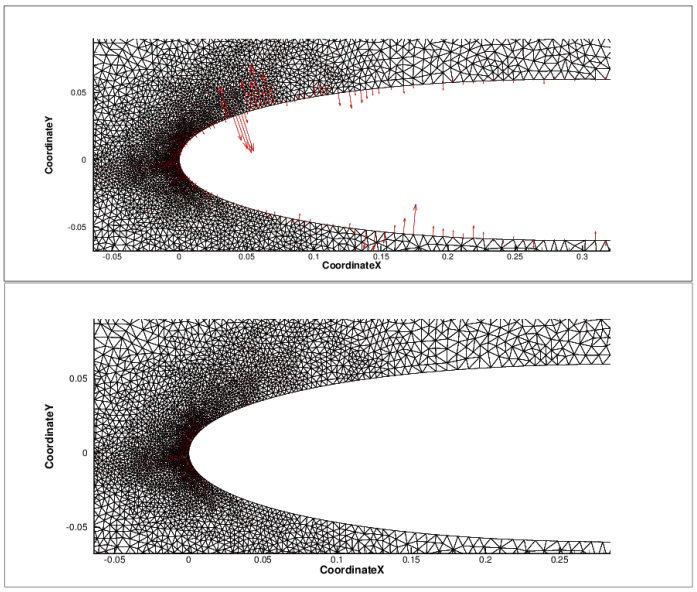
(a) Starting mesh.



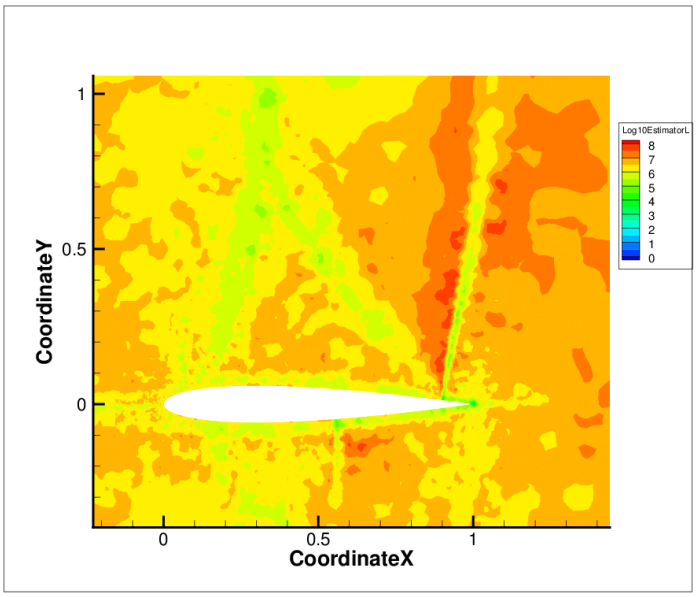
(b) Flow computed on the starting mesh.



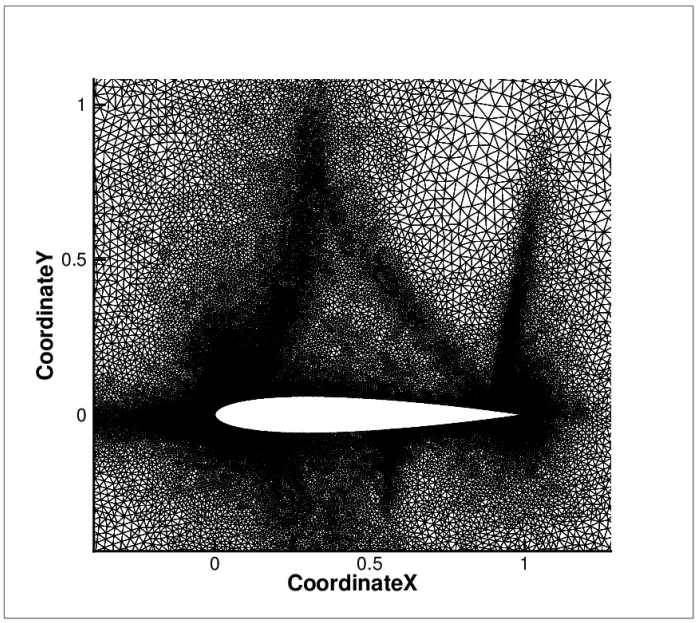
(c) 1st component of the adjoint vector.



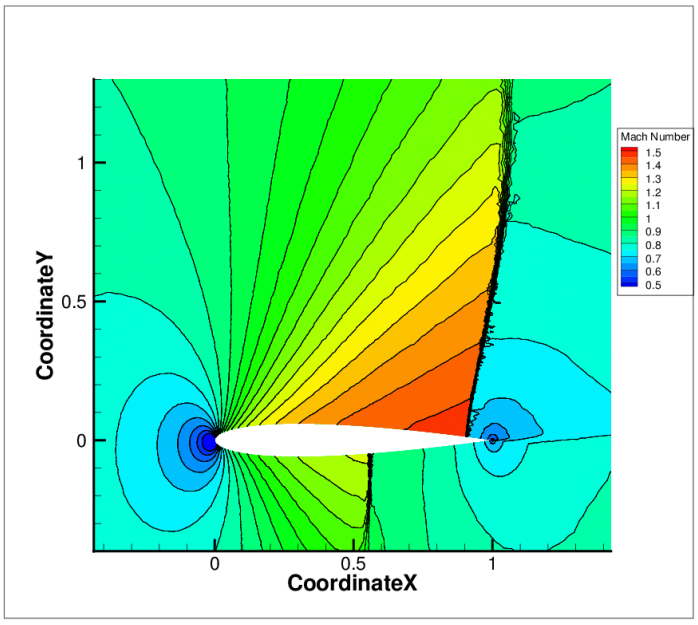
(d) $\frac{dJ}{dX}$ (above) and $P\frac{dJ}{dX}$ (below).



(e) Estimator.



(f) Adapted mesh.



(g) Flow computed on the adapted mesh.

Figure 5.2: All the quantities involved in the adaptation procedure.

5.2 Results NACA0012 - Lift and drag

The adaptation procedure described, has been applied to a NACA0012 mesh with three different flow conditions. A subsonic case, $M = 0.5$ $AoA = 0°$, a transonic case, $M = 0.85$ $AoA = 2°$ and a supersonic case $M = 1.5$ $AoA = 1°$. The goal functions considered were lift and pressure drag.
For all the three cases, the initial mesh was the same, figure 5.3. This mesh has 4841 nodes and the distance of the boundaries is 8 chord lengths upwind and 10 chord lengths downwind of the airfoil.
Due to the difficulties encountered in obtaining a steady state solution, or a solution to the adjoint system of equation, only 3 successive adaptation steps were completed for the transonic and supersonic test cases and only 2 steps for the subsonic one. The results of the three cases are presented in the following sections.

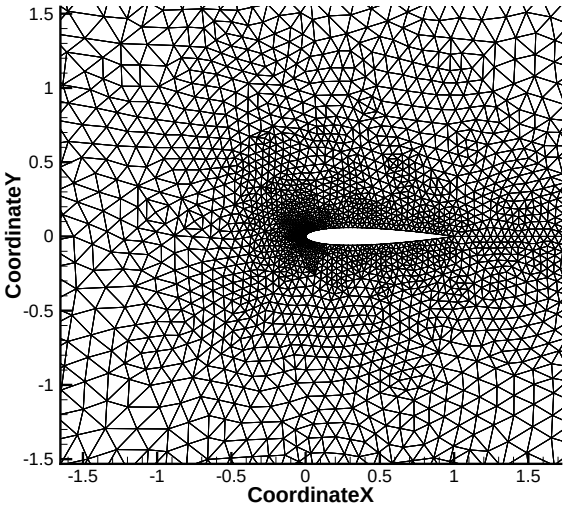


Figure 5.3: The initial mesh

Since lift and drag, for Euler flows, are both functions of pressure around the airfoil, the quantities involved in the mesh adaptation procedure (and the adapted meshes themselves) are not so different in the lift and drag adaptation procedures. For this reason the plots presented in the following sections are relative to lift only. For each case, the plot of the first component of the adjoint vector is given, together with a plot of the $\frac{dL}{dX}$ vector field in proximity of the airfoil. Moreover, a plot of the meshes obtained after each step of adaptation is given, together with the plot of the logarithm of the estimator. In conclusion, for each case, a table is presented with the average values of the estimator θ respectively for lift and drag. In all the cases and

after each adaptation step, the average value of the estimator is proven to decrease, as also shown by Peter et al. in [6, 7]. The tolerance value ϵ appearing in formula (5.4) has been heuristically fixed to 10^{-8} for drag and 10^{-7} for lift.

5.2.1 $M = 0.5$ $AoA = 0°$

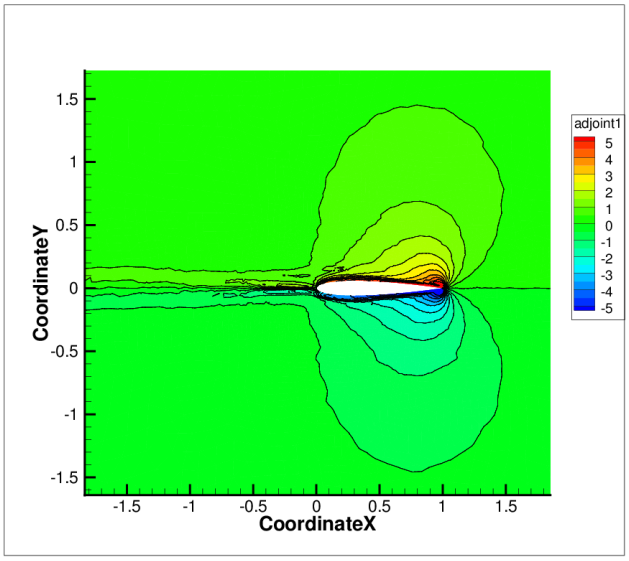


Figure 5.4: Subsonic case - 1st component of adjoint vector relative to lift.

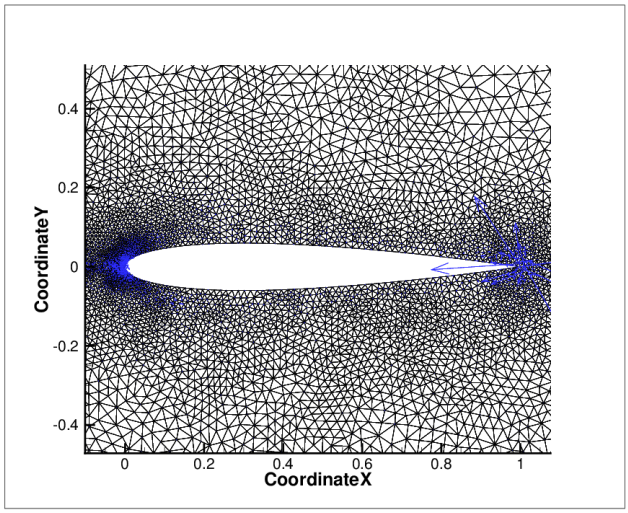


Figure 5.5: Subsonic case - $\frac{dJ}{dX}$ vector field.

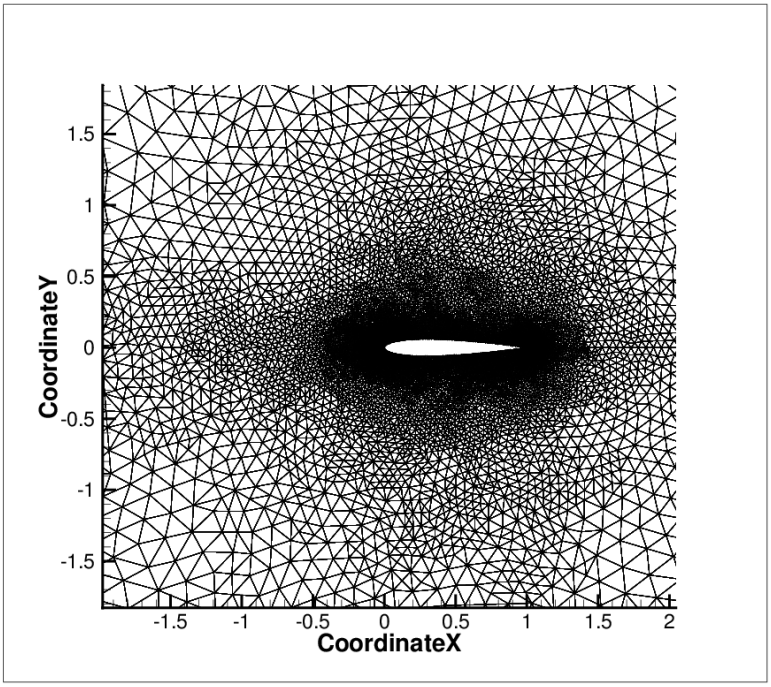


Figure 5.6: Subsonic case - 1st adaptation step.

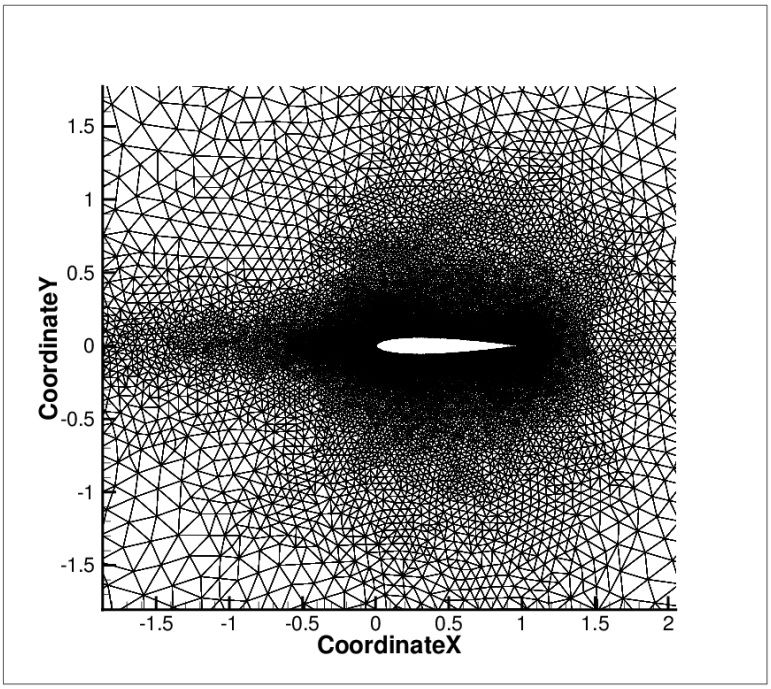


Figure 5.7: Subsonic case - 2nd adaptation step.

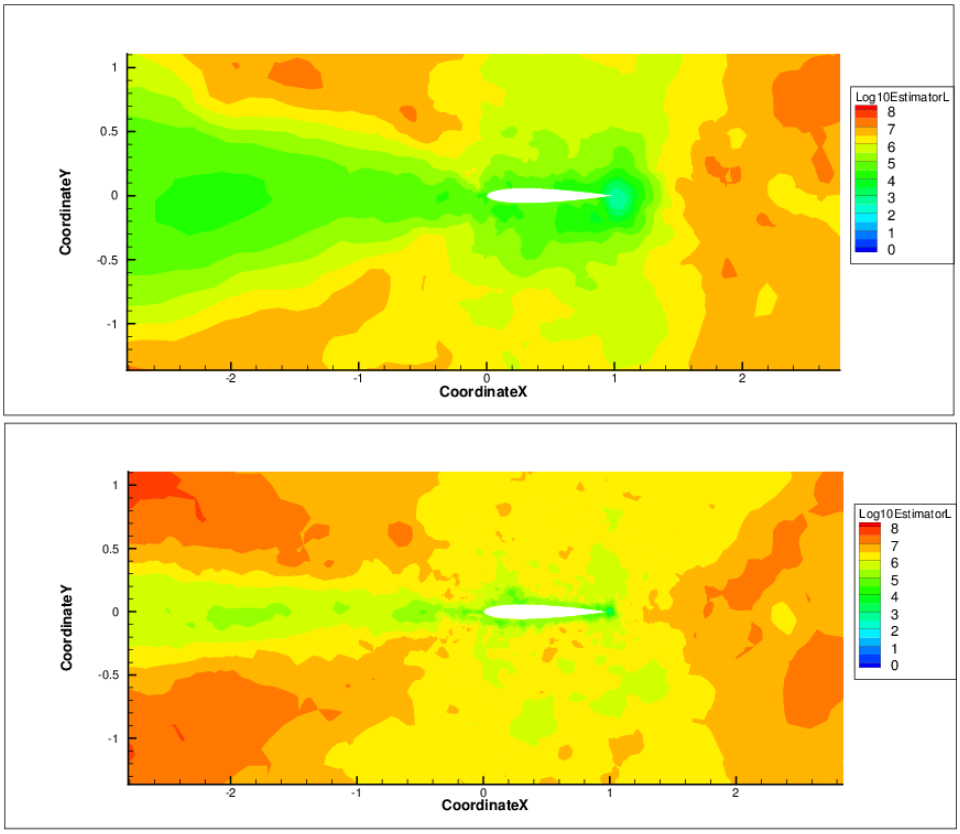


Figure 5.8: Subsonic case - Plot of minus the logarithm of the estimator.

Table 5.1: Subsonic case - Number of nodes at each step.

	STEP1	STEP2	STEP3
Nodes number (drag adaptation)	9493	17562	-
Nodes number (lift adaptation)	11109	21829	-

Table 5.2: Subsonic case - Average values of the estimator at each step.

	STEP1	STEP2	STEP3
$\bar{\theta}_{drag}$	4.5184e-07	1.3780e-07	-
$\bar{\theta}_{lift}$	1.2103e-05	2.9148e-06	-

5.2.2 $M = 0.85$ $AoA = 2°$

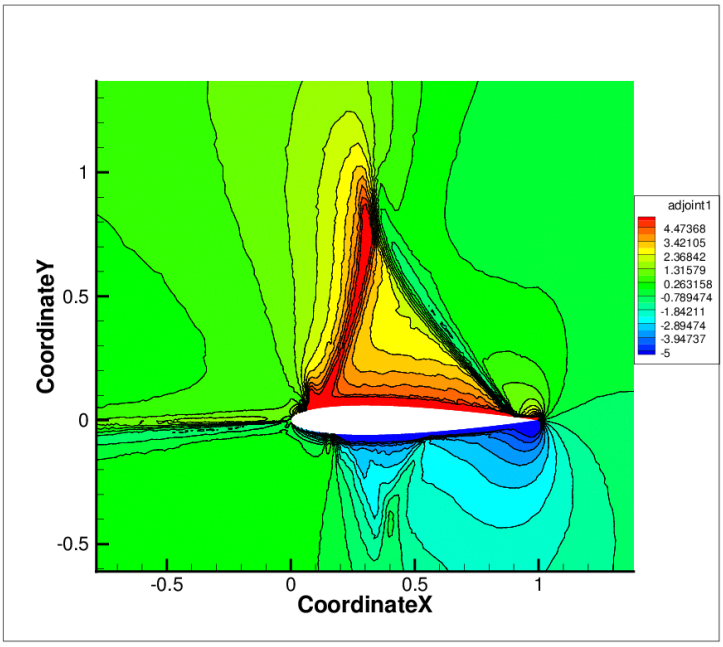


Figure 5.9: Transonic case - 1st component of adjoint vector relative to lift.

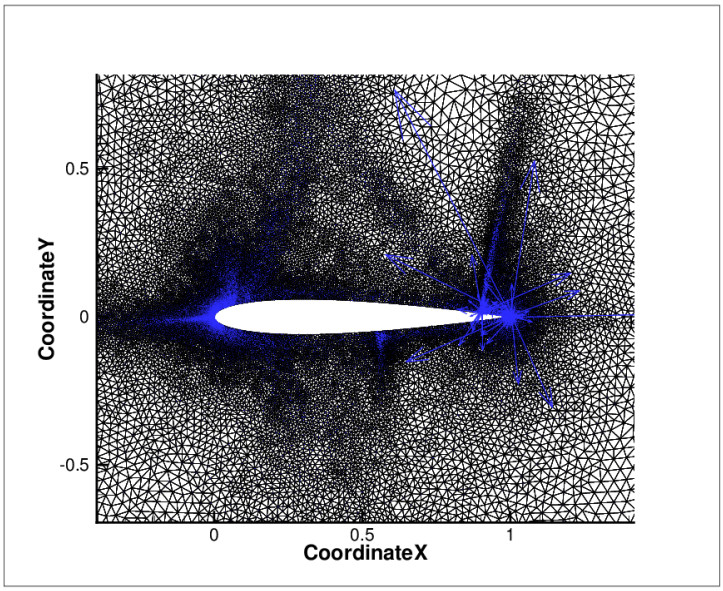


Figure 5.10: Transonic case - $\frac{dL}{dX}$ vector field.

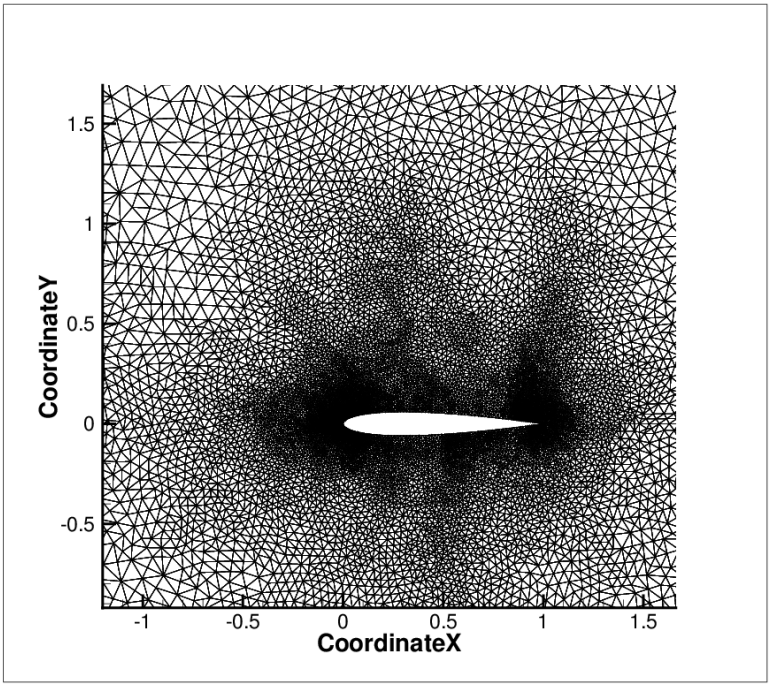


Figure 5.11: Transonic case - 1st adaptation step.

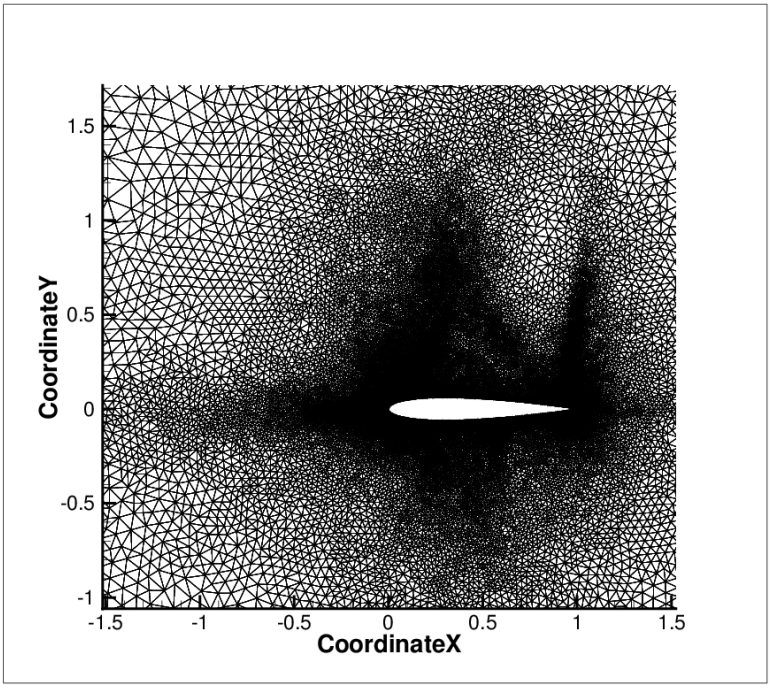


Figure 5.12: Transonic case - 2nd adaptation step.

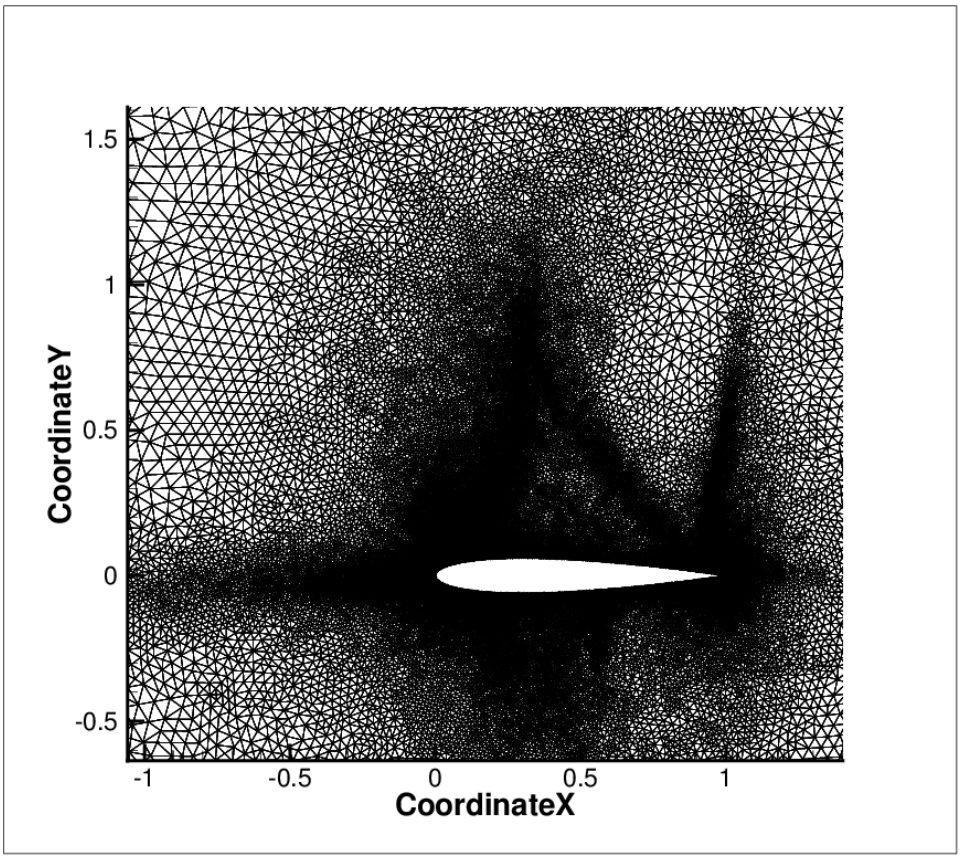


Figure 5.13: Transonic case - 3rd adaptation step.

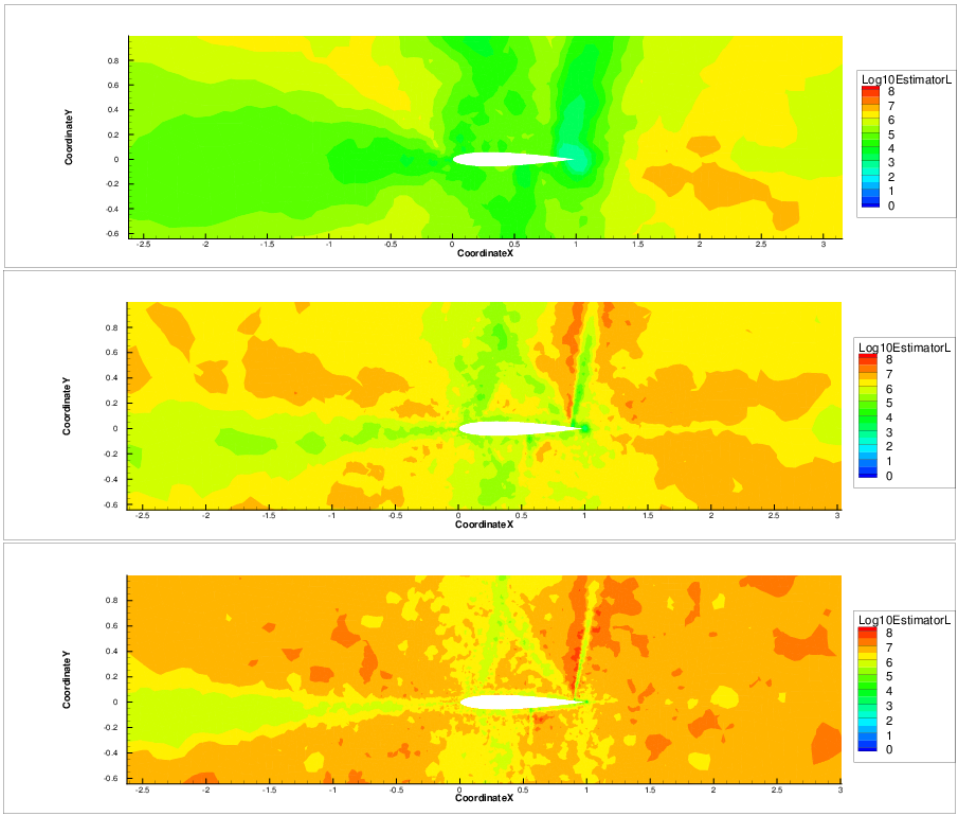


Figure 5.14: Transonic case - Plot of minus the logarithm of the estimator.

Table 5.3: Transonic case - Number of nodes at each step.

	STEP1	STEP2	STEP3
Nodes number (drag adaptation)	8423	24636	49755
Nodes number (lift adaptation)	15784	32341	47939

Table 5.4: Transonic case - Average values of the estimator at each step.

	STEP1	STEP2	STEP3
$\bar{\theta}_{drag}$	2.3632e-06	7.4657e-07	1.9818e-07
$\bar{\theta}_{lift}$	2.6818e-05	2.6584e-06	7.8659e-07

5.2.3 $M = 1.50$ $AoA = 1°$

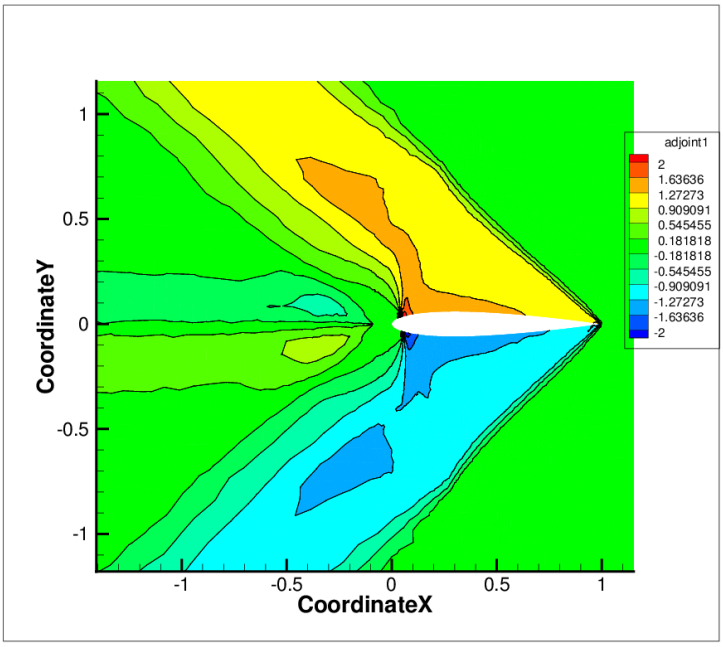


Figure 5.15: Supersonic case - 1st component of the adjoint vector relative to lift.

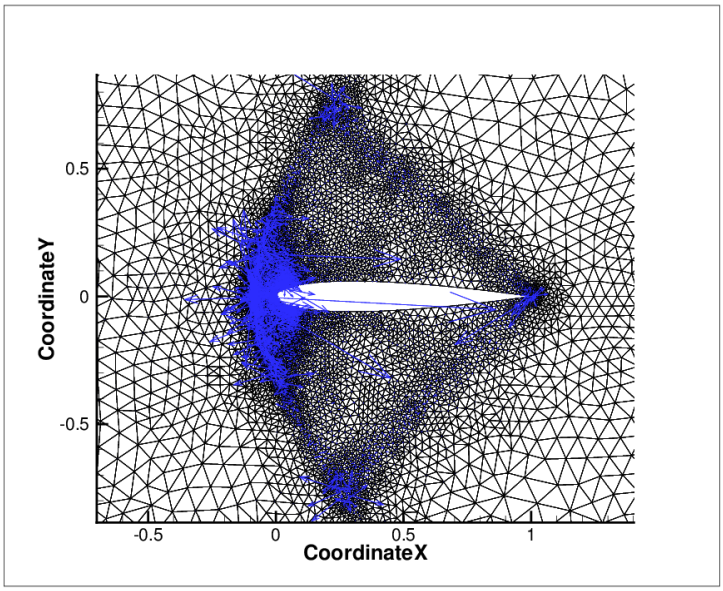


Figure 5.16: Supersonic case - $\frac{dL}{dX}$ vector field.

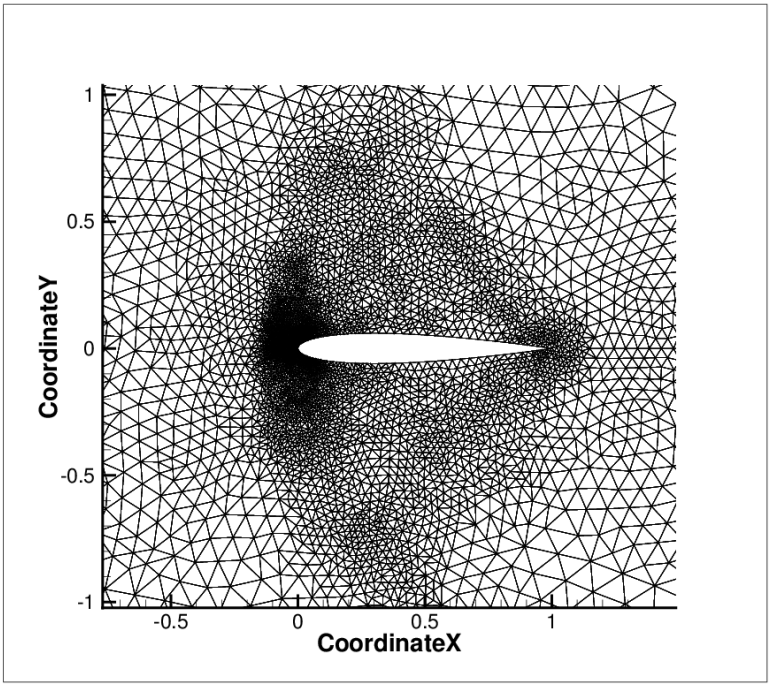


Figure 5.17: Supersonic case - 1st adaptation step.

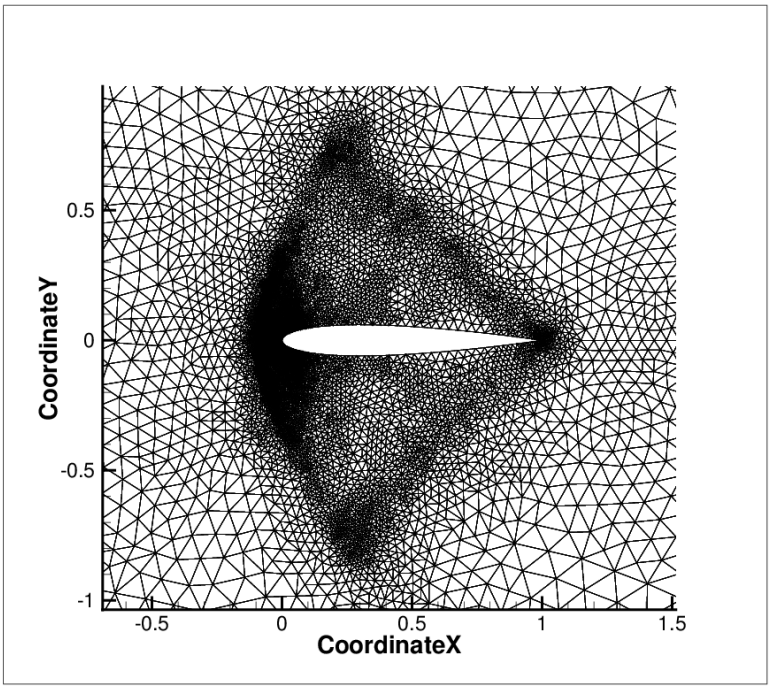


Figure 5.18: Supersonic case - 2nd adaptation step.

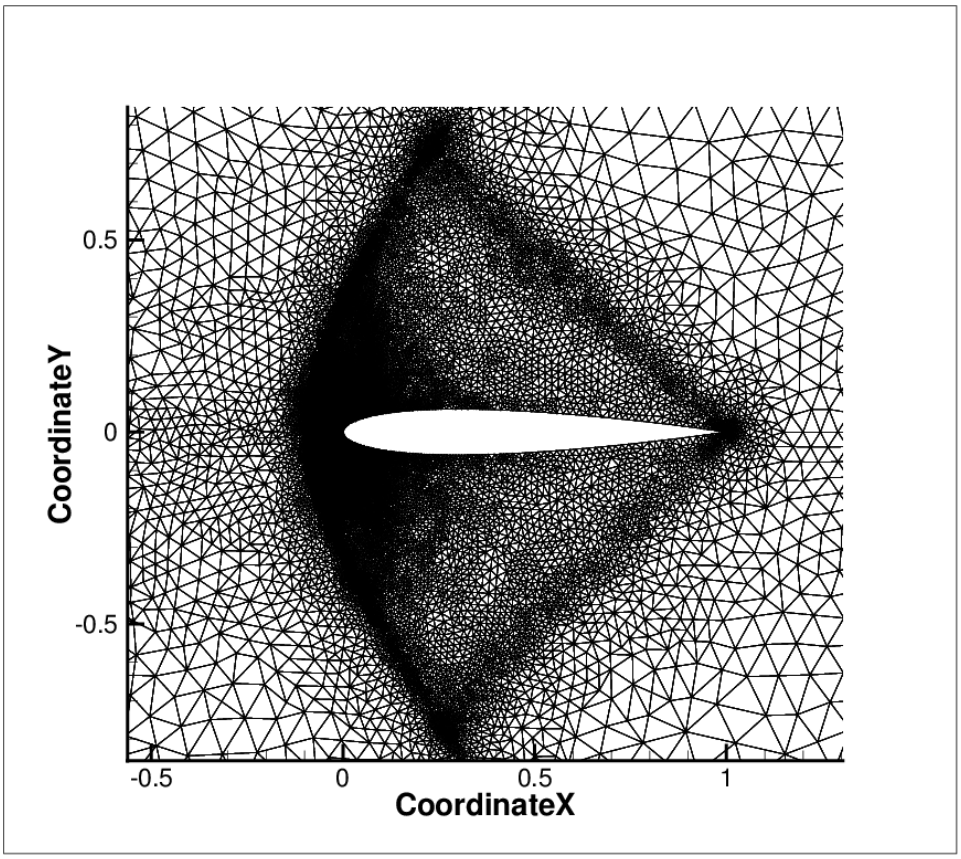


Figure 5.19: Supersonic case - 3rd adaptation step.

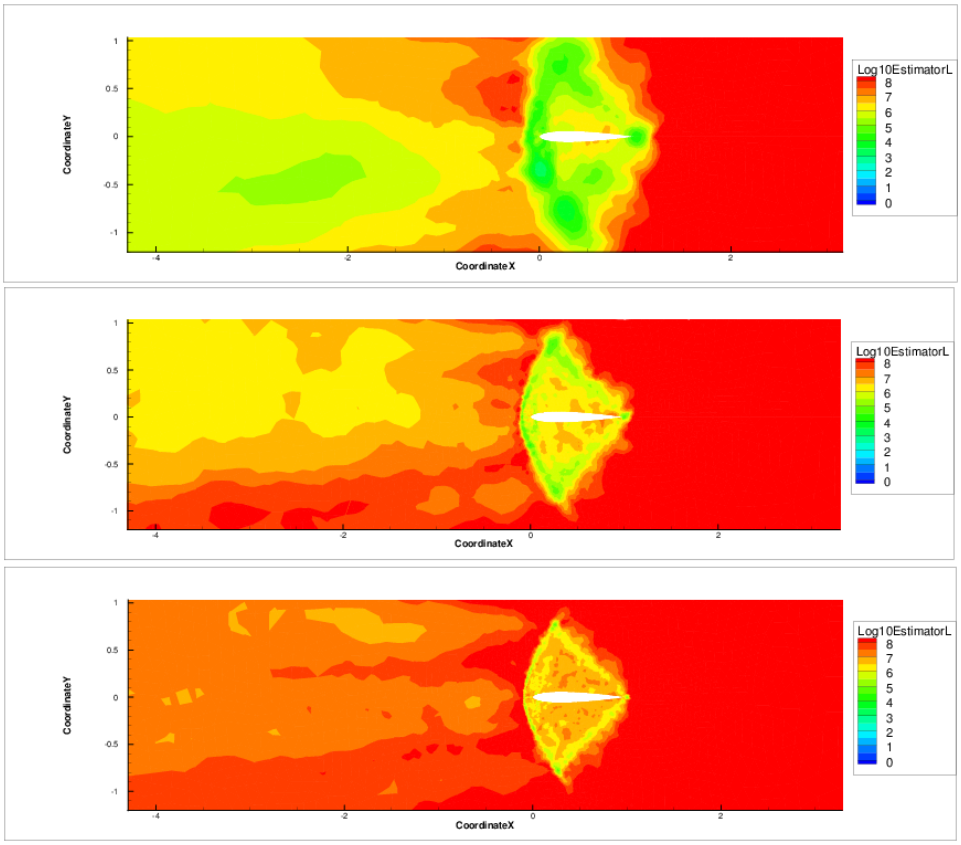


Figure 5.20: Supersonic case - Plot of minus the logarithm of the estimator.

Table 5.5: Supersonic case - Number of nodes at each step.

	STEP1	STEP2	STEP3
Nodes number (drag adaptation)	10182	17849	25427
Nodes number (lift adaptation)	8096	12072	15092

Table 5.6: Supersonic case - Average values of the estimator at each step.

	STEP1	STEP2	STEP3
$\bar{\theta}_{drag}$	1.1623e-06	1.6233e-07	5.6068e-08
$\bar{\theta}_{lift}$	3.8668e-06	9.0549e-07	3.2832e-07

CHAPTER 6

CONCLUSIONS

The objective of this thesis is to extend to unstructured meshes the results of Peter et al., [6, 7], proving that the derivative of a functional output, e.g. lift or drag, with respect to the mesh nodes coordinates, can be used as a local indicator in a goal oriented mesh adaptation strategy.
The work has been divided in three main stages:

- Code development ($\approx 35\%$ of the total work)
- Code verification ($\approx 40\%$ of the total work)
- Creation of a complete mesh adaptation chain ($\approx 25\%$ of the total work)

The three stages have been described in the same order in chapter 3, 4 and 5 of the present thesis.
In the first part, the objective has been the extension of the *Opt* module of the code *elsA* [26] developed at ONERA, adding the capability to handle the computation of the Jacobian of the second order Roe flux [33, 34], equations (3.9), for Euler flows on unstructured meshes. Three different gradient computation methods [21] have been implemented: direct differentiation method, parameter mode adjoint method and mesh mode adjoint method. Even if only the latter has been directly used for mesh adaptation, being responsible for the computation of the $\frac{dJ}{dX}$ vector field, the implementation of the other two methods has been necessary for the verification of the accuracy of the code.
In the second part of the work, the objective has been the verification of the accuracy of the code implemented, using a NACA0012 transonic test case ($M = 0.85$, $AoA = 1°$). A verification chain has been adopted, the first step being the comparison of the output of the direct differentiation code with finite differences. Once the direct differentiation method has been proven

to be correctly coded, the accuracy of the adjoint vector has been easily checked using formula (4.6). At this point, it was possible to proceed with the verification of the mesh mode adjoint method. The main advantage of the verification chain procedure was that, being the adjoint vector and the partial derivatives of the function J already verified in the former steps, the only source of error in equation (3.35) remains the term $\frac{\partial R}{\partial X}$.
The main obstacle in the code verification work has been the difficulty in bringing to convergence the systems of equations of type (3.20) and (3.30). The problem has been solved only after coding in *elsA* an extension to unstructured meshes of the recursive projection method described in chapter 2.
In the last part of the work, the code developed has been used in order to build a complete mesh adaptation chain. This chain has the capability of taking as input an initial mesh in CGNS format, that is the format needed by *elsA* for computations on unstructured grids, and execute a full optimization step that produces an output mesh, again in CGNS format, ready for a possible successive step. The complete chain is illustrated in figure 5.1 and makes use of:

- *elsA* for steady state calculation and to compute the vector field $\frac{dJ}{dX}$
- Two sets of Python scripts, developed during the present thesis, one of which handles the computation of partial derivatives of lift and drag (one script of the set is given in appendix A) and the other one handling all the operations needed after the computation of $\frac{dJ}{dX}$ in order to create the output mesh in CGNS format (one script of the set is given in appendix B).
- The open source software MMG2D [41], for the actual creation of the output mesh from the initial one, based on the desired metric input.

The mesh adaptation chain has been tested for a NACA0012 geometry, with 3 different flow conditions, subsonic ($M = 0.5$, $AoA = 0°$), transonic ($M = 0.85$, $AoA = 2°$) and supersonic ($M = 1.5$, $AoA = 1°$).
In this case, the main obstacle has been the robustness of the implicit stage of the code *elsA* for steady state computations on unstructured meshes, that is particularly evident for flow conditions exhibiting strong shocks as in the transonic and supersonic case, when large variations of cells sizes are present in some areas of the grid, as resulting from the local mesh adaptation procedure. This issue limited considerably the automation of the mesh adaptation chain and, as a consequence, the number of adaptation steps carried out for each case (maximum 3 adaptation steps). In order to circumvent this problem, the parameter ψ representing the percentage of the spectral radius, below which the Harten correction is applied (see equation (3.6)), has been changed from one adaptation step to the other in order to

obtain convergence of the steady state computation.
Moreover, due to the lack of robustness of the implicit stage of *elsA* on unstructured meshes, it has not been possible to carry out steady state computations on a hierarchy of globally adapted meshes, especially for the finest meshes. The results obtained on the hierarchy of meshes would have been the basis for a quantitative assessment of the convergence of the values of lift and drag obtained through the local mesh adaptation chain.
A second relevant issue has been the solution of the system of equations (3.30), that has been successfull only when using the recursive projection algorithm. Especially in the subsonic case, the algorithm required a very large number of iterations, more than 100000, making the adaptation procedure slow and time consuming. This problem limited the number of adaptation steps performed in the subsonic case to only 2.
The low number of mesh adaptation steps completed, in combination with the difficulty to obtain steady state solutions on a hierarchy of globally adapted meshes, are the reason why a final analysis of the convergence of lift and drag is missing in the present report.
Although the difficulties encountered, the adaptation procedure produced satisfactory meshes at each adaptation step, for all the 3 cases analyzed. Moreover the average value of the local estimator θ has been proven to gradually decrease at each adaptation step, in all the cases considered, as already shown by Peter et al. in [6] and [7] for structured meshes.
The steps to be taken in the future for the improvement of the work done in the present thesis are the extension to 3D meshes of the code of the adjoint method in mesh mode and, most of all, the development of a more robust implicit stage for the convergence of the steady state computation on unstructured meshes in *elsA*.

BIBLIOGRAPHY

[1] R. Balasubramanian and J. C. Newman III. Comparison of adjoint-based and feature based grid adaption for functional outputs. *International Journal for Numerical Methods in Fluids*, 53:1541–1569, 2007.

[2] M. J. Marchant and N. P. Weatherill. Adaptive techniques for compressible inviscid flows. *Journal of Numerical Methods in Applied Mechanics and Engineering*, 106:83–106, 1993.

[3] N. P. Weatherill and M. J. Marchant. Grid adaptation using a distribution of sources applied to inviscid compressible flow simulations. *International Journal for Numerical Methods in Fluids*, 19:739–764, 1994.

[4] D. L. Marcum and N. P. Weatherill. A procedure for efficient generation of solution adapted unstructured grids. *Computer Methods in Applied Mechanics and Engineering*, 127:259–268, 1995.

[5] G. Warren, W. Anderson, J. Thomas, and S. Krist. Grid convergence for adaptive methods. In *AIAA Paper Series*, 91-1592, 1991.

[6] J. Peter, P. Trontin, and M. Nguyen-Dinh. Goal oriented mesh adaptation using total derivative of aerodynamic functions with respect to mesh coordinates. In *AIAA Paper Series*, 2011–30, 2011.

[7] J. Peter, P. Trontin, and M. Nguyen-Dinh. Goal oriented mesh adaptation using total derivative of aerodynamic functions with respect to mesh coordinates - With application to Euler flows. *Computer and Fluids*, 66:194–214, 2012.

[8] C. Johnson, R. Rannacher, and M. Boman. Numerics and hydrodynamics theory. *SIAM Journal pf Numerical Analysis*, 32:1058–1079, 1995.

[9] M. Giles, M. Larson, M. Levenstam, and E. Süli. Adaptive error control for finite element approximations of the lift and drag coefficients in viscous flow. Technical Report NA-97/06, Comlab, Oxford University, 1997.

[10] R. Becker and R. Rannacher. Weighted a posteriori error control in FE methods. In *Proceedings of ENUMATH-97*. Heidelberg: World Scientific Publishing, 1998.

[11] S. Prudhomme and J. Oden. On goal oriented error estimation for elliptic problems: application to the control of pointwise errors. *Computer Methods in Applied Mechanics and Engineering*, 176:313–331, 1999.

[12] R. Hartmann and P. Houston. Adaptive discontinuous Galerkin methods for the compressible euler equations. *Journal of Computational Physics*, 182(2):508–532, 2002.

[13] N.A. Pierce and M.B. Giles. Adjoint recovery of superconvergent functionals from PDE approximations. *SIAM Review*, 42(2):247–264, 2000.

[14] N.A. Pierce and M.B. Giles. Adjoint and defect error bounding and correction for functional estimates. In *AIAA Paper Series*, 2003–3846, 2003.

[15] D.A. Venditti and D.L. Darmofal. Adjoint error estimation and grid adaptation for functional outputs: application to quasi-one dimensional flow. *Journal of Computational Physics*, 164:204–227, 2000.

[16] D.A. Venditti and D.L. Darmofal. Grid adaptation for functional outputs: application to two-dimensional inviscid flow. *Journal of Computational Physics*, 176:40–69, 2002.

[17] D.A. Venditti and D.L. Darmofal. Anisotropic grid adaptation for functional outputs: application to viscous flow. *Journal of Computational Physics*, 187:22–46, 2003.

[18] R.P. Dwight. Goal-oriented mesh adaptation using a dissipation based error indicator. *International Journal for Numerical Methods in Fluids*, 56(8):1193–2000, 2007.

[19] R.P. Dwight. Heuristic a posteriori estimation of error due to dissipation in finite volume schemes and application to mesh adaptation. *Journal of Computational Physics*, 227:2845–2863, 2008.

[20] A. Jameson, W. Schmidt, and E. Turkel. Numerical solutions of the Euler equations by finite volume methods using Runge-Kutta time stepping schemes. In *AIAA Paper Series*, 1981–1259, 1981.

[21] J. Peter and R.P. Dwight. Numerical sensitivity analysis: a survey of approaches. *Computers and Fluids*, 39(3):373–391, 2010.

[22] E. Nielsen and M. Park. Using an adjoint approach to eliminate mesh sensitivities in aerodynamics design. *AIAA Journal*, 44(5):948–953, 2005.

[23] D.J. Mavripilis. Unstructured mesh generation and adaptivity. Technical Report 95/26, MS 132C, NASA Langley Research Center, Hampton, VA 23681-0001, 1995.

[24] D.J. Mavripilis. Unstructured grid techniques. *Annual Review of Fluid Mechanics*, 29:473–514, 1997.

[25] K. Fidkowski and D. Darmofal. Review of output-based error estimation and mesh adaptation in computational fluid dynamics. *AIAA journal*, 49(4):673–694, 2011.

[26] L. Cambier and J.P. Veuillot. Status of the elsA CFD software for flow simulation and multidisciplinary applications. In *AIAA Paper Series*, 2008-664, 2008.

[27] D. Destarac. Far-field/near field drag balance and applications of drag extraction in CFD. *VKI Lecture Series*, 2:3–7, 2003.

[28] G. Shroff and H. Keller. Stabilization of unstable procedures: the recursive projection method. *SIAM Journal on Numerical Analysis*, 39(4):1099–1120, 1993.

[29] B. van Leer. Flux vector splitting for the Euler equation. *In 8th International Conference on Numerical Methods in Fluid Dynamics*, pages 507–512, 1982.

[30] S. Godunov. A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics. *Matematicheskii Sbornik*, 89(3):271–306, 1959.

[31] C. Hirsch. *Numerical computation of internal and external flows*. Wiley, 1990.

[32] E. Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Springer, 2009.

[33] P. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics*, 43(2):357–372, 1981.

[34] PL Roe. Characteristic-based schemes for the euler equations. *Annual review of fluid mechanics*, 18(1):337–365, 1986.

[35] B. van Leer. Towards the ultimate conservative difference scheme. I. The quest of monotonicity. In *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics*, pages 163–168. Springer, 1973.

[36] B. van Leer. Towards the ultimate conservative difference scheme. II. Monotonicity and conservation combined in a second-order scheme. *Journal of computational physics*, 14(4):361–370, 1974.

[37] B. van Leer. Towards the ultimate conservative difference scheme. III. Upstream-centered finite-difference schemes for ideal compressible flow. *Journal of Computational Physics*, 23(3):263–275, 1977.

[38] B. van Leer. Towards the ultimate conservative difference scheme. IV. A new approach to numerical convection. *Journal of computational physics*, 23(3):276–299, 1977.

[39] B. van Leer. Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. *Journal of computational Physics*, 32(1):101–136, 1979.

[40] G. van Albada, B. van Leer, and W. Roberts. A comparative study of computational methods in cosmic gas dynamics. *Astronomy and Astrophysics*, 108:76–84, 1982.

[41] C. Dobrzynski. Mmg3d: User guide. Technical Report 422, INRIA, Bordeaux, March 2012. Project-Team Bacchus.

APPENDIX A

GOAL FUNCTION PARTIAL DERIVATIVES

The following piece of code is an extract of the Python set of scripts that has been created in order to compute partial derivatives of lift and drag. The script presented computes the partial derivatives of the aerodynamic functions of interest with respect to the flow variables at the boundaries W_b.

```
import Converter.PyTree as C
import Converter.Internal as Internal
import numpy as np
from math import *
import os
t = C.convertFile2PyTree('../mesh/initial_mesh_D_NGON.adf')

alpha   = 0.0 * pi /180.

# Read file with boundary cells indices and store bnd cells indices in an array
bncells = np.loadtxt('bndcells.dat',dtype = int)

#Read file with boundary nodes list
bnnodes = np.loadtxt('bndnodes.dat',dtype = int)

#Compute pressure at boundary cells

#Extract quantities of interest from cgns tree

#conservative variables
consvar = Internal.getNodesFromName(t, 'O2I_sol')
wcons1 = consvar[0][2][1][1]
wcons2 = consvar[0][2][2][1]
wcons3 = consvar[0][2][3][1]
wcons4 = consvar[0][2][4][1]
wcons5 = consvar[0][2][5][1]

#nodes coordinates
gridcoord = Internal.getNodesFromName(t, 'GridCoordinates')
xcoord = gridcoord[0][2][0][1]
ycoord = gridcoord[0][2][1][1]

#penta connectivity
connect =  Internal.getNodesFromName(t, 'PENTA_6')
penta = connect[0][2][0][1]
penta = penta.reshape(-1,6)


gam = 1.4

gam1 = gam - 1.
```

```
bncells_m1 = bncells -1

nb_bsurfs = len(bncells)

dddwb1 = []
dddwb2 = []
dddwb3 = []
dddwb4 = []
dddwb5 = []

for ind in range (nb_bsurfs):

    n1 = bnnodes[ind][0]
    n2 = bnnodes[ind][1]
    n3 = bnnodes[ind][2]
    n4 = bnnodes[ind][3]

    sx = 0.5*((ycoord[n3-1] - ycoord[n1-1]) - (ycoord[n4-1] - ycoord[n2-1]))
    sy = 0.5*((xcoord[n4-1] - xcoord[n2-1]) - (xcoord[n3-1] - xcoord[n1-1]))

    # Check surface orientation (sign of scalar product with one of edges of the neighbouring
        boundary cell)
    ind_glo = bncells_m1[ind]
    penta1 = penta[ind_glo][0]
    penta2 = penta[ind_glo][1]
    penta3 = penta[ind_glo][2]

    if (penta1 != n1 and penta1 != n2):
        vertex = penta1
    elif (penta2 != n1 and penta2 != n2):
        vertex = penta2
    else:
        vertex = penta3

    #scalar product
    edgex = xcoord[vertex -1] - xcoord[n1-1]
    edgey = ycoord[vertex -1] - ycoord[n1-1]

    scprod = sx * edgex + sy *edgey

    # normal must be pointing inward because we're considering pressure forces
    if scprod > 0 :
        sx = -sx
        sy = -sy


    surf = (sx**2 + sy**2)**0.5
    nx = sx/surf
    ny = sy/surf

    u  = wcons2[ind_glo]/wcons1[ind_glo]
    v  = wcons3[ind_glo]/wcons1[ind_glo]
    ub = u - u*nx*nx - v*nx*ny
    vb = v - u*nx*ny - v*ny*ny

    ee = 0.5*(ub*ub+vb*vb)

    #pressure derivatives with respect to cons variables at the boundary
    dpdw1 = gam1 * ee
    dpdw2 = - (gam1 *ub)
    dpdw3 = - (gam1 *vb)
    dpdw4 = 0
    dpdw5 = gam1

    dddwb1.append(dpdw1*(sx*cos(alpha) + sy*sin(alpha)))
    dddwb2.append(dpdw2*(sx*cos(alpha) + sy*sin(alpha)))
    dddwb3.append(dpdw3*(sx*cos(alpha) + sy*sin(alpha)))
    dddwb4.append(dpdw4*(sx*cos(alpha) + sy*sin(alpha)))
    dddwb5.append(dpdw5*(sx*cos(alpha) + sy*sin(alpha)))

dddwb1 = np.array(dddwb1)
dddwb2 = np.array(dddwb2)
dddwb3 = np.array(dddwb3)
dddwb4 = np.array(dddwb4)
dddwb5 = np.array(dddwb5)


#Write results to file in standard format: index, idom, i, j, k, dfdw1,dfdw2,dfdw3,dfdw4,dfdw5


##########################DRAG#############################
```

```
h1 = open("header1.dat","a")
h1.write('TITLE     = "drag sensitivities"\n')
h1.write('VARIABLES = "index" "idom" "i" "j" "k" "dfdu1" "dfdu2" "dfdu3" "dfdu4" "dfdu5"\n')
h1.write('ZONE T=" ",I=%s, J=1,  K=1, F=BLOCK\n' % str(nb_bsurfs))
h1.close()
f5 = open('dfunctiondwb_001.dat', 'w')

#write index
index= range(nb_bsurfs)
index = np.array(index)
block = index +1
width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f5.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block[indx0:indx1]))+"\n")

#write domain number
idom = [1] * nb_bsurfs
block = idom
width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f5.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block[indx0:indx1]))+"\n")

#write index i
bnfaces = np.loadtxt('bndfaces.dat',dtype = int)

block = bnfaces
width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f5.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block[indx0:indx1]))+"\n")

#write index j =  -1
block = [-1] *nb_bsurfs
width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f5.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block[indx0:indx1]))+"\n")
#write index k =  -1
block = [-1] *nb_bsurfs
width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f5.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block[indx0:indx1]))+"\n")

#write to file drag derivative with respect to wb1
block=dddwb1

width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f5.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block[indx0:indx1]))+"\n")

#write to file drag derivative with respect to wb2
block=dddwb2

width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f5.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block[indx0:indx1]))+"\n")

#write to file drag derivative with respect to wb3
```

```
block=dddwb3

width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f5.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block[indx0:indx1]))+"\n")


#write to file drag derivative with respect to wb4
block=dddwb4

width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f5.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block[indx0:indx1]))+"\n")

#write to file drag derivative with respect to wb5
block=dddwb5

width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f5.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block[indx0:indx1]))+"\n")

f5.close()

os.system("cat header1.dat dfunctiondwb_001.dat > ../elsainfiles/drag/dfunctiondwb_001.tp")
```

APPENDIX B

PYTHON PACKAGE FOR MESH ADAPTATION

The following code is an extract of the Python package created in order to manage the operations that go from the computations of the $\frac{dJ}{dX}$ vector field in *elsA* to the creation of the adapted mesh through MMG2D. It is a necessary bridge between *elsA* and MMG2D.
In the specific part of code presented, the user computes the estimator for mesh adaptation starting from the raw $\frac{dJ}{dX}$ vector field computed in *elsA* and defines the target metric, that will be one of the inputs required by MMG2D.

```
import numpy as np
from math import *
import Converter.PyTree as C
import Converter.Internal as Internal
import os


################################################
def belong(var,array):
    length = len (array)
    if length==0:
        return False
    count = 0
    for j in range(length):
        if (var==array[j]):
            return True
        else:
            if j==length-1:
                return False

#######################################################

######Get relevant arrays from CGNS tree
t = C.convertFile2PyTree('../mesh/initial_mesh_D_NGON.adf')


# Total number of nodes
dom = Internal.getNodesFromName(t, 'dom1')
nnodes = dom[0][1][0][0]
ncell=  dom[0][1][0][1]

coords = Internal.getNodesFromName(t, 'GridCoordinates')
x = coords[0][2][0][1]
y = coords[0][2][1][1]
```

```
connect = Internal.getNodesFromName(t, 'PENTA_6')
penta =  connect[0][2][0][1]
penta= penta.reshape(-1,6)
triangles =  penta[:,0:3]


####Store indices of corners (in this case only trailing edge)
corners = []

#find index of trailing edge
for i in range (nnodes):
    if (x[i]==1. and y[i]==0.):
        ind_cnr = i
        break
corners.append([i])
corners = np.array(corners)
corners = corners[0]
nb_cnr = len(corners)

####Read PdDdX file
f11 = open('../elsaoutfiles/drag/PDDDX.tp', 'r')
l11 = f11.readlines()
f11.close()
emptstr1=''
str11=emptstr1.join(l11[:])
emptstr21 = ' '
str21=emptstr21.join(str11.split('\n'))
array1=np.fromstring(str21, dtype=float, sep=' ')
pdddx = array1[0:nnodes]
pdddy = array1[nnodes:2*nnodes]



####Compute modulus of PdJdX only on z=0 plane
nnodes_plane0 = nnodes/2
mod_pdddx = [0.] *nnodes_plane0

for i in range(nnodes_plane0):
    mod_pdddx[i] = sqrt((pdddx[i])**2 +(pdddy[i])**2 )



####Compute characteristic length for each node: average between min and max distance to
     neighbouring nodes
char_length = [0.] * nnodes_plane0
char_length_min = [1000.] * nnodes_plane0
char_length_max = [0.] * nnodes_plane0

for j in range(ncell):
    indn1 = triangles[j][0] -1
    indn2 = triangles[j][1] -1
    indn3 = triangles[j][2] -1
    dist1_2 = sqrt((x[indn1] - x[indn2])**2 +(y[indn1] - y[indn2])**2 )
    dist1_3 = sqrt((x[indn1] - x[indn3])**2 +(y[indn1] - y[indn3])**2 )
    dist2_3 = sqrt((x[indn2] - x[indn3])**2 +(y[indn2] - y[indn3])**2 )
    char_length_min[indn1] =min(min(dist1_2,dist1_3),char_length_min[indn1])
    char_length_min[indn2] =min(min(dist1_2,dist2_3),char_length_min[indn2])
    char_length_min[indn3] =min(min(dist1_3,dist2_3),char_length_min[indn3])
    char_length_max[indn1] =max(max(dist1_2,dist1_3),char_length_max[indn1])
    char_length_max[indn2] =max(max(dist1_2,dist2_3),char_length_max[indn2])
    char_length_max[indn3] =max(max(dist1_3,dist2_3),char_length_max[indn3])

for j in range (nnodes_plane0):
    char_length[j] = 0.5*(char_length_min[j] + char_length_max[j])



#####Compute local estimator in each node: product between minimum distance to neighbouring
     nodes and modulus of PdJdX
estimator_D = [0.]* nnodes_plane0

for k in range(nnodes_plane0):

    estimator_D[k] =0.5*char_length_min[k] * mod_pdddx[k]



######For corners define estimator as average value of the estimator in neighbouring nodes
for i in range (nb_cnr):
    divby = 0
    for j in range(ncell):
        indn1 = triangles[j][0] -1
        indn2 = triangles[j][1] -1
        indn3 = triangles[j][2] -1
```

```
        if (indn1 == corners[i]):
            estimator_D[i]=estimator_D[i]+estimator_D[indn2] + estimator_D[indn3]
            divby = divby +2
        elif (indn2 == corners[i]):
            estimator_D[i]=estimator_D[i]+estimator_D[indn1] + estimator_D[indn3]
            divby = divby +2
        if (indn3 == corners[i]):
            estimator_D[i]=estimator_D[i]+estimator_D[indn1] + estimator_D[indn2]
            divby = divby +2
    estimator_D[i] = estimator_D[i]/divby


####Compute spatial average of the estimator
estimator_D_avg = [0.]*nnodes_plane0
included = [[] for i in range(nnodes_plane0)]
for j in range(ncell):
    indn1 = triangles[j][0] -1
    indn2 = triangles[j][1] -1
    indn3 = triangles[j][2] -1
    if not belong(indn1,included[indn1]):
        estimator_D_avg[indn1] = estimator_D_avg[indn1] + estimator_D[indn1]
        included[indn1].append(indn1)
    if not belong(indn2,included[indn1]):
        estimator_D_avg[indn1] = estimator_D_avg[indn1] + estimator_D[indn2]
        included[indn1].append(indn2)
    if not belong(indn3,included[indn1]):
        estimator_D_avg[indn1] = estimator_D_avg[indn1] + estimator_D[indn3]
        included[indn1].append(indn3)

    if not belong(indn1,included[indn2]):
        estimator_D_avg[indn2] = estimator_D_avg[indn2] + estimator_D[indn1]
        included[indn2].append(indn1)
    if not belong(indn2,included[indn2]):
        estimator_D_avg[indn2] = estimator_D_avg[indn2] + estimator_D[indn2]
        included[indn2].append(indn2)
    if not belong(indn3,included[indn2]):
        estimator_D_avg[indn2] = estimator_D_avg[indn2] + estimator_D[indn3]
        included[indn2].append(indn3)

    if not belong(indn1,included[indn3]):
        estimator_D_avg[indn3] = estimator_D_avg[indn3] + estimator_D[indn1]
        included[indn3].append(indn1)
    if not belong(indn2,included[indn3]):
        estimator_D_avg[indn3] = estimator_D_avg[indn3] + estimator_D[indn2]
        included[indn3].append(indn2)
    if not belong(indn3,included[indn3]):
        estimator_D_avg[indn3] = estimator_D_avg[indn3] + estimator_D[indn3]
        included[indn3].append(indn3)

for j in range(nnodes_plane0):
    if estimator_D_avg[j]!=0.:
        estimator_D_avg[j]=estimator_D_avg[j]/(len(included[j]))


###Write to file
f1 = open('../elsaoutfiles/drag/estimator_D_avg.tp', 'w')
block_1=np.array([estimator_D_avg,estimator_D_avg])
block_1 = block_1.reshape(1,-1)
block_1 = block_1[0]

width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block_1)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f1.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block_1[indx0:indx1]))+"\n")



###Write to file
f100 = open('../elsaoutfiles/drag/logestimator_D_avg.tp', 'w')
block_100 = [0.] * len(block_1)
for i in range(len(block_1)):
    if block_1[i] != 0.:
        block_100[i] = -log10(block_1[i])

width, fmt = 6, "%14.7E"
indx0, indx1, indxm = -width, 0, len(block_100)
while indx1 < indxm:
    indx0 = min(indx0 + width, indxm)
    indx1 = min(indx1 + width, indxm)
    f100.write(reduce(lambda x,y:"%s %s"%(x,y),
                    map(lambda x:fmt%x, block_100[indx0:indx1]))+"\n")
```

```
#####Identify possible fake nodes (nodes used only to define geometry but not belonging to any
      cell)
real_nodes = []
newind = [-1] * nnodes_plane0
count = 0

for k in range(nnodes_plane0):

    if (mod_pdddx[k] != 0.):     ### or  (mod_pdldx[k] != 0.) would be the same
        count = count + 1
        real_nodes.append(k)
        newind[k] = count
    else:
        for i in range(nb_cnr):
            if k==corners[i]:
                count = count + 1
                real_nodes.append(k)
                newind[k] = count

real_nodes = np.array(real_nodes)
ngeomnodes = nnodes_plane0 - len(real_nodes)
print ngeomnodes, " geometric nodes detected"


#####Reduce arrays of interest to include real nodes only
estimator_D_avg_rn = []
char_length_rn = []
for l in real_nodes:
    estimator_D_avg_rn.append(estimator_D_avg[l])
    char_length_rn.append(char_length[l])

estimator_D_avg_rn= np.array(estimator_D_avg_rn)
char_length_rn= np.array(char_length_rn)


#####Define 'objective metric': desired characteristic length of the output mesh

###For DRAG
metric_D = [0.] * len(real_nodes)


tol = 10.**(-8.)
nbadp = 0
for l in range (len(real_nodes)):

    if (estimator_D_avg_rn[l] < tol):

        metric_D[l] = char_length_rn[l]
    else:
        split_factor = sqrt(2**(log10(estimator_D_avg_rn[l]) - log10(tol)))
        metric_D[l] = char_length_rn[l]*(1./split_factor)
        nbadp = nbadp +1

print "Estimator_D over tolerance in ", nbadp, " nodes"

#####Build _D.sol file
hsol_D = open("header_sol_D.dat","a")
nvert = len(real_nodes)
hsol_D.write("MeshVersionFormatted\n1\nDimension\n2\nSolAtVertices\n%s\n1 1\n" % str(nvert)  )
hsol_D.close()
np.savetxt('metric_D.dat',metric_D, fmt=['%.15e'])

os.system("cat header_sol_D.dat metric_D.dat > inputfile_D.sol")



#####Build .mesh file
h1 = open("header1.dat","a")

h1.write("MeshVersionFormatted\n1\nDimension\n2\nVertices\n%s\n" % str(nvert)  )
h1.close()

listtowrite=[]
ref = 1

for i in real_nodes:
    listtowrite.append([x[i],y[i],ref])
np.savetxt('vertices.dat',listtowrite, fmt=['%.15e','%.15e','%d'])

h2 = open("header2.dat","a")
```

```
h2.write("\nTriangles\n%s\n" % str(ncell)  )
h2.close()

connect = Internal.getNodesFromName(t, 'PENTA_6')
penta =  connect[0][2][0][1]
penta= penta.reshape(-1,6)
triangles =  penta[:,0:3]
listtowrite2 = []
for i in range(ncell):
    listtowrite2.append([newind[triangles[i][0] - 1], newind[triangles[i][1] - 1],newind[
        triangles[i][2] - 1], ref ])

np.savetxt('triangles.dat',listtowrite2, fmt=['%d','%d','%d','%d'])

os.system("cat header1.dat vertices.dat header2.dat triangles.dat > inputfile_D.mesh")

os.system("rm header* triangles.dat vertices.dat metric*.dat")
```

APPENDIX C

ADJOINT METHOD SUBROUTINE

The Fortran code given in the following, is used for the computation of the adjoint vector in parameter mode.
The operation that is actually carried out here, is the second of the two loops described in chapter 3, in the section related to the adjoint method in parameter mode. The subroutine presented here is specific for interior interfaces. Different subroutines have been used, instead, to manage the operations on boundary interfaces.

```
      gam1    = gam - ONE
      gam1_1  = ONE/gam1

      sen = ZERO

      DO iface=0,nfaceint-1

        iL  = indicVolL(iface)
        iR  = indicVolR(iface)

        surf = snl(iface)
        snx  = slx(iface)
        sny  = sly(iface)
        snz  = slz(iface)

        nx  = snx/(surf+E_MIN_SURFACE)
        ny  = sny/(surf+E_MIN_SURFACE)
        nz  = snz/(surf+E_MIN_SURFACE)

C**********************************************************************
C    I     calculation of the left and right states of the problem
C**********************************************************************
C---------
        r1  = wcons1(iL)
        u1  = wcons2(iL)/wcons1(iL)
        v1  = wcons3(iL)/wcons1(iL)
        w1  = wcons4(iL)/wcons1(iL)
        ee1 = HALF*(u1**2+v1**2+w1**2)
        p1 = gam1*(wcons5(iL)-r1*ee1)
C---------
        r2  = wcons1(iR)
        u2  = wcons2(iR)/wcons1(iR)
        v2  = wcons3(iR)/wcons1(iR)
        w2  = wcons4(iR)/wcons1(iR)
        ee2 = HALF*(u2**2+v2**2+w2**2)
        p2  = gam1*(wcons5(iR)-r2*ee2)
C     vector from the left cell center to the right one
```

```
C
          dx = cellcenter(iR,1) - cellcenter(iL,1)
          dy = cellcenter(iR,2) - cellcenter(iL,2)
          dz = cellcenter(iR,3) - cellcenter(iL,3)
C
C     Upwind Left cell contribution to limiter (gradL.C_L C_R) -> ul
C
          ulr = grad_r(iL         ) * dx +
     &          grad_r(iL+  ncell) * dy +
     &          grad_r(iL+2*ncell) * dz
          ulu = grad_u(iL         ) * dx +
     &          grad_u(iL+  ncell) * dy +
     &          grad_u(iL+2*ncell) * dz
          ulv = grad_v(iL         ) * dx +
     &          grad_v(iL+ncell   ) * dy +
     &          grad_v(iL+2*ncell) * dz
          ulw = grad_w(iL         ) * dx +
     &          grad_w(iL+ncell   ) * dy +
     &          grad_w(iL+2*ncell) * dz
          ulp = grad_p(iL         ) * dx +
     &          grad_p(iL+  ncell) * dy +
     &          grad_p(iL+2*ncell) * dz
C
C     Upwind Right cell contribution to limiter (gradR.C_L C_R) -> ur
C
          urr = grad_r(iR         ) * dx +
     &          grad_r(iR+  ncell) * dy +
     &          grad_r(iR+2*ncell) * dz
          uru = grad_u(iR         ) * dx +
     &          grad_u(iR+  ncell) * dy +
     &          grad_u(iR+2*ncell) * dz
          urv = grad_v(iR         ) * dx +
     &          grad_v(iR+  ncell) * dy +
     &          grad_v(iR+2*ncell) * dz
          urw = grad_w(iR         ) * dx +
     &          grad_w(iR+ncell   ) * dy +
     &          grad_w(iR+2*ncell) * dz
          urp = grad_p(iR         ) * dx +
     &          grad_p(iR+  ncell) * dy +
     &          grad_p(iR+2*ncell) * dz

C*************************************************************
C     I      calculation of interface states and Roe average state
C*************************************************************
C-----------------------------------------
          rg  =  r1 + HALF*vanalbada(ulr,r2-r1,epsilon)
          ug  =  u1 + HALF*vanalbada(ulu,u2-u1,epsilon)
          vg  =  v1 + HALF*vanalbada(ulv,v2-v1,epsilon)
          wg  =  w1 + HALF*vanalbada(ulw,w2-w1,epsilon)
          pg  =  p1 + HALF*vanalbada(ulp,p2-p1,epsilon)
          eeg =  HALF*(ug*ug+vg*vg+wg*wg)
          hg  =  gam*pg*gam1_1/rg + eeg
          vng =  ug*nx+vg*ny+wg*nz

          rd  =  r2 - HALF*vanalbada(urr,r2-r1,epsilon)
          ud  =  u2 - HALF*vanalbada(uru,u2-u1,epsilon)
          vd  =  v2 - HALF*vanalbada(urv,v2-v1,epsilon)
          wd  =  w2 - HALF*vanalbada(urw,w2-w1,epsilon)
          pd  =  p2 - HALF*vanalbada(urp,p2-p1,epsilon)
          eed =  HALF*(ud*ud+vd*vd+wd*wd)
          hd  =  gam*pd*gam1_1/rd + eed
          vnd =  ud*nx+vd*ny+wd*nz

C-----------------------------------------
          r  = SQRT(rd/rg)
          rr = SQRT(rd*rg)

          uu = (ud*r+ug)/(r+ONE)
          vv = (vd*r+vg)/(r+ONE)
          ww = (wd*r+wg)/(r+ONE)
          ee = HALF*(uu**2+vv**2+ww**2)
          hh = (hd*r+hg)/(r+ONE)
          cc = SQRT(gam1*(hh-ee))
          vn = uu*nx+vv*ny+ww*nz

C*************************************************************************
C linearization of the double of Roe flux with respect to primitive variables is
C  done in an included file.
C   input variables of file :
C    mean state       : uu, vv, ww, cc, hh, vn, rr
C    left/right state : rg, ug, vg, wg, vng, eeg, pg, rd, ud, vd, wd, vnd, eed, pd
C    geometry         : nx,ny,nz,snx,sny,snz

#include "FxcHartenF.h"
```

```
#include "dRoedgddF.h"

C   output variables of file : dpg11, dpg12, ... dpg55, dpd11, dpd12, ... dpd55
C   caution factor TWO (the double of ...)

C************************************************************************
C     limiter derivatives
C---------------------------------

        dvaL_r2 = dvanalb_db(ulr,r2-r1,epsilon)
        dvaL_u2 = dvanalb_db(ulu,u2-u1,epsilon)
        dvaL_v2 = dvanalb_db(ulv,v2-v1,epsilon)
        dvaL_w2 = dvanalb_db(ulw,w2-w1,epsilon)
        dvaL_p2 = dvanalb_db(ulp,p2-p1,epsilon)

        dvaR_r2 = dvanalb_db(urr,r2-r1,epsilon)
        dvaR_u2 = dvanalb_db(uru,u2-u1,epsilon)
        dvaR_v2 = dvanalb_db(urv,v2-v1,epsilon)
        dvaR_w2 = dvanalb_db(urw,w2-w1,epsilon)
        dvaR_p2 = dvanalb_db(urp,p2-p1,epsilon)

C************************************************************************
C   BUILD RHS  (lambda^T * dR/dW)

C   VA = (a^2 * b + a * b^2)/(a^2 + b^2)
C
C   Pg = P(left)  + 1/2 * VA(grad(left).D , P(right)-P(left))
C   Pd = P(right) - 1/2 * VA(grad(right).D , P(right)-P(left))
C
C   Flux = Froe(Pg,Pd) = Froe(P(left)  + 1/2 * VA(grad(left).D , P(right)-P(left)),
C                             P(right) - 1/2 * VA(grad(right).D , P(right)-P(left)) )
C
C   rhsnwt(R) += [lambda(R) - lambda(L)] * (dFlux/dPd * dPd/dPR + dFlux/dPg * dPg/dPR) +
C           ---- index j refers to the remaining
C                sum_j [(lambda(L) +/- lambda(j))*dFlux(j+1/2)/dP(j+1/2)*dP(j+1/2)/dP(R)]
C               neighbours of the left cell (apart from right cell
c
C   rhsnwt(L) += [lambda(R) - lambda(L)] * (dFlux/dPd * dPd/dPL + dFlux/dPg * dPg/dPL) +
C           ---- index k refers to the remaining
C                sum_k [(lambda(R) +/- lambda(k))*dFlux(k+1/2)/dP(k+1/2)*dP(k+1/2)/dP(L)]
C               neighbours of the right cell (apart from left cell)
C************************************************************************

C-------------------------------------------------------------------------
C   STEP 1 - DERIVATIVES WITH RESPECT TO GRADIENTS ARE NOT CONSIDERED IN THIS STEP
C
C   rhsnwt(R) += [lambda(R) - lambda(L)] * [dFlux/dPd * dPd/dP(R) + dFlux/dPg * dPg/dP(R)] *
     dP(R)/dW(R)
C   rhsnwt(L) += [lambda(R) - lambda(L)] * [dFlux/dPd * dPd/dP(L) + dFlux/dPg * dPg/dP(L)] *
     dP(L)/dW(L)
C
C-------------------------------------------------------------------------

        delta1 = HALF*(lmb1(iR) - lmb1(iL))
        delta2 = HALF*(lmb2(iR) - lmb2(iL))
        delta3 = HALF*(lmb3(iR) - lmb3(iL))
        delta4 = HALF*(lmb4(iR) - lmb4(iL))
        delta5 = HALF*(lmb5(iR) - lmb5(iL))

C  Compute  (lambdaR - lambdaL)*dF/dPg
        yg1 =delta1*dpg11+delta2*dpg21+delta3*dpg31+delta4*dpg41
     1      +delta5*dpg51
        yg2 =delta1*dpg12+delta2*dpg22+delta3*dpg32+delta4*dpg42
     1      +delta5*dpg52
        yg3 =delta1*dpg13+delta2*dpg23+delta3*dpg33+delta4*dpg43
     1      +delta5*dpg53
        yg4 =delta1*dpg14+delta2*dpg24+delta3*dpg34+delta4*dpg44
     1      +delta5*dpg54
        yg5 =delta1*dpg15+delta2*dpg25+delta3*dpg35+delta4*dpg45
     1      +delta5*dpg55

C  Compute  (lambdaR - lambdaL)*dF/dPd
        yd1 =delta1*dpd11+delta2*dpd21+delta3*dpd31+delta4*dpd41
     1      +delta5*dpd51
        yd2 =delta1*dpd12+delta2*dpd22+delta3*dpd32+delta4*dpd42
     1      +delta5*dpd52
        yd3 =delta1*dpd13+delta2*dpd23+delta3*dpd33+delta4*dpd43
     1      +delta5*dpd53
        yd4 =delta1*dpd14+delta2*dpd24+delta3*dpd34+delta4*dpd44
     1      +delta5*dpd54
        yd5 =delta1*dpd15+delta2*dpd25+delta3*dpd35+delta4*dpd45
     1      +delta5*dpd55
```

```
C  Compute   [(lambdaR - lambdaL)*dF/dPd*dPd/dP(rc) + (lambdaR - lambdaL)*dF/dPg*dPg/dP(rc)]
        step11R = yd1*(1-HALF*dvaR_r2) + yg1*(HALF*dvaL_r2)
        step12R = yd2*(1-HALF*dvaR_u2) + yg2*(HALF*dvaL_u2)
        step13R = yd3*(1-HALF*dvaR_v2) + yg3*(HALF*dvaL_v2)
        step14R = yd4*(1-HALF*dvaR_w2) + yg4*(HALF*dvaL_w2)
        step15R = yd5*(1-HALF*dvaR_p2) + yg5*(HALF*dvaL_p2)

C  Compute   [(lambdaR - lambdaL)*dF/dPd*dPd/dP(lc) + (lambdaR - lambdaL)*dF/dPg*dPg/dP(lc)]
        step11L = yg1*(1-HALF*dvaL_r2) + yd1*(HALF*dvaR_r2)
        step12L = yg2*(1-HALF*dvaL_u2) + yd2*(HALF*dvaR_u2)
        step13L = yg3*(1-HALF*dvaL_v2) + yd3*(HALF*dvaR_v2)
        step14L = yg4*(1-HALF*dvaL_w2) + yd4*(HALF*dvaR_w2)
        step15L = yg5*(1-HALF*dvaL_p2) + yd5*(HALF*dvaR_p2)

C-------------------------------------------------------------------------------
C   STEP 2 - ADD DERIVATIVES WITH RESPECT TO GRADIENTS
C
C   VA(a,b) --->  a = [gradP . (dx,dy,dz)]   b = [P(R) - P(L)]
C
C  lmbdmRdGFB = - Sigma lambda (dR/dGrad) (dGrad/dGrenFluxBalOfGrad)
C                                          = 1/vol
C this cell-centred line-vector (15 components) needs to be multiplied
C  by  (dGrenFluxBalOfGrad/dP)(dP/dW)
C  In all conntributioms to (dGrenFluxBalOfGrad/dP) except at boundary interfaces
C   factor HALF coming from the mean of cell-centred values
C
C   rhsnwt(L) += HALF * [ lmbdmRdGFB(L) - lmbdmRdGFB(R) . S] * dP/dW(L)
C   rhsnwt(R) += HALF * [ lmbdmRdGFB(L) - lmbdmRdGFB(R) . S] * dP/dW(R)
C
C-------------------------------------------------------------------------------
C
        step21 =HALF*(lmbdmRdGFB1(iL,1)-lmbdmRdGFB1(iR,1))*snx
     1         +HALF*(lmbdmRdGFB1(iL,2)-lmbdmRdGFB1(iR,2))*sny
     1         +HALF*(lmbdmRdGFB1(iL,3)-lmbdmRdGFB1(iR,3))*snz
        step22 =HALF*(lmbdmRdGFB2(iL,1)-lmbdmRdGFB2(iR,1))*snx
     1         +HALF*(lmbdmRdGFB2(iL,2)-lmbdmRdGFB2(iR,2))*sny
     1         +HALF*(lmbdmRdGFB2(iL,3)-lmbdmRdGFB2(iR,3))*snz
        step23 =HALF*(lmbdmRdGFB3(iL,1)-lmbdmRdGFB3(iR,1))*snx
     1         +HALF*(lmbdmRdGFB3(iL,2)-lmbdmRdGFB3(iR,2))*sny
     1         +HALF*(lmbdmRdGFB3(iL,3)-lmbdmRdGFB3(iR,3))*snz
        step24 =HALF*(lmbdmRdGFB4(iL,1)-lmbdmRdGFB4(iR,1))*snx
     1         +HALF*(lmbdmRdGFB4(iL,2)-lmbdmRdGFB4(iR,2))*sny
     1         +HALF*(lmbdmRdGFB4(iL,3)-lmbdmRdGFB4(iR,3))*snz
        step25 =HALF*(lmbdmRdGFB5(iL,1)-lmbdmRdGFB5(iR,1))*snx
     1         +HALF*(lmbdmRdGFB5(iL,2)-lmbdmRdGFB5(iR,2))*sny
     1         +HALF*(lmbdmRdGFB5(iL,3)-lmbdmRdGFB5(iR,3))*snz

C-----------------------------------------------------------------------
C     multiply contributions from 1st and 2nd step by dP/dW
C-----------------------------------------------------------------------

C Right cell
        m1  = step11R + step21
        m2  = step12R + step22
        m3  = step13R + step23
        m4  = step14R + step24
        m5  = step15R + step25

        addrhs1R  =  m1 - ONE/r2*(m2*u2+m3*v2+m4*w2) + m5*gam1*ee2
        addrhs2R  =  m2/r2 - m5*gam1*u2
        addrhs3R  =  m3/r2 - m5*gam1*v2
        addrhs4R  =  m4/r2 - m5*gam1*w2
        addrhs5R  =  m5*gam1

C Left cell
        t1  = step11L + step21
        t2  = step12L + step22
        t3  = step13L + step23
        t4  = step14L + step24
        t5  = step15L + step25

        addrhs1L  =  t1 - ONE/r1*(t2*u1+t3*v1+t4*w1) + t5*gam1*ee1
        addrhs2L  =  t2/r1 - t5*gam1*u1
        addrhs3L  =  t3/r1 - t5*gam1*v1
        addrhs4L  =  t4/r1 - t5*gam1*w1
        addrhs5L  =  t5*gam1

C-------------------------------------------------------------------------------
C     add contribution to left and right cell
C-------------------------------------------------------------------------------
        rhsnt1(iL) = rhsnt1(iL) + addrhs1L
        rhsnt2(iL) = rhsnt2(iL) + addrhs2L
        rhsnt3(iL) = rhsnt3(iL) + addrhs3L
        rhsnt4(iL) = rhsnt4(iL) + addrhs4L
```

```
  rhsnt5(iL) = rhsnt5(iL) + addrhs5L

  rhsnt1(iR) = rhsnt1(iR) + addrhs1R
  rhsnt2(iR) = rhsnt2(iR) + addrhs2R
  rhsnt3(iR) = rhsnt3(iR) + addrhs3R
  rhsnt4(iR) = rhsnt4(iR) + addrhs4R
  rhsnt5(iR) = rhsnt5(iR) + addrhs5R

ENDDO
END
```