Scuola Normale Superiore di Pisa

Classe di Scienze

Université de Nice - Sophia Antipolis

Sensitivity Evaluation in Aerodynamic Optimal Design

Ph.D. Thesis

Candidate

Massimiliano Martinelli

Supervisors

François Beux
Alain Dervieux

Pisa, Italy - 2007

Contents

Context of the PhD thesis

The present PhD thesis has been done in the context of a cotutelle agreement between the Scuola Normale Superiore di Pisa and the university of Nice - Sophia Antipolis[1] for the award of a single doctoral degree in Mathematics jointly-badged by the two institutions (a degree of "docteur en Sciences/spécialité: Mathématiques" for the University of Nice - Sophia Antipolis and a "diploma di perfezionamento in Matematica per la Tecnologia e l'Industria" for the Scuola Normale Superiore di Pisa).

[1]In the framework of the french-italian University, and also, for the italian party, under the terms of the law 26/5/2000, n. 161.

Acknowledgements

I am very grateful to F. Beux for his help and suggestions in the supervision of my research, and in particular for his contribution to Chapter 4 of the thesis. I am also grateful to A. Dervieux for the opportunity he made for me to work at the research unit INRIA Sophia-Antipolis, and for his supervision of my thesis. Thanks also to L. Hascoët and V. Pascual for the development of the Automatic Differentiation tool TAPENADE and for their quick responses to my requests for explanations/modifications of TAPENADE. Thanks to R. Duvigneau for the explanations about the flow solver and for his useful suggestions.

Finally, I am extremely grateful to my family and friends for their assistance, support and encouragement throughout my PhD

This work was partially supported by the project NODESIM-CFD "Non-Deterministic Simulation for CFD-based Design Methodologies" funded by the European Community represented by the CEC, Research Directorate-General, in the 6th Framework Programme, under Contract No. AST5-CT-2006-030959.

Sommario

La possibilità di poter calcolare le derivate prime e seconde di funzionali soggetti a vincoli di uguaglianza dati da equazioni di stato (ed in particolare da sistemi di equazioni alle derivate parziali non lineari) permette l'utilizzo di tecniche efficienti per la soluzione di vari problemi di tipo industriale. Tra le applicazioni possibili che richiedono la conoscenza di derivate ricordiamo: l'ottimizzazione di forme in aerodinamica attraverso algoritmi di discesa basati sul gradiente, lo studio della propagazione di incertezze attraverso tecniche perturbative, la definizione di funzionali per l'ottimizzazione robusta e l'aumento dell'accuratezza di un funzionale attraverso l'utilizzo dello stato aggiunto.

In questo lavoro verrano sviluppate ed analizzate diverse strategie per la valutazione di derivate prime e seconde di funzionali vincolati, utilizzando tecniche basate sulla Differenziazione Automatica. Inoltre, si descriverà un algoritmo di discesa per l'ottimizzazione di forme in aerodinamica basato su tecniche di gradiente multilivello ed applicabile a diversi tipi di parametrizzazione.

Résumé

La possibilité de calculer les dérivées premières et secondes de fonctionnelles soumises à des contraintes d'égalité données par des équations d'état (et notamment par des systèmes d'équations aux dérivées partielles non linéaires) permet l'emploi de techniques efficaces pour la résolution de plusieurs problèmes de nature industrielle. Parmi les applications possibles qui demandent la connaissance des dérivées, on peut rappeler: l'optimisation de formes en aérodynamique par des algorithmes de descente fondés sur le gradient, l'étude de la propagation des incertitudes par des techniques de perturbation, la définition de fonctionnelles pour l'optimisation robuste et l'augmentation de la précision d'une fonctionnelle par l'utilisation de l'état adjoint.

Dans ce travail, diverses stratégies pour l'évaluation de dérivées premières et secondes de fonctionnelles soumises à des contraintes seront développées et analysées par l'emploi de techniques basées sur la Différentiation Automatique. De plus, on décrira un algorithme de descente pour l'optimisation de formes en aérodynamique fondé sur des techniques de gradient multiniveau et applicable à différents types de paramétrisations.

Introduction

Numerical optimization procedures are based on mathematical techniques for finding the extremum of an objective function subject to various contraints. When the costs associated with evaluating the objective function and constraints are high (as in the case of nonlinear aerodynamic simulations), zero-th order methods (simplex methods, genetic algorithms, ...) are usually prohibitively expensive and, thus, gradient-based design optimization procedure are frequently adopted. These procedure requires the gradient of the objective function and constraints (dependent variables) with respect to the design variables (independent variables). These gradients, commonly referred to as sensitivity derivatives, provide the mechanism for changing the design variables to improve the objective function without any violation of the given constraints.

Today, analysis and design methods in the aeronautical industry, particularly the aerodynamic simulation tools based on Computational Fluid Dynamics (CFD), are based on simulations with a unique set of input data and model variables. However, realistic operating conditions are a superposition of numerous uncertainties under which the industrial products operate. The presence of these uncertainties is the major source of risk in the design decision process, and consequently, increases the level of risk of failure of a given component. Therefore, the actual trend in the CFD community is to introduce these uncertainties within the simulation process by applying non-deterministic methodologies in order to obtain, instead of a single predicted value, an associated domain of variation of the predicted output quantities. One of these methodologies (Method of Moments) uses a perturbative approach in order to quantify the output uncertainties and therefore requires the knowledge of first- and second-order derivatives of the constrained functional of interest.

The sensitivity derivatives can be obtained by the differentiation of the functional of interest subject to satisfying an equality constraint defined by the flow equations. This task, if performed by hand, can be highly tedious and error-prone since the governing equations are also involved through the flow variables. These difficulties are magnified when high-fidelity models are used as it typically happens for aerodynamic optimal design of practical interest. To avoid this exact differentiation, the sensitivity derivatives can be approximated by finite differencing in which the flow solver is used only as a black box. Thus, the resulting approach is very easy to implement and can be applied in a rather general context. Nevertheless, this approach requires a careful parameter monitoring in order to obtain an accurate gradient approximation, and on the other hand, gives dramatic low computational performances as soon as not a low number of control variables is used. Alternatively, in the direct differentiation or flow sensitivity approach, the

Gâteaux derivatives with respect to each component direction are exactly computed. But, the computational cost problem for a large number of control variables is still present, since, for one gradient computation, n (n being the number of design parameters) linearised systems of large dimension should be solved. Finally, an efficient computation of the exact discrete gradient can be achieved through an adjoint formulation (see, e.g. [Giles and Pierce, 2000]). Indeed, the gradient evaluation requires to solve only one extra linear system (the adjoint system), and thus, is independently of the number of design variables. Consequently, at present, adjoint-based methods seem to be the more suitable way to solve complex aerodynamic shape optimization problem. An alternative or additional approach, often used in presence of complex physical models and numerical discretisation (Navier-Stokes equations with a turbulent model, high-order schemes, non-structured meshes, etc.), is to freeze or approximate some steps in the differentiation of the flow solver (see e.g. [Nemec and Zingg, 2002]). We refer to [Dwight and Brézillon, 2006; Carpentieri et al., 2007] for a study on the effect on the gradient accuracy of the various approximations of the discrete adjoint computation. A more drastic approach has been introduced in [Mohammadi, 1997], in which the adjoint computation is dropped out by, on the one hand, adding some intermediate geometrical quantities in the differentiation, and on the other hand, by neglecting the flow derivatives. This incomplete gradient formulation is based on the fact that when the objective functional is defined as a boundary integral of quantities evaluated on the shape, sensibilities with respect to the geometrical quantities give the main contribution to the gradient value. In [de' Michieli Vitturi and Beux, 2006] this approach has been coupled with a multi-level method allowing to consider a completed gradient computation in which the flow derivatives are also (at least partially) taken into account. Moreover, efficient approaches based on finite-difference sensitivities have been also proposed; indeed, in [Catalano et al., 2005] a progressive optimization coupled with a multigrid-aided finite-difference is considered while an one-shot method coupled with a multi-level strategy is used in [Held et al., 2002]. Finally, for more details and references on sensitivity analysis for aerodynamic shape optimization, we refer, for instance, to [Newman III et al., 1999] and [Mohammadi and Pironneau, 2001].

Nevertheless, to avoid the very hard task of a differentiation implementation by hand-coding, *Automatic Differentiation* (AD) tools have been also developed (see e.g. [Giering et al., 2005; Hascoët et al., 2005]) and this seems to be the most promising approach to compute *exact* sensitivities.

In order to improve the optimization efficiency, gradient-based methods can be used with multilevel approach as described in [Beux and Dervieux, 1994]. In this preconditioned gradient method, the minimisation is done alternatively on different subsets of control parameters according to multigrid-like cycles. More particularly, using shape grid-point coordinates as design variables, a hierarchical parametrization was defined considering different subsets of parameters extracted from the complete parameterisation, which can be prolongated to the higher level by linear mapping. This approach acts as a smoother and, on another hand, makes the convergence rate of the gradient-based method low dependent of the number of control parameters. Another type of multilevel approach based on a family of embedded parametrizations has been also proposed in [Désidéri, 2003]. However, this formulation is based on a polynomial representation of the shape through the use of Bézier curves, and is not specifically focused on gradient-based methods.

In the present study we propose a new multilevel strategy which generalizes the approach proposed in [Beux and Dervieux, 1994] to other kinds of parametrizations. This extension, still defined in the context of gradient-based methods, requires the elaboration of an adequate family of sub-parametrizations, possibly embedded, associated to affine prolongation operators. Between the different parametrizations defined, one is grounded on some basic concepts already proposed in [Désidéri, 2003], as, for instance, the degree elevation property of the Bézier curves. Consequently, the associated approach can be also interpreted as a multilevel strategy in which the control parameters are Bézier control points instead of shape grid-points building up, in this way, an explicit link between two kinds of multi-levelling.

Moreover, the developments made during the last years of the three aspects mentioned above, i.e. computational power, adjoint formulation and Automatic Differentiation tools, raised a lot of interest and efforts by research community (like the European project NODESIM-CFD[2]) to build a new (and more robust) approach to optimization that takes into account uncertainty of the input parameters (e.g. uncertainties on geometry or on the model parameters): we are moving from *deterministic design* toward *robust design*. Some of these new methodologies (i.e. perturbation and adjoint-based methods, [Putko et al., 2001; Ghate and Giles, 2006]) are based on Taylor expansion of the functional of interest around the mean (deterministic) value of the uncertain variables and require the knowledge of second-order derivatives of the constrained functional: in fact linear perturbation analysis predicts zero change in the expected value of the mean if the random input perturbations are normalized to have zero mean. Although the theory about uncertainty propagation is well-established (see for example [Walters and Huyse, 2002]), the problem regarding the evaluation of second-order derivatives (Hessian and related quantities) of a functional subject to an equality constraint (typically a set of nonlinear PDE's) is still open.

There already exists several tools to evaluate high-order derivatives, all based on a *Tangent Linearization*. In other words, these are higher-order derivatives in a particular direction corresponding to a particular combination of changes in the input variables. On the other hand, the Reverse mode of AD [Griewank, 2000] is a very efficient approach to computing gradients of a functional with respect to all of the input variables, but the tools that perform *Reverse Differentiation* (e.g. TAPENADE[3], OpenAD[4], TAF[5], ADOL-C[6]) do so only for first-order derivatives.

Thus, we are interested in the study and development of methodologies and algorithms to compute second-order derivatives of constrained functionals, using Automatic Differentiation and adjoint formulation, and then, we plan to define the best strategies for building these derivatives accordingly with the cardinality of independent variables and system size.

Adjoint methods have also been used for error analysis and correction [Pierce and Giles, 2000, 2004] and grid adaptation [Venditti and Darmofal, 2002]. The key idea of Pierce and Giles is to

[2]`http://www.nodesim.eu`

[3]`http://www-sop.inria.fr/tropics/`

[4]`http://www-unix.mcs.anl.gov/~utke/OpenAD/`

[5]`http://www.fastopt.de/`

[6]`http://www.math.tu-dresden.de/~adol-c/`

use a quantity that depends on the numerical approximation of the adjoint solution to correct the original (approximated) functional. Therefore, if we want to build a gradient-based algorithm to minimize this corrected functional, we need to evaluate terms that involve derivatives of the adjoint state and, due to the dependancy of the adjoint state on first-order derivatives, the gradient of the corrected functional will involve second-order derivatives. Therefore, we want develop an algorithm to evaluate the derivative of the *adjoint-correction* term using AD tools.

On order to afford the previous arguments, this work has the following structure:

- in Chapter 1 we describe the mathematical flow model and the numerical finite-volume schemes implemented by our CFD codes (2D and 3D) in order to solve the steady Euler equations. Note that, even if among the different high-fidelity models it not appears as the more complete one, the system of Euler equations is already largely enough complex to illustrate our purpose and moreover, it still exists problems of practical interest in which turbulent and viscous phenomena can be neglected (as for instance, for studies on supersonic jets);

- in Chapter 2 we describe some techniques developed for uncertainty analysis and propagation, and we give some examples of techniques used for robust design in which first- and second-order derviatives are needed;

- in Chapter 3 we start giving an overview of Automatic Differentiation and then we develop first- and second-order differentiation of a functional subject to an equality constraint that can be solved using fixed-point methods (like the steady Euler equations). Then we recall an existing algorithm for the Hessian evaluation [Sherman et al., 1996] (Tangent-on-Tangent, or forward-on-forward in [Ghate and Giles, 2007]) and we develop a new approach based on Tangent-on-Reverse differentiation, then the performance of both method (that are theoretically equivalents) will be compared. Moreover, we discuss some new AD issues regarding the stack management for Tangent-on-Reverse mode and finally we describe how to build a framework that permits the final users to organize the algorithms in a library and reuse them for a broad range of different problems, resulting in a big saving of development time;

- in Chapter 4 we present a new gradient-like approach for aerodynamic shape optimization based on multilevel concepts for different kinds of parametrization (Bézier curves and shape function basis) and we give some numerical experiments;

- in Chapter 5 we develop a new algorithm to compute the gradient of an adjoint-corrected functional [Pierce and Giles, 2004, 2000]. The algorithm is based on Automatic Differentiation techniques and introduce a new differentiation mode for second-order derivatives (Reverse-on-Tangent);

- in Chapter 6 we present some numerical experiments regarding the application of the algorithms developed in order to perform the gradient and Hessian evaluation of the drag coefficient with respect to Mach number and angle of attack. For our tests we consider two different 3D geometries and different orders of spatial accuracy of the solution. Moreover, we present some numerical experiments regarding the behaviour of the iterative linear solvers built using Automatic Differentiation and some strategies for preconditioning.

Publications

- M. Martinelli, R. Duvigneau, *"Comparison of second-order derivatives and metamodel-based Monte-Carlo approaches to estimate statistics for robust design of a transonic wing"*, in preparation.
- M. Martinelli, F. Beux, "Multi-level gradient-based methods and parametrisation in aerodynamic shape design", *Revue Européenne de Mécanique Numérique - European Journal of Computational Mechanics*, vol.17/1-2, pp. 173-201, Hermès Science - Lavoisier Pub., 2008.
- M. Martinelli, F. Beux, "Multilevel gradient-based methods in aerodynamic shape design" *ESAIM (European Series in Applied and Industrial Mathematics): Proceedings*, EDP Sciences. In press.

Proocedings of conference

- M. Martinelli, A. Dervieux, L. Hascöet, *"Strategies for computing second-order derivatives in CFD design problems"*, WEHSFF2007 conference, Moscow.
- M. Martinelli, F. Beux, "Multilevel gradient method with Bézier parametrisation for aerodynamic shape optimisation", *Applied and Industrial mathematics in Italy II*, Series on Advances in Mathematics for Applied Sciences - Vol. 75, pp. 432-443, V. Cutello et al. Eds, World Scientific Publishing, 2007.
- M. Martinelli, F. Beux, "Optimum shape design through multilevel gradient-based method using Bézier parametrisation", *The Fourth International Conference on Computational Fluid Dynamics*, 10-14 July 2006, Ghent, Belgium. in *Computational Fluid Dynamics 2006*, Springer, Engineering series, Deconinck and Dick (Eds).

European project

- Partecipation to the project NODESIM-CFD "Non-Deterministic Simulation for CFD-based Design Methodologies" funded by the European Community represented by the CEC, Research Directorate-General, in the 6th Framework Programme, under Contract No. AST5-CT-2006-030959.

Talks

- 7^{th} European AD Workshop, University of Hertfordshire, Hatfield (UK), 21-22 May 2007: talk on *"Hessian computation of constrained functionals using Automatic Differentiation"*.

- 1st NODESIM meeting, Dassault Aviation, Paris (France), 10-11 May 2007: talk on *"Perturbation techniques in non-deterministic simulations with Adjoint methods and Automatic Differentiation"*.
- 8th Congress of SIMAI, Baia Samuele (Italy), 22-26 May 2006: talk on *"Multilevel gradient method with Bézier parametrisation for aerodynamic shape optimisation"*. during a minisymposium in Optimization.

Chapter 1

Flow modelling and CFD solver

The first step in order to numerically solve an aerodynamic optimal design problem is the definition of the governing equations and its corresponding discretization. Thus, in this chapter we present the mathematical basis for the model used in our multilevel optimization procedure (Chapter 4) and for the validation of the algorithms developed to evaluate second-order derivatives (Chapters 3 and 6).

The model describes the steady flow of a compressible inviscid fluid (Euler equations) on 2D or 3D domains. The numerical approximation is a vertex-centered finite-volume method on a dual mesh constructed from a finite element discretization of the computational domain by triangle (2D) or tetrahedra (3D). The numerical fluxes are based on Flux Vector Splitting [van Leer, 1982] for the 2D case and on Roe's Flux Difference Splitting [Roe, 1981] for the 3D case. In order to improve the spatial accuracy we can adopt a MUSCL (Monotonic Upstream Scheme for system of Conservation Laws) scheme [van Leer, 1979; Fézoui and Dervieux, 1989], while slope limiters can be used to keep the scheme non-oscillatory [Cournède et al., 2006]. The pseudo-time iterations are performed applying a linearized implicit algorithm with first-order exact Jacobian [Fézoui and Stoufflet, 1989].

1.1 Governing equations

Let D be an open subset of $\mathbb{R}^p$, and let $\mathbf{F}_i$, $1 \leq i \leq d$, be d smooth functions from D into $\mathbb{R}^p$; the general form of a system of conservation laws in several space variables is

$$\frac{\partial \mathbf{W}(\mathbf{x},t)}{\partial t} + \sum_{i=0}^{d} \frac{\partial}{\partial x_i} \mathbf{F}_i(\mathbf{W}) = \mathbf{0}, \quad \mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d, \quad t > 0 \tag{1.1}$$

where $\mathbf{W}(\mathbf{x},t) = (w_1, \ldots, w_p)^T$ is a vector valued function from $\mathbb{R}^d \times [0, +\infty[$ into D. The set D is called the set of states and the functions $\mathbf{F}_i = (F_{1i}, \ldots, F_{pi})^T$ are called the flux-functions. The system (1.1) is said to be written in *conservative form*[1].

[1]In the sequel, to keep the notation as light as possible, we omit to write the space and time dependancy for the various quantities take in account. So, the state variable we write $\mathbf{W}$ instead of $\mathbf{W}(\mathbf{x}, t)$. When we need to refer to a specific time-step $t = n\Delta t$ we write $\mathbf{W}^n$ instead of $\mathbf{W}(\mathbf{x}, n\Delta t)$.

Formally, the system (1.1) expresses the conservation of the p quantities $W_1, \dots, W_p$. In fact, let Ω an arbitrary domain of $\mathbb{R}^d$, and let $\hat{\mathbf{n}} = (\hat{n}_1, \dots, \hat{n}_d)^T$ be the outward unit normal to the boundary $\partial\Omega$ of Ω. Then, it follows from (1.1) that

$$\frac{\partial}{\partial t} \int_{\Omega} \mathbf{W} \, d\Omega + \int_{\partial\Omega} \sum_{i=0}^{d} \mathbf{F}_i \hat{n}_i \, d\sigma = \mathbf{0}. \tag{1.2}$$

Sometimes the system (1.1) is said to be in *divergence form*, in fact if we introduce the new vector $\mathcal{F} = (\mathbf{F}_1, \dots, \mathbf{F}_d)^T$ then its divergence is

$$\operatorname{div} \mathcal{F} = \nabla \cdot \mathcal{F} = \sum_{i=0}^{d} \frac{\partial}{\partial x_i} \mathbf{F}_i$$

and the term $\sum_{i=0}^{d} \mathbf{F}_i \hat{n}_i$ can be expressed as the dot product $\mathcal{F} \cdot \hat{\mathbf{n}}$. Then, for the Gauss-Green theorem the system (1.2) is equivalent to

$$\frac{\partial}{\partial t} \int_{\Omega} \mathbf{W} \, d\Omega + \int_{\Omega} \operatorname{div} \mathcal{F} \, d\Omega = \mathbf{0}. \tag{1.3}$$

It is important to note that the systems (1.1) and (1.2) are equivalent in the sense of that a *regular* solution that satisfies (1.1) then satisfies (1.2) and vice-versa, but the integral form permits to consider more general solutions (weak solutions), and in particular, it takes into account the possibility of *discontinuous solutions*.

2D Euler equations.

Using the previous notation, the 2D Euler equations in integral form is given by the (1.2) with $d = 2$, $p = 4$ and

$$\mathbf{W} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} \qquad \mathbf{F}_1(\mathbf{W}) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ u(E+p) \end{pmatrix} \qquad \mathbf{F}_2(\mathbf{W}) = \begin{pmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ v(E+p) \end{pmatrix}$$

where ρ is the density, $\mathbf{v} = (u, v)^T$ is the velocity vector, E is the total energy per unit volume and p is the pressure. If we assume that the fluid satisfies the perfect gas law, we have

$$p = (\gamma - 1)\big(E - \frac{1}{2}\rho ||\mathbf{v}||^2\big) \tag{1.4}$$

where γ is the ratio of specific heat and is equal to 1.4 for the air.

3D Euler equations.

Using the previous notation, the 3D Euler equations in integral form is given by the (1.2) with $d = 3$, $p = 5$. For this case the state vector is defined to be as

$$\mathbf{W} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{pmatrix} \tag{1.5}$$

and the flux-functions are given by

$$\mathbf{F}_1(\mathbf{W}) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ u(E+p) \end{pmatrix} \qquad \mathbf{F}_2(\mathbf{W}) = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho uw \\ v(E+p) \end{pmatrix} \qquad \mathbf{F}_3(\mathbf{W}) = \begin{pmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho w^2 + p \\ w(E+p) \end{pmatrix} \tag{1.6}$$

As for the 2D case, the pression p is given by the (1.4), but the velocity vector is now $\mathbf{v} = (u, v, w)^T$.

The vector $\mathbf{W}$ defined in (1.5) is commonly referred to as the "conservative" state vector and the variables ρ, $\rho\mathbf{v}$, E are referred as *conservative variables*.

It can be shown [Toro, 1999; Godlewski and Raviart, 1996] that the 2D and the 3D Euler equations are invariant under rotation: this fact permits us to choose a rotated reference frame $\bar{\mathbf{x}}$ (respect to the original reference $\mathbf{x}$) in which the multidimensional Euler equations can be transformed in the $\bar{x}$-split version

$$\frac{\partial \bar{\mathbf{W}}}{\partial t} + \frac{\partial}{\partial \bar{x}} \mathbf{F}_1(\bar{\mathbf{W}}) = \mathbf{0}. \tag{1.7}$$

In other words, if $\mathbf{W} = (\rho, \rho\mathbf{v}, E)^T$, we can choose an invertible operator $\mathbf{R}$ such that

$$\mathcal{F}(\mathbf{W}) \cdot \hat{\mathbf{n}} = \mathbf{R}\mathbf{F}_1(\bar{\mathbf{W}}) \tag{1.8}$$

with $\bar{\mathbf{W}} = \mathbf{R}^{-1}\mathbf{W} = (\rho, \rho\bar{\mathbf{v}}, E)^T$. The rotation that transforms the original Euler equation (1.1) into the x-split version (1.7), is the one that align the $\bar{x}$-axis of the new reference frame along the direction of $\hat{\mathbf{n}}$.

Moreover, the x-split Euler equations (1.7) with the ideal-gas equation of state (1.4) satisfy the homogeneity property (e.g. [Toro, 1999])

$$\mathbf{F}_1(\bar{\mathbf{W}}) = \mathbf{A}(\bar{\mathbf{W}})\bar{\mathbf{W}} \tag{1.9}$$

where $\mathbf{A}(\bar{\mathbf{W}})$ is the Jacobian matrix of the first component $\mathbf{F}_1$ of the flux functions $\mathcal{F}$, namely $\mathbf{A}(\bar{\mathbf{W}}) = \partial\mathbf{F}_1/\partial\bar{\mathbf{W}}$. Then, the differentiation of (1.8) writes as

$$\mathbf{A}(\mathbf{W}, \hat{\mathbf{n}}) = \frac{\partial \mathcal{F}(\mathbf{W}) \cdot \hat{\mathbf{n}}}{\partial \mathbf{W}} = \mathbf{R}\frac{\partial \mathbf{F}_1(\bar{\mathbf{W}})}{\partial \bar{\mathbf{W}}}\mathbf{R}^{-1} = \mathbf{R}\mathbf{A}(\bar{\mathbf{W}})\mathbf{R}^{-1} \tag{1.10}$$

so that for any $\mathbf{V} = \mathbf{R}\bar{\mathbf{V}}$

$$\mathbf{A}(\mathbf{W}, \hat{\mathbf{n}})\mathbf{V} = \mathbf{R}\mathbf{A}(\bar{\mathbf{W}})\bar{\mathbf{V}}$$

These properties of the Euler equations form the basis for the numerical schemes that we will use in the following, i.e. *Flux Splitting* type methods.

1.1.1 Boundary conditions

In the sequel, to numerically solve the Euler equation, we consider bounded domains of computation Ω related to external flow around bodies, so we have two kind of boundary (see the Fig. 1.1):

- a physical boundary Γ_B, i.e. the wall boundary of the body;
- the farfield boundary Γ_∞ (this is an artificial boundary introducted to bound the diameter of the computational domain Ω).

On the wall $\partial\Omega_B$ we assume the following slip condition

$$\mathbf{v} \cdot \hat{\mathbf{n}} = 0$$

in which $\mathbf{v}$ is the fluid velocity and $\hat{\mathbf{n}}$ is the unit vector normal to the wall. This condition states that the particle fluid of velocity $\mathbf{v}$ cannot pass through the wall boundary. Remembering the flux-function for the Euler equation, this condition gives us the system

$$\frac{\partial}{\partial t}\int_\Omega \mathbf{W}\, d\Omega + \int\limits_{\partial\Omega\backslash\Gamma_B} \mathcal{F} \cdot \hat{\mathbf{n}}\, d\sigma + \int\limits_{\Gamma_B} p \begin{pmatrix} 0 \\ \hat{\mathbf{n}} \\ 0 \end{pmatrix} d\sigma = \mathbf{0}. \tag{1.11}$$

For the farfield we assume the flow to be uniform at infinity and we prescribe an unitary density $\rho_\infty = 1$, and the velocity vector given by $\mathbf{v}_\infty = (0, M_\infty \cos\alpha, M_\infty \sin\alpha, 0)^T$ for the 2D case and $\mathbf{v}_\infty = (0, M_\infty \cos\alpha, M_\infty \sin\alpha, 0, 0)^T$ for the 3D case, where α is the angle of attack and M_∞ denote the free-stream Mach number. The pression at infinity is computed with the formula $p_\infty = \frac{1}{\gamma M_\infty^2}$.

1.2 Numerical approach

1.2.1 Finite Volume method

In the *Finite Volume (FV)* method, the computational domain Ω is composed of cells, or control volumes, Ω_i with the following properties

- $\Omega = \bigcup_i \Omega_i$;
- $\overset{\circ}{\Omega}_i \cap \overset{\circ}{\Omega}_j = \varnothing$ for $i \neq j$ (non overlapping);
- $\partial\Omega_i = \bigcup_i \Gamma_{ij}$ where $\Gamma_{ij} = \Omega_i \cap \Omega_j$ is the common boundary separating Ω_i and Ω_j.

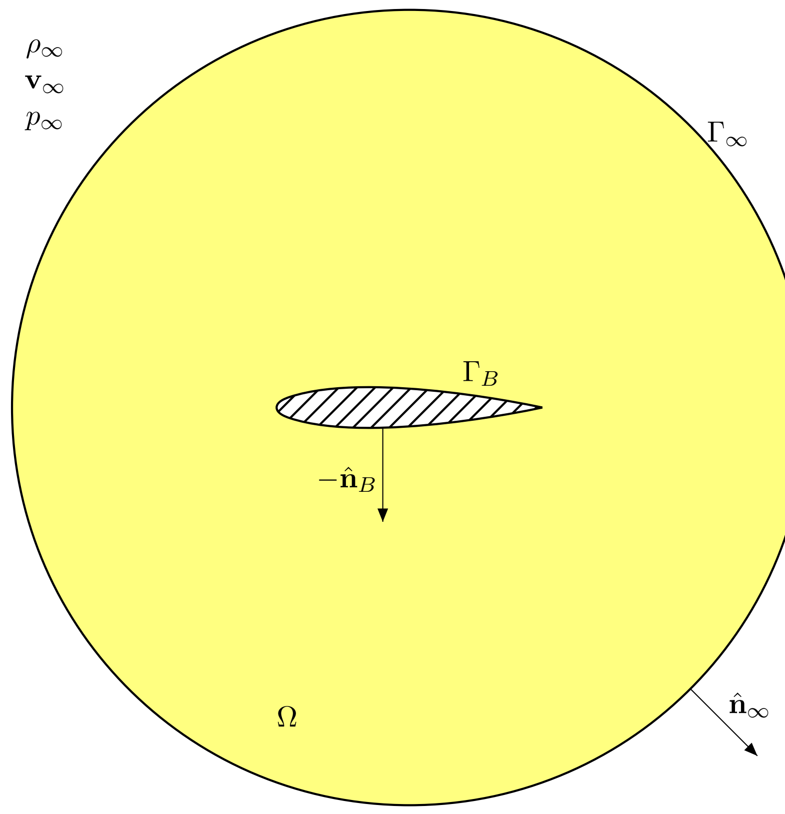


Figure 1.1: Γ_B it the wall boundary and Γ_∞ is the far-field boundary. The positive direction of the boundary normals $\hat{\mathbf{n}}_\infty$, $\hat{\mathbf{n}}_\infty$ is toward the external of the computational domain Ω.

On Ω_i, $\mathbf{W}(\cdot,t)$ is approximated by a constant $\tilde{\mathbf{W}}_i(t)$, which should be considered as an approximation of the mean value of $\mathbf{W}$ over the cell Ω_i (and not the value of $\mathbf{W}$ at the center of the cell Ω_i)

$$\tilde{\mathbf{W}}_i(t) \simeq \frac{1}{|\Omega_i|}\int_{\Omega_i} \mathbf{W}(\mathbf{x},t)\,d\Omega.$$

where $|\Omega_i|$ is the measure of the volumes Ω_i. With these assumption, it's clear that the relation (1.11) should be verified on each control volumes Ω_i and then the first term in the equation (1.11) is approximated by

$$\frac{\partial}{\partial t}\int_{\Omega_i} \mathbf{W}\,d\Omega \simeq |\Omega_i|\frac{\partial \tilde{\mathbf{W}}_i}{\partial t}. \tag{1.12}$$

The second term represents the flux across the boundary of the cell at time t and it can be written

$$\int\limits_{\partial\Omega_i\backslash(\partial\Omega_i\cap\Gamma_B)} \mathcal{F}(\mathbf{W})\cdot\hat{\mathbf{n}}\,d\sigma = \sum_{\Gamma_{ij}\subset\partial\Omega_i\backslash(\partial\Omega_i\cap\Gamma_B)} \int_{\Gamma_{ij}} \mathcal{F}(\mathbf{W})\cdot\hat{\mathbf{n}}\,d\sigma \tag{1.13}$$

where the sum is taken over all the edges of the cell Ω_i that do not belong to the wall boundary Ω_B. We note that the flux across the farfield boundary is taken in account into (1.13), so we can be more explicit writing the last relation as

$$\begin{aligned}\int\limits_{\partial\Omega_i\backslash(\partial\Omega_i\cap\Gamma_B)} \mathcal{F}(\mathbf{W})\cdot\hat{\mathbf{n}}\,d\sigma =& \sum_{\Gamma_{ij}\subset\partial\Omega_i\backslash\big(\partial\Omega_i\cap(\Gamma_B\cup\Gamma_\infty)\big)} \int_{\Gamma_{ij}} \mathcal{F}(\mathbf{W})\cdot\hat{\mathbf{n}}\,d\sigma \\ &+ \sum_{\Gamma_{i\infty}\subset(\partial\Omega_i\cap\Gamma_\infty)} \int_{\Gamma_{i\infty}} \mathcal{F}(\mathbf{W})\cdot\hat{\mathbf{n}}\,d\sigma\end{aligned} \tag{1.14}$$

where the edge $\Gamma_{i\infty}$ belongs to the cell Ω_i and the farfield.

The flux across the wall boundary is taken in account by the third term in (1.11) and then is approximated by

$$\int\limits_{\partial\Omega_i\cap\Gamma_B} p(\mathbf{W})\begin{pmatrix}0\\ \hat{\mathbf{n}}\\ 0\end{pmatrix} d\sigma \simeq \sum_{\Gamma_{iB}\subset(\partial\Omega_i\cap\Gamma_B)} p(\tilde{\mathbf{W}}_i)\int_{\Gamma_{iB}}\begin{pmatrix}0\\ \hat{\mathbf{n}}\\ 0\end{pmatrix} d\sigma \tag{1.15}$$

where the pression $p(\tilde{\mathbf{W}}_i)$ is constant over the cell Ω_i and the edges Γ_{iB} belongs to the wall boundary Γ_B.

The problem is then to define the numerical fluxes approximating $\int_{\Gamma_{ij}} \mathcal{F}(\mathbf{W})\cdot\hat{\mathbf{n}}\,d\sigma$, using only the values $\tilde{\mathbf{W}}_i$. For this purpose, we introduce the numerical flux-function $\Phi\colon (\mathbf{U},\mathbf{V},\boldsymbol{\eta}) \to \Phi(\mathbf{U},\mathbf{V},\boldsymbol{\eta})$ such that

$$\int_{\Gamma_{ij}} \mathcal{F}(\mathbf{W})\cdot\hat{\mathbf{n}}\,d\sigma \simeq |\eta^{ij}|\Phi(\tilde{\mathbf{W}}_i,\tilde{\mathbf{W}}_j,\hat{\boldsymbol{\eta}}^{ij}) \tag{1.16}$$

with the metric coefficient

$$|\eta^{ij}| = ||\boldsymbol{\eta}^{ij}|| \quad , \quad \hat{\boldsymbol{\eta}}^{ij} = \frac{\boldsymbol{\eta}^{ij}}{||\boldsymbol{\eta}^{ij}||} \quad , \quad \boldsymbol{\eta}^{ij} = \int_{\Gamma_{ij}} \hat{\mathbf{n}}\,d\sigma \tag{1.17}$$

and the normal $\hat{\mathbf{n}}$ to the boundary Γ_{ij} is pointing outward to Ω_i in the direction of Ω_j[2]. Furthermore, we assume that for two generic states $\mathbf{U}$, $\mathbf{V}$ and for the generic direction $\hat{\mathbf{n}}$ the following properties hold:

- $\Phi(\mathbf{U},\mathbf{V},\hat{\mathbf{n}})$ is *locally Lipshitz continuous* with respect to $\mathbf{U}$, $\mathbf{V}$;
- $\Phi(\mathbf{U},\mathbf{V},\hat{\mathbf{n}})$ is *conservative*:
$$\Phi(\mathbf{U},\mathbf{V},\hat{\mathbf{n}}) = -\Phi(\mathbf{V},\mathbf{U},-\hat{\mathbf{n}})$$
- $\Phi(\mathbf{U},\mathbf{V},\hat{\mathbf{n}})$ is *consistent*:
$$\Phi(\mathbf{U},\mathbf{U},\hat{\mathbf{n}}) = \mathcal{F}(\mathbf{U})\cdot\hat{\mathbf{n}}$$

Before to define the meaning (and the formulation) for the numerical flux Φ, we can introduce the n_s-dimensional vector Ψ (with n_s the number of cells that are in the computational domain) in which the i-th is related to the cell Ω_i

$$\begin{aligned}\Psi_i(\tilde{\mathbf{W}}) =& \frac{1}{|\Omega_i|}\Bigg(\sum_{\Gamma_{ij}\subset\partial\Omega_i\setminus\big(\partial\Omega_i\cap(\Gamma_B\cup\Gamma_\infty)\big)} |\eta^{ij}|\Phi(\tilde{\mathbf{W}}_i,\tilde{\mathbf{W}}_j,\hat{\boldsymbol{\eta}}^{ij}) + \\ &+ \sum_{\Gamma_{i\infty}\subset(\partial\Omega_i\cap\Gamma_\infty)} |\eta^{i\infty}|\Phi(\tilde{\mathbf{W}}_i,\tilde{\mathbf{W}}_\infty,\hat{\boldsymbol{\eta}}^{i\infty}) + \sum_{\Gamma_{iB}\subset\partial\Omega_i\cap\Gamma_B} |\eta^{iB}| p(\tilde{\mathbf{W}}_i)\begin{pmatrix}0\\ \hat{\boldsymbol{\eta}}^{iB}\\ 0\end{pmatrix}\Bigg)\end{aligned} \tag{1.19}$$

with this function, we can rewrite (1.2) in the more compact way

$$\frac{\partial\tilde{\mathbf{W}}}{\partial t} + \Psi(\tilde{\mathbf{W}}) = \mathbf{0} \tag{1.20}$$

1.2.2 Numerical Fluxes

The scheme adopted here to define the numerical flux, is based on the Flux Vector Splitting Methods [Toro, 1999] which consists in decomposing the flux $\mathbf{F}$ in two part

$$\mathcal{F}(\mathbf{W})\cdot\hat{\mathbf{n}} = \mathbf{F}^+_{\hat{\mathbf{n}}}(\mathbf{W}) + \mathbf{F}^-_{\hat{\mathbf{n}}}(\mathbf{W})$$

As for the x-split case, for the multidimensional Euler equations (for perfect gas) the homogeneity property holds

$$\mathcal{F}(\mathbf{W})\cdot\hat{\mathbf{n}} = \mathbf{A}_{\hat{\mathbf{n}}}(\mathbf{W})\mathbf{W}$$

where $\mathbf{A}_{\hat{\mathbf{n}}}(\mathbf{W}) = \mathbf{A}(\mathbf{W},\hat{\mathbf{n}})$ is the Jacobian matrix $\partial(\mathcal{F}\cdot\hat{\mathbf{n}})/\partial\mathbf{W}$.

[2]With the introduction of the vector $\hat{\boldsymbol{\eta}}^{iB}$, i.e. the normal to the edge between the cell Ω_i and wall boundary, we can rewrite the total flux across the wall boundary (1.15) as

$$\int_{\partial\Omega_i\cap\Gamma_B} p(\mathbf{W})\begin{pmatrix}0\\ \hat{\mathbf{n}}\\ 0\end{pmatrix} d\sigma \simeq \sum_{\Gamma_{iB}\subset(\partial\Omega_i\cap\Gamma_B)} |\eta^{iB}| p(\tilde{\mathbf{W}}_i)\begin{pmatrix}0\\ \hat{\boldsymbol{\eta}}^{iB}\\ 0\end{pmatrix} \tag{1.18}$$

Then, following [Steger and Warming, 1981] we can define

$$\mathbf{F}^{\pm}_{\hat{\mathbf{n}}}(\mathbf{W}) = \mathbf{A}^{\pm}_{\hat{\mathbf{n}}}(\mathbf{W})\mathbf{W}$$

and

$$\Phi(\mathbf{W}_L, \mathbf{W}_R, \hat{\mathbf{n}}) = \mathbf{F}^{+}_{\hat{\mathbf{n}}}(\mathbf{W}_L) + \mathbf{F}^{-}_{\hat{\mathbf{n}}}(\mathbf{W}_R)$$

Due to the rotation-invariant property of the Euler equations and using the (1.10) we can choose a reference frame in which holds $\mathbf{A}^{\pm}_{\hat{\mathbf{n}}} = \mathbf{R}\mathbf{A}^{\pm}\mathbf{R}^{-1}$ where $\mathbf{A}^{\pm} = \partial\mathbf{F}_1/\partial\mathbf{W}$, then we can write (remembering that $\bar{\mathbf{W}} = \mathbf{R}^{-1}\mathbf{W}$)

$$\Phi(\mathbf{W}_L, \mathbf{W}_R, \hat{\mathbf{n}}) = \mathbf{R}\big(\mathbf{F}^{+}_1(\bar{\mathbf{W}}_L) + \mathbf{F}^{-}_1(\bar{\mathbf{W}}_R)\big) = \mathbf{R}\Phi(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R, \hat{\mathbf{e}}_1) \tag{1.21}$$

where $\hat{\mathbf{e}}_1$ is the versor parallel to the $\bar{x}$-axis and the problem is therefore to define the numerical fluxes $\mathbf{F}^{\pm}_1(\bar{\mathbf{W}})$.

To specify the numerical fluxes, one ask that some condition must be verified. A classical request is that the eigenvalues λ^{+}_i and λ^{-}_i of the Jacobian matrices

$$\mathbf{A}^{+} = \frac{\partial\mathbf{F}^{+}_1}{\partial\bar{\mathbf{W}}}, \quad \mathbf{A}^{-} = \frac{\partial\mathbf{F}^{-}_1}{\partial\bar{\mathbf{W}}}$$

satisfies the condition $\lambda^{+}_i \geq 0$ and $\lambda^{-}_i \leq 0$. Moreover, the decomposition is also required to reproduce regular upwinding when all the eigenvalues of the Jacobian matrix $\partial\mathbf{F}_1/\partial\bar{\mathbf{W}} = \mathbf{A}(\mathbf{W}) = \mathbf{R}^{-1}\mathbf{A}_{\hat{\mathbf{n}}}(\mathbf{W})\mathbf{R}$ are of the same sign, that is to say $\mathbf{F}^{+}_1 = \mathbf{F}_1$ ($\mathbf{F}^{-}_1 = \mathbf{0}$) if the eigenvalues of $\mathbf{A}(\mathbf{W})$ are ≥ 0, and $\mathbf{F}^{-}_1 = \mathbf{F}_1$ ($\mathbf{F}^{+}_1 = \mathbf{0}$) if the eigenvalues of $\mathbf{A}(\mathbf{W})$ are ≤ 0. In this sense $\mathbf{F}^{+}_1$ represents the flux transported upstream, while $\mathbf{F}^{-}_1$ represents the flux transported downstream. The definion of the $\mathbf{F}^{\pm}_1$ satisfying the previous restriction is not unique, therefore we have many numerical schemes with different properties. Two of them are the van Leer's flux splitting (used in our 2D code) and the Roe's scheme (used in our 3D code).

Van Leer flux vector splitting

Van Leer flux vector splitting [van Leer, 1982] satisfies the extra properties:

- the Jacobian matrices of the numerical flux $\mathbf{A}^{\pm} = \partial\mathbf{F}^{\pm}_1/\partial\bar{\mathbf{W}}$ are continuous;
- if the normal speed[3] $\mathbf{v}\cdot\hat{\mathbf{n}}$ is supersonic then

$$\begin{cases} \mathbf{v}\cdot\hat{\mathbf{n}} \geq a & \Longrightarrow \quad \mathbf{F}^{+}_1(\bar{\mathbf{W}}) = \mathbf{F}_1(\bar{\mathbf{W}}), \quad \mathbf{F}^{-}_1(\bar{\mathbf{W}}) = \mathbf{0} \\ \mathbf{v}\cdot\hat{\mathbf{n}} \leq -a & \Longrightarrow \quad \mathbf{F}^{+}_1(\bar{\mathbf{W}}) = \mathbf{0}, \quad \mathbf{F}^{-}_1(\bar{\mathbf{W}}) = \mathbf{F}_1(\bar{\mathbf{W}}) \end{cases} \tag{1.22}$$

where a is the sound speed (for a perfect gas $a = \sqrt{\gamma p/\rho}$).

[3] In the rotated frame of reference, the normal speed is the first component (i.e. along the $\bar{x}$-axis, let say $\bar{u}$) of the velocity $\bar{\mathbf{v}}$, where $\bar{\mathbf{W}} = (\rho, \rho\bar{\mathbf{v}}^T, E)^T = \mathbf{R}^{-1}(\rho, \rho\mathbf{v}^T, E)^T$.

Then, (for the 3D case) the flux splitting is defined as

$$\mathbf{F}_1^{\pm}(\bar{\mathbf{W}}) = \pm\frac{\rho(\bar{u} \pm a)^2}{4a}\begin{pmatrix} 1 \\ \dfrac{(\gamma-1)\bar{u} \pm 2a}{\gamma} \\ \bar{v} \\ \bar{w} \\ \dfrac{[(\gamma-1)\bar{u} \pm 2a]^2}{2(\gamma^2-1)}. \end{pmatrix}, \qquad |\bar{u}| < a. \tag{1.23}$$

For the 2D case, the flux splitting is of the same form of (1.23), but without the fourth vector element.

Roe's flux difference splitting

Roe's approach [Toro, 1999] represents a successful attempt to extend the exact linear wave decomposition to non-linear hyperbolic equations, and consists in a quasi-linearization of the flux $\mathbf{F}_1(\bar{\mathbf{W}}) = \mathbf{A}(\bar{\mathbf{W}})\bar{\mathbf{W}}$, replacing the Jacobian matrix $\mathbf{A}(\bar{\mathbf{W}})$ by a *constant* Jacobian matrix

$$\tilde{\mathbf{A}} = \tilde{\mathbf{A}}(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R)$$

which is a function of the data states $\bar{\mathbf{W}}_L$, $\bar{\mathbf{W}}_R$. At the interface between two adiacent cells, the matrix $\tilde{\mathbf{A}}(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R)$ is required to satisfy the following properties:

- $\tilde{\mathbf{A}}(\bar{\mathbf{W}}, \bar{\mathbf{W}}) = \mathbf{A}(\bar{\mathbf{W}}) = \dfrac{\partial \mathbf{F}_1}{\partial \bar{\mathbf{W}}}$ (consistency with the exact Jacobian);
- $\tilde{\mathbf{A}}$ has a set of real eigenvalues $\tilde{\lambda}_i = \tilde{\lambda}_i(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R)$ and a complete set of linearly independent right eigenvectors
$$\tilde{\mathbf{A}} = \tilde{\mathbf{K}}\tilde{\mathbf{\Lambda}}\tilde{\mathbf{K}}^{-1}$$
where $\tilde{\mathbf{\Lambda}}$ is the diagonal matrix whose non-zero coefficient are the eigenvalues of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{K}}$ is the matrix in which the columns are the eigenvectors of $\tilde{\mathbf{A}}$;
- $\mathbf{F}_1(\bar{\mathbf{W}}_R) - \mathbf{F}_1(\bar{\mathbf{W}}_L) = \tilde{\mathbf{A}}(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R)(\bar{\mathbf{W}}_R - \bar{\mathbf{W}}_L)$ (consistency across discontinuities).

We define the following decomposition of $\tilde{\mathbf{\Lambda}}$

$$\tilde{\mathbf{\Lambda}} = \tilde{\mathbf{\Lambda}}^+ + \tilde{\mathbf{\Lambda}}^- \quad ; \quad \tilde{\mathbf{\Lambda}}^+ = \frac{\tilde{\mathbf{\Lambda}} + |\tilde{\mathbf{\Lambda}}|}{2} \quad ; \quad \tilde{\mathbf{\Lambda}}^- = \frac{\tilde{\mathbf{\Lambda}} - |\tilde{\mathbf{\Lambda}}|}{2}$$

where $|\tilde{\mathbf{\Lambda}}|$ is the diagonal matrix in which the nonzero coefficients are $|\tilde{\lambda}_i|$, $i = 1, \ldots, p$. We also define the corresponding matrices

$$\tilde{\mathbf{A}}^{\pm} = \tilde{\mathbf{K}}\tilde{\mathbf{\Lambda}}^{\pm}\tilde{\mathbf{K}}^{-1} \quad ; \quad |\tilde{\mathbf{A}}| = \tilde{\mathbf{K}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{K}}^{-1}$$

Then, from the properties of the matrix $\tilde{\mathbf{A}}$ we obtain

$$\mathbf{F}_1^{\pm}(\bar{\mathbf{W}}_R) - \mathbf{F}_1^{\pm}(\bar{\mathbf{W}}_L) = \tilde{\mathbf{A}}^{\pm}(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R)(\bar{\mathbf{W}}_R - \bar{\mathbf{W}}_L)$$

Remembering the definition of the numerical flux splitting (1.21) and with some algebra, the following relations hold

$$\begin{cases} \Phi(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R, \hat{\mathbf{e}}_1) = \mathbf{F}_1(\bar{\mathbf{W}}_L) + \tilde{\mathbf{A}}^{-}(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R)(\bar{\mathbf{W}}_R - \bar{\mathbf{W}}_L) \\ \Phi(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R, \hat{\mathbf{e}}_1) = \mathbf{F}_1(\bar{\mathbf{W}}_R) - \tilde{\mathbf{A}}^{+}(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R)(\bar{\mathbf{W}}_R - \bar{\mathbf{W}}_L) \end{cases} \tag{1.24}$$

Then, averaging the (1.24) we obtain

$$\Phi(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R, \hat{\mathbf{e}}_1) = \frac{\mathbf{F}_1(\bar{\mathbf{W}}_L) + \mathbf{F}_1(\bar{\mathbf{W}}_R)}{2} - \frac{1}{2}|\tilde{\mathbf{A}}(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R)|(\bar{\mathbf{W}}_R - \bar{\mathbf{W}}_L). \tag{1.25}$$

The diffusive term $|\tilde{\mathbf{A}}(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R)|(\bar{\mathbf{W}}_R - \bar{\mathbf{W}}_L)$ in the last equation can be expressed as linear combination of the eigenvectors of $\tilde{\mathbf{A}}$ (see [Toro, 1999])

$$\Phi(\bar{\mathbf{W}}_L, \bar{\mathbf{W}}_R, \hat{\mathbf{e}}_1) = \frac{\mathbf{F}_1(\bar{\mathbf{W}}_L) + \mathbf{F}_1(\bar{\mathbf{W}}_R)}{2} - \frac{1}{2}\sum_{i=1}^{p} \tilde{\alpha}_i |\tilde{\lambda}_i| \tilde{\mathbf{k}}^{(i)} \tag{1.26}$$

where the quantities $\tilde{\lambda}_i$ (eigenvalues of $\tilde{\mathbf{A}}$), $\tilde{\mathbf{k}}^{(i)}$ (eigenvectors of $\tilde{\mathbf{A}}$) and $\tilde{\alpha}_i$ are function of the two state $\bar{\mathbf{W}}_L$, $\bar{\mathbf{W}}_R$ and depend on the specific hyperbolic problem under exam. Two approach was developed in order to find $\tilde{\alpha}_i$, $\tilde{\lambda}_i$ and $\tilde{\mathbf{k}}^{(i)}$, namely the original approach presented by Roe [Roe, 1981] (in which the matrix $\tilde{\mathbf{A}}$ is sought and from that its eigenvalues and eigenvectors are computed), and the Roe-Pike approach [Roe and Pike, 1985] in which the constrution of $\tilde{\mathbf{A}}$ is avoided. The technical details and the algorithm for the two approach for a general hyperbolic problem can be found (apart from the original article of Roe and Roe and Pike), for example in [Toro, 1999] or in [Godlewski and Raviart, 1996].

The case of the 3D Euler equations for perfect gas, gives for the eigenvalues

$$\tilde{\lambda}_1 = \tilde{u} - \tilde{a}, \quad \tilde{\lambda}_2 = \tilde{\lambda}_3 = \tilde{\lambda}_4 = \tilde{u}, \quad \tilde{\lambda}_5 = \tilde{u} + \tilde{a}$$

and the corresponding right eigenvectors are

$$\tilde{\mathbf{K}} = \left(\tilde{\mathbf{k}}^{(1)}, \tilde{\mathbf{k}}^{(2)}, \tilde{\mathbf{k}}^{(3)}, \tilde{\mathbf{k}}^{(4)}, \tilde{\mathbf{k}}^{(5)}\right) = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ \tilde{u} - \tilde{a} & \tilde{u} & 0 & 0 & \tilde{u} + \tilde{a} \\ \tilde{v} & \tilde{v} & 1 & 0 & \tilde{v} \\ \tilde{w} & \tilde{w} & 0 & 1 & \tilde{w} \\ \tilde{H} - \tilde{u}\tilde{a} & \frac{1}{2}||\tilde{\mathbf{v}}||^2 & \tilde{v} & \tilde{w} & \tilde{H} + \tilde{u}\tilde{a} \end{pmatrix}$$

where $||\tilde{\mathbf{v}}||^2 = \tilde{u}^2 + \tilde{v}^2 + \tilde{w}^2$ and $\tilde{a} = \left[(\gamma - 1)\left(\tilde{H} - \frac{1}{2}||\tilde{\mathbf{v}}||^2\right)\right]^{\frac{1}{2}}$. The tilde symbol over the variables u, v, w and H denotes a "Roe average" for that variable. The Roe average (for, let say, the specific entalpy $H = (E + p)/\rho)$ is defined as follow

$$\tilde{H} = \frac{\sqrt{\rho_L} H_L + \sqrt{\rho_R} H_R}{\sqrt{\rho_L} + \sqrt{\rho_R}} \tag{1.27}$$

where H_L, H_R are the specific enthalpies of the left and right states respectively. The wave strenghts $\tilde{\alpha}_i$ are given by

$$\begin{cases} \tilde{\alpha}_1 = \dfrac{1}{2\tilde{a}^2}\left[(p_R - p_L) - \tilde{\rho}\tilde{a}(u_R - u_L)\right] \\ \tilde{\alpha}_2 = (\rho_R - \rho_L) - \dfrac{1}{\tilde{a}^2}(p_R - p_L) \\ \tilde{\alpha}_3 = \tilde{\rho}(v_R - v_L) \\ \tilde{\alpha}_4 = \tilde{\rho}(w_R - w_L) \\ \tilde{\alpha}_5 = \dfrac{1}{2\tilde{a}^2}\left[(p_R - p_L) + \tilde{\rho}\tilde{a}(u_R - u_L)\right] \end{cases}$$

where $\tilde{\rho} = \sqrt{\rho_L \rho_R}$.

Remark 1.1. The two approaches presented here (i.e. Van Leer flux vector splitting and Roe's flux difference splitting) can be used to solve numerically the Euler equations, but they have different mathematical properties and performances:

- Van Leer flux vector splitting is fully differentiable (by construction) while the Roe's scheme is not. This non-differentiability is due to the presence of the absolute value function into (1.26) and therefore the Roes's scheme is only *piecewise* differentiable.
- Despite the lower regularity, Roe's flux difference splitting is more accurate with respect to Van Leer approach, which is more dissipative (see [van Leer et al., 1987]).

Remark 1.2. For the flux across the farfield boundary Γ_∞, we used the flux-splitting of Steger and Warming [Steger and Warming, 1981]

$$\Phi(\tilde{\mathbf{W}}_i, \tilde{\mathbf{W}}_\infty, \hat{\boldsymbol{\eta}}^{i\infty}) = \left(\mathbf{A}^+_{\hat{\boldsymbol{\eta}}^{i\infty}}(\tilde{\mathbf{W}}_i)\tilde{\mathbf{W}}_i + \mathbf{A}^-_{\hat{\boldsymbol{\eta}}^{i\infty}}(\tilde{\mathbf{W}}_\infty)\tilde{\mathbf{W}}_\infty\right) .$$

Remark 1.3. In the following chapters we will need to *differentiate* (at first- and second-order) the (1.19) and from Remark 1.1, if applied to Roe's scheme, this differentiation seems to be not justified. The fact that permits us to perform the differentiation is due to the discretize-then-differentiate approach we used for this work. In this approach we differentiate the discretized equations and therefore the not-differentiability can occur only if, for a given gridpoint, we have for the variables $\mathbf{W}_i$ the exact numerical values for which (1.19) is not differentiable (although this is a set of zero-measure, it can rarely happens that we have numerical values for which our differentiation model is not justified: in these case we could have wrong numerical results). In our experience, we have not encountered any of such problems for first-order differentiation, while second-order differentiation it seems to be a little more sensitive to these issues (see Chapter 6 for some numerical experiments).

1.2.3 Spatial discretization

In this work, we use a vertex-centered finite-volume approximation on a dual mesh constructed from a finite element discretization of the computational domain by triangle (2D) or tetrahedra (3D). In the 2D case, the cells are delimited by the triangle medians (Figure. 1.2), and in 3D the cells are delimited by planes through the middle of an edge (Figure. 1.3).

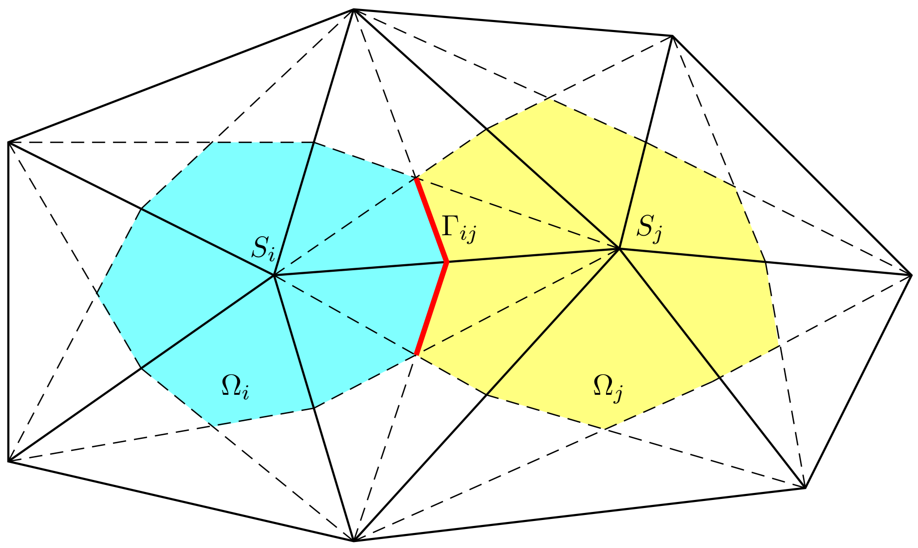


Figure 1.2: Control volumes in the 2D case. Triangular mesh (solid lines) and dual mesh (dashed lins) build with the median segments. The thick red line is the boundary Γ_{ij} common to the cells Ω_i, Ω_j.

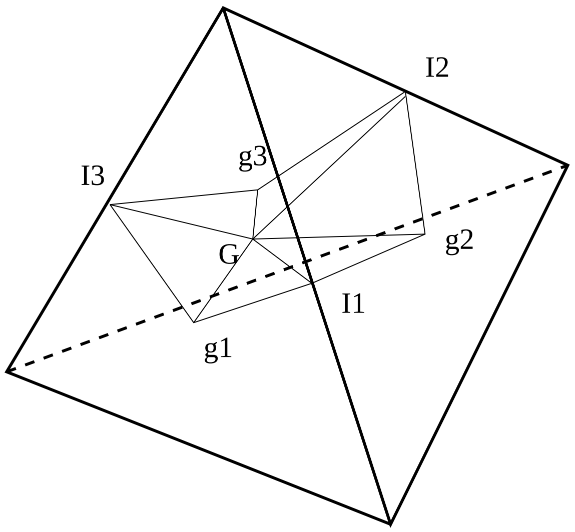


Figure 1.3: Control volume in the 3D case

1.2.4 High-order approximation

With constant-by-cell state variables $\tilde{\mathbf{W}}_i$, the above numerical split-flux integration will result in schemes which are, at best, only first-order spatially-accurate [Godlewski and Raviart, 1996].

To improve the spatial accuracy we can adopt a MUSCL (Monotonic Upstream Scheme for system of Conservation Laws) scheme introducted by [van Leer, 1979]. In this approach, the order of the space-accuracy for the numerical flux function (1.19) is improved by using, not the values $\mathbf{W}_i$, $\mathbf{W}_j$ that are constants in the cells Ω_i, Ω_j, but some interpolated values $\mathbf{W}_{ij}$, $\mathbf{W}_{ji}$ at the interface Γ_{ij}. In order to define $\mathbf{W}_{ij}$ and $\mathbf{W}_{ji}$ we use the upstream and downstream triangles (or tetrahedra) T_{ij} and T_{ji} (see Figure 1.4 for the 2D case and Figure 1.5 for the 3D case), as introducted in [Fézoui and Dervieux, 1989].

Element T_{ij} is *upstream* to vertex S_i with respect to edge $\overline{S_iS_j}$ if for any small enough real number ϵ the vector $-\epsilon\overrightarrow{ij}$ is inside the element T_{ij}. Symmetrically, T_{ji} is *downstream* to vertex S_i with respect to edge $\overline{S_iS_j}$ if for any small enough real number ϵ the vector $\epsilon\overrightarrow{ij}$ is inside the element T_{ij} [Cournède et al., 2006].

Moreover, in order to keep the scheme *non-oscillatory* and positive, we have to introduce *slope limiters.* The main idea behind the construction of slope limiter schemes is to limit the spatial derivatives to realistic values and only come into operation when sharp wave fronts are present. For smoothly changing waves, the slope limiters do not operate and the spatial derivatives can be represented by higher order approximations without introducing unphysical oscillations (see [Cournède et al., 2006] for an application to mixed-element-volume method).

Introducing the notation

$$\Delta^-\mathbf{W}_{ij} = \nabla\mathbf{W}\big|_{T_{ij}} \cdot \overrightarrow{ij}\ , \quad \Delta^0\mathbf{W}_{ij} = \mathbf{W}_j - \mathbf{W}_i\ , \quad \Delta^+\mathbf{W}_{ij} = \nabla\mathbf{W}\big|_{T_{ji}} \cdot \overrightarrow{ij}$$

where the gradient $\nabla\mathbf{W}\big|_T$ is relative to the P1 (continuous and linear) interpolation of $\mathbf{W}$ over the element T, the interpolated values $\mathbf{W}_{ij}$, $\mathbf{W}_{ji}$ write as

$$\begin{cases} \mathbf{W}_{ij} = \mathbf{W}_i + \dfrac{1}{2}\mathbf{L}(\Delta^-\mathbf{W}_{ij}, \Delta^0\mathbf{W}_{ij}) \\ \mathbf{W}_{ji} = \mathbf{W}_j - \dfrac{1}{2}\mathbf{L}(\Delta^0\mathbf{W}_{ij}, \Delta^+\mathbf{W}_{ji}) \end{cases}$$

where $\mathbf{L}$ is the limiter function. Several definitions are possible for limiters (see for example [Cournède et al., 2006]) and each one has different properties (e.g. differentiability) and are selected according to the particular problem and solution scheme. Our 3D Euler solver implements the so-called Van Albada-Van Leer limiter

$$L(x,y) = \begin{cases} \dfrac{x(y^2+\varepsilon) + y(x^2+\varepsilon)}{x^2+y^2+\varepsilon} & \text{if } xy > 0 \\ 0 & \text{otherwise} \end{cases}$$

where ε is a small parameter to prevent division by zero. Note that this limiter is differentiable.

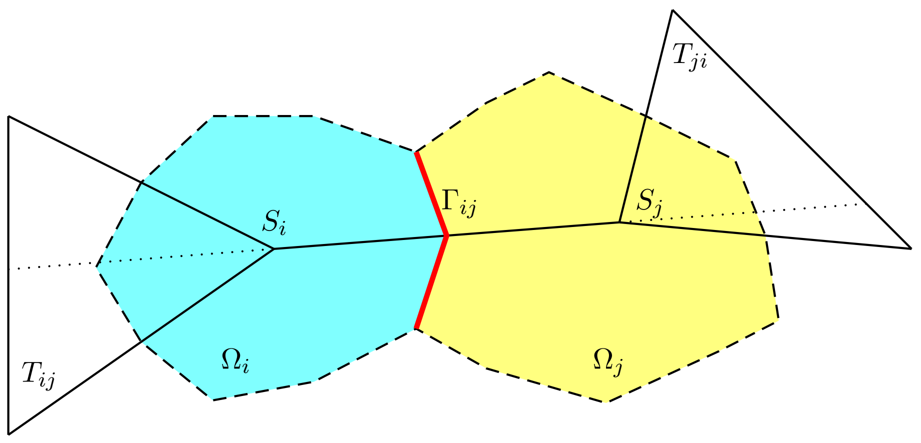


Figure 1.4: Downstream and upstream triangle respect to the boundary Γ_{ij} are triangles having S_i and S_j as vertex and such that line $\overline{S_iS_j}$ intersects the opposite edge (see the Fig. 1.2 for a comparison with the original mesh).

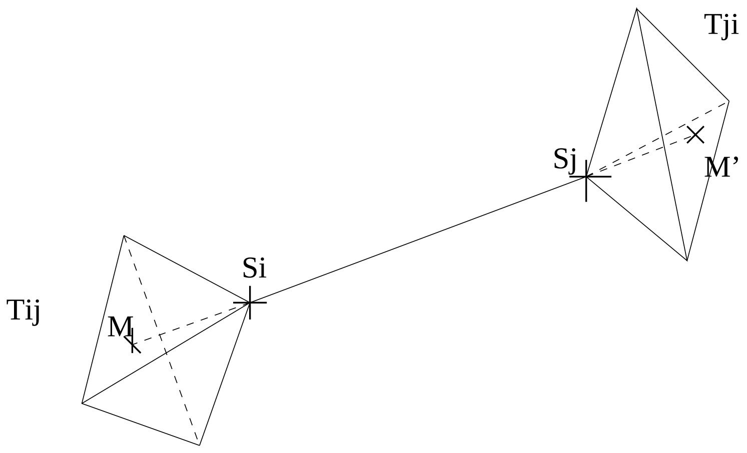


Figure 1.5: Downstream and upstream tetrahedra are tetrahedra having respectively S_i and S_j as vertex and such that line $\overline{S_iS_j}$ intersects the opposite face.

1.2.5 Time discretization

The semi-discrete equations (1.20) can be discretized in time using a simple Euler scheme (for a first-order time accuracy) as follows

$$\frac{\delta \tilde{\mathbf{W}}^{n+1}}{\Delta t^n} + \Psi(\tilde{\mathbf{W}}^n) = \mathbf{0}$$

where $\delta \tilde{\mathbf{W}}^{n+1} = \tilde{\mathbf{W}}^{n+1} - \tilde{\mathbf{W}}^n$ and $\tilde{\mathbf{W}}^n$ is the approximation of $\mathbf{W}(t^n)$. Nevertheless, explicit time integration procedures are subject to a stability condition expressed in terms of a CFL (Courant-Friedrichs-Lewy) number. An efficient time advancing strategy can be obtained by means of an implicit linearized formulation such as the one described in [Fézoui and Stoufflet, 1989] and briefly outlined here. First, the implicit variant of the equation above writes as:

$$\frac{\delta \tilde{\mathbf{W}}^{n+1}}{\Delta t^n} + \Psi(\tilde{\mathbf{W}}^{n+1}) = \mathbf{0}\ .$$

Then, applying a first-order linearization through differentiation of the flux $\Psi(\tilde{\mathbf{W}}^{n+1})$ the following Newton-like formulation is obtained:

$$\left(\frac{1}{\Delta t^n}\mathbf{I} + \left.\frac{\partial \Psi}{\partial \mathbf{W}}\right|_{\tilde{\mathbf{W}}^n}\right)\delta \tilde{\mathbf{W}}^{n+1} = -\Psi(\tilde{\mathbf{W}}^n) \tag{1.28}$$

where $\mathbf{I}$ is the identity matrix. Note that the resulting Euler implicit scheme tends to an iteration of Newton algorithm when the time step $\Delta t^n \to \infty$. As a consequence, one can ensure that this formulation will yield a quadratically converging method if very large time steps can be used.

If a first-order accuracy (both in space and time) is required, the first-order flux $\Psi^{(1)}$ and its Jacobian should be used

$$A(\tilde{\mathbf{W}}^n)\delta \tilde{\mathbf{W}}^{n+1} = \left(\frac{1}{\Delta t^n}\mathbf{I} + \left.\frac{\partial \Psi^{(1)}}{\partial \mathbf{W}}\right|_{\tilde{\mathbf{W}}^n}\right)\delta \tilde{\mathbf{W}}^{n+1} = -\Psi^{(1)}(\tilde{\mathbf{W}}^n)\ .$$

On the other hand, in the case of schemes of second-order spatial accuracy, a possible approach usually used (see [Fézoui and Stoufflet, 1989]) is to avoid the computation of the exact Jacobian of the second-order flux $\Psi^{(2)}$. Indeed, apart the fact that to define its exact expression is a more tedious task then the case of the first-order Jacobian (but this issue can be overcome using Automatic Differentiation, see Chapter 3), the biggest difficulty is the computational effort required in terms of memory (for the storage of the Jacobian matrix) and CPU time (for its inversion), which both increase notably with the order of accuracy. Thus we replace the exact Jacobian of the second-order flux $\frac{\partial \Psi^{(2)}}{\partial \mathbf{W}}\big|_{\tilde{\mathbf{W}}^n}$ by the Jacobian matrix $\frac{\partial \Psi^{(1)}}{\partial \mathbf{W}}\big|_{\tilde{\mathbf{W}}^n}$ resulting from the analytical differentiation of the first-order flux with respect to the cell-averaged states $\tilde{\mathbf{W}}^n$ (therefore we consider the first-order Jacobian matrix as an approximate second-order Jacobian)

$$A(\tilde{\mathbf{W}}^n)\delta \tilde{\mathbf{W}}^{n+1} = -\Psi^{(2)}(\tilde{\mathbf{W}}^n).$$

The matrix $A(\tilde{\mathbf{W}}^n)$ is structurally-symmetric block-sparse and has the suitable properties (diagonal dominance in the scalar case) allowing the use of relaxation procedure (e.g. associated with Jacobi or Gauss-Seidel iterations) in order to solve the involved linear system. The above implicit time integration approach, which is a particular case of defect-correction technique [Barrett et al., 1988; Koren, 1988; Skeel, 1981; Stetter, 1978] is well suitable for steady flows calculations (we refer to [Désidéri and Hemker, 1995] for a study of convergence properties); for unsteady flow computation, this first-order time accurate scheme is generally unacceptably dissipative. Note that second-order defect-correction schemes for unsteady problems have been also developed (see [Martin and Guillard, 1996])

Nevertheless, in the following, we will refer only to the steady solution of the equation (1.20).

1.3 Synthesis

The numerical algorithm presented here will be used in the following chapters for the application of a gradient-based algorithm for multilevel optimization (Chapter 4) and for the application of some Automatic Differentiaton algorithms to compute the gradient and Hessian of a functional in which the flow variables are used (Chapter 3). From the description made in this chapter we emphasize two aspects:

- if we use the Roe's scheme (this is the case for our 3D code) the numerical scheme is not differentiable. The same remark applies to the Steger-Warming fluxes used for the farfield boundary treatment;
- the scheme is implicit and relies on a preconditioned iteration in which a linearization of the first-order accurate numerical flux is used. This operator presents the two interesting properties of
  - compactness (only neighbouring nodes are linked by non-zero coefficients);
  - positivity (extended version of diagonal dominance).

These aspects will have to be considered in the AD developments.

Chapter 2

Uncertainty Analysis and Robust Design

2.1 Introduction

Due to the high complexity reached by computational fluid dynamics codes combined with the rapid advance of computational power, research in the field of the aerodynamical shape design has experienced a large development in the last years, allowing to deal with more and more complex optimization problems. However, high fidelity models are generally directly used only in deterministic design loops, which assume a perfect knowledge of the environmental and operational parameters. In reality, uncertainty can arise in many aspects of the entire design-production-operational process: from the assumptions done in the mathematical model describing the underlying physical process, to the manufacturing tolerances, and to the operational parameters and conditions that could be affected by unpredictable factors (e.g. atmospheric conditions). Exact and approximate techniques for propagating these uncertainties require additional computational effort but are progressively well established ([Putko et al., 2001], [Walters and Huyse, 2002], [Ghate and Giles, 2006]) and could be applied to many optimization problems in order to improve the robustness of the design (see for instance [Beyer and Sendhoff, 2007] for an excellent survey or [Huyse, 2001] for a shape optimization application). In addition, a systematic uncertainty analysis can lead to the identification of the key sources of uncertainty which merit further research, as well as the source of uncertainty that are not important with respect to a given response. The review proposed here is part of a contribution to the NODESIM-CFD European project, focused on non-deterministic simulation in CFD.

Before embarking on the study of these issues, a discussion on nomenclature is useful. We consider a modeling activity which tends to predict some events assuming some initial knowledge. We propose to adopt the definitions for uncertainty and error by the AIAA Guidelines [G-077-1998, 1998]:

Uncertainty: *"A potential deficiency in any phase or activity of the modeling process that is due to the lack of knowledge."*

Error: *"A recognizable deficiency in any phase or activity of modeling and simulation that is not due to lack of knowledge."*

The key words differentiating the definitions of uncertainty and error are *lack of knowledge*: since error is a recognizable deficiency, all errors could be corrected (at least in principle), so it is *deterministic*[1]. To the opposite, uncertainty is due to a lack of knowledge and therefore it cannot be eliminated, we can only quantify our "degree of ignorance" about the real value and analyze the impact that this lack of knowledge will have on the output of our model or simulation.

In a physical phenomena governed by PDE's (e.g. fluid dynamics), errors and uncertainties arising from the simulation can arise from different sources that can be grouped in four broad categories and many other sub-categories [Oberkampf and Blottner, 1998]:

- Physical Models
    - Inviscid flow
    - Viscous flow
    - Incompressible flow
    - Chemically reacting gas
    - Transitional/turbulent flow
    - Auxiliary physical models
        - Equation of state
        - Thermodynamic properties
        - Transport properties
        - Chemical models, rates
        - Turbulence model
    - Boundary conditions
        - Wall, e.g., roughness
        - Open, e.g., far-field
        - Free surface
        - Geometry representation
    - Initial conditions
- Discretization and solution
    - Truncation error (spatial and temporal)
    - Iterative convergence error

[1]The definition of error presented here is different than that an experimentalist may use, which is "the difference between the measured value and the exact value". Experimentalist usually define uncertainty as "the estimate of error". These definitions are inadequate for computational simulations because the exact value is typically not known.

- Round-off error
- Programmer and user error

In particular, for aerodynamical problems, it could be some operational uncertainties due to unpredictable factors that can alter the flow condition for a given geometry, for example:

- the angle of attack may vary during the flight due to atmospheric conditions
- instrumentation errors, or changes in flight profile compared to the scenario, or atmospheric condition could give uncertainty on Mach number
- variation in altitude due to instrumentation errors or changes in flight plan (or, again, variations of atmospheric conditions) could give uncertainty on Reynolds number.

It should be noted that geometric uncertainty, in addition to manifacturing tolerances, can arise during the working cycle also:

- temporary geometric variations, due to factors as icing or deformation under the weight-loads
- transient or permanent geometric variation due to the motion regime (e.g. blades in compressors)
- permanent geometric variation due to degradation (erosion,. . .).

Errors being not due to a lack of initial knowledge can be reduced. We shall be interested by truncation errors.

Uncertainty being due to a lack of initial knowledge must be modeled and propagated into the information processing.

We gather some of the existing methods to apply this programme and explain how perturbation methods can help in realizing it.

2.2 Uncertainty propagation techniques

In optimization problems, uncertainty propagation analysis concerns the study of the *cost functional*

$$j(\gamma) = J(\gamma, W) \in \mathbb{R} \tag{2.1}$$

where all varying parameters are represented by the *control variables* $\gamma \in \mathbb{R}^n$ and the state variables $W = W(\gamma) \in \mathbb{R}^N$ are solution of the (nonlinear) *state equation*

$$\Psi(\gamma, W) = 0 \in \mathbb{R}^N. \tag{2.2}$$

It is important to note that the state equation (2.2) contains the governing PDE of the mathematical model of the physical system of interest (for example the stationary part of the Euler or Navier-Stokes equations) and due to its nonlinearity is usually solved with iterative fixed-point methods (using, for example, Jacobian-free Newton-Krylov methods [Knoll and Keyes, 2004]).

Because of the complexity of the equations (2.2), the only way to study the behaviour of the functional (2.1) when the control γ is affected by uncertainties $\delta\gamma$ (here γ and its uncertainty $\delta\gamma$ are relative to a general control, i.e. the design/geometrical variables and/or the model parameters), is through *simulation* of these uncertainties and/or transforming the equations (2.1)-(2.2) in a way such the uncertainties are kept into account (introducing appropriate models or parameters that should characterize them).

The two main type of approaches for analyzing the propagation of uncertainties are the *deterministic* approach (mainly using the interval arithmetic) and the *probabilistic* one (Monte Carlo-like methods and the Method of Moments). Both approaches rely on the fact that we cannot assign a single value to a quantity affected by an uncertainty, but we should instead keep in account the different values that the quantities under consideration could have.

2.2.1 Interval arithmetics

The main idea with interval arithmetics is to assume uncertain parameters "unknow but bounded": every uncertain value is then described by an interval without a probability structure [Schweppe, 1973]. Then we can define an arithmetic defined on sets of intervals, rather than sets of real numbers [Moore, 1962]

$$\boldsymbol{x} \diamond \boldsymbol{y} = \{x \diamond y \mid x \in \boldsymbol{x} \text{ and } y \in \boldsymbol{y}\} \qquad \text{for } \diamond \in \{+, -, \times, \div\} \tag{2.3}$$

where $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\boldsymbol{y} = [\underline{y}, \overline{y}]$ with $\underline{x} \leq \overline{x}$ and $\underline{y} \leq \overline{y}$. Thus, the image of each one of the four basic interval operation is the exact image of the corresponding real operation. It's no difficult to see that the definition (2.3) is analogous to the operational definitions

$$\begin{aligned}
\boldsymbol{x} + \boldsymbol{y} &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
\boldsymbol{x} - \boldsymbol{y} &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\
\boldsymbol{x} \times \boldsymbol{y} &= [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}] \\
\frac{1}{\boldsymbol{x}} &= [1/\overline{x}, 1/\underline{x}] \quad \text{if } \underline{x} > 0 \text{ or } \overline{x} < 0 \\
\boldsymbol{x} \div \boldsymbol{y} &= \boldsymbol{x} \times \frac{1}{\boldsymbol{y}}
\end{aligned} \tag{2.4}$$

If such operations are composed, bounds on the ranges of real functions can be obtained [Kearfoot, 1996]. Since every function in the program used to solve the state equation (2.2) and compute the functional (2.1) can be expressed as composition of the four basic arithmetic operations, we can compute the interval bounds of the variables involved in the program, in particular the bounds of the interval for the ouput variable. This method is therefore fully deterministic, and interval result for uncertainty propagation represents maximal error bounds (i.e. *worst case result*).

It should be noted that intervals appearing in rational expression as denominator, cannot contain the values 0 (zero), otherwise result interval will be of the form $[a, \infty)$, $(-\infty, a]$ or $(-\infty, \infty)$ (correct from a mathematical point of view, but not very useful for computations).

The appealing feature about interval arithmetic is that it can be implemented in a existing CFD code in systematic way, with the aim of specific computational tool developed for different languages like FORTRAN or C/C++ but special attention should be paid to implementations due to the fact that the interval result depends on the order of the operation: in particular, for interval arithmetics the distributive property does not hold anymore [Moore, 1962].

The main criticism made to interval arithmetic approach for uncertainty propagation is that it can lead to overestimates the output intervals [Walters and Huyse, 2002]. Another criticism is that it does not assume any probabilistic structure for the uncertainty of input variables where, in real contexts involving measured data, one usually does (for example assuming that extremal values have low probability to occur, or assuming some correlations between the uncertainties). Some extensive references about this subject could be found at the page

`http://www.cs.utep.edu/interval-comp/`

2.2.2 Monte Carlo methods

Rewieving the existing literature, the most straightforward, general and accurate method to study the uncertainty propagation is full nonlinear Monte Carlo technique, see for example [Liu, 2001] or [Garzon, 2003]).

This method is not new: it is based on *statistical sampling* and an early application could be found in [Hall, 1873] where is used to determinate the value of π. Despite that the idea was quite old, the method could not be used for real problems until 1947, when the first electronical computer were available for researcher involved in the development of the nuclear weapons [Metropolis, 1987].

The basic idea behind Monte Carlo methods is to generate a sample of *independent states* of the input variables $\gamma_1, \gamma_2, \ldots, \gamma_{N_{mc}}$ with known distribution and analyze the distribution of the output $j(\gamma_1), j(\gamma_2), \ldots, j(\gamma_{N_{mc}})$. Despite the fact that this Monte Carlo algorithm is simple to implement (the solver can be used as a *black box*) and intrinsically parallel (any evaluation of $j(\gamma_i)$ is totally independent of the others), for CFD computation is prohibitively expensive in terms of CPU time required, due to the fact that to construct a *statistic*, i.e. computing a *Cumulative Distribution Function* (CDF) or a *Probability Density Function* (PDF) (see Appendix A), for the γ_i we need a rather high number of (costly) nonlinear simulations (tipically $N_{mc} > 10^3$). Therefore there is a need for computationally inexpensive high fidelity methods for uncertainty propagation.

2.2.3 Method of Moments

In the previous section, we have seen that a full nonlinear Monte Carlo method gives us complete and exact information about uncertainty propagation in the form of its PDF (or CDF), but with a prohibitively expensive cost in terms of CPU time. To reduce the computational cost, we may think to use only some (derivate) quantities characterizing the distribution of the input variables instead of an entire sample drawn from a population with a given PDF. Therefore, the idea behind the Method of Moments is based on the Taylor series expansion of the original nonlinear functional (2.1) around the *mean value* of the input control ($\mu_\gamma = E[\gamma]$), and then computing some statistical moments of the output (usually mean and variance). In this way, we are assuming that the input control γ can be decomposed as sum of a fully deterministic quantity

μ_γ with a stochastic perturbation $\delta\gamma$ with the property $E[\delta\gamma] = 0$. With these definitions, the Taylor series expansion of the functional $j(\gamma)$ around the mean value μ_γ is

$$j(\gamma) = j(\mu_\gamma + \delta\gamma) = j(\mu_\gamma) + G\delta\gamma + \frac{1}{2}\delta\gamma^* H\delta\gamma + O(||\delta\gamma||^3) \tag{2.5}$$

where $G = \left.\frac{\partial j}{\partial \gamma}\right|_{\mu_\gamma}$ is the gradient of the functional with respect to the uncertain variables and $H = \left.\frac{\partial^2 j}{\partial \gamma^2}\right|_{\mu_\gamma}$ is the Hessian matrix, both evaluated at the mean of the input variables μ_γ.

By considering various orders of the Taylor expansion (2.5) and taking the first and the second statistical moment, we can approximate the mean μ_j and the variance σ_j^2 of the functional $j(\gamma)$ in terms of its derivatives evaluated at μ_γ and in terms of statistical moments of the control γ (see the Appendix A for the definitions and Appendix B for the computation using the Taylor expansion).

First-order moment methods:

$$\begin{cases} \mu_j = j(\mu_\gamma) + O\Big(E\big[\delta\gamma^2\big]\Big) \\ \sigma_j^2 = E\Big[\big(G\delta\gamma\big)^2\Big] + O\Big(E\big[\delta\gamma^3\big]\Big) \end{cases} \tag{2.6}$$

Second-order moment methods:

$$\begin{cases} \mu_j = j(\mu_\gamma) + \frac{1}{2}E\big[\delta\gamma^* H\delta\gamma\big] + O\Big(E\big[\delta\gamma^3\big]\Big) \\ \sigma_j^2 = E\Big[\big(G\delta\gamma\big)^2\Big] + E\big[\big(G\delta\gamma\big)\big(\delta\gamma^* H\delta\gamma\big)\big] - \frac{1}{4}E\big[\delta\gamma^* H\delta\gamma\big]^2 + \\ \qquad + \frac{1}{4}E\Big[\big(\delta\gamma^* H\delta\gamma\big)^2\Big] + O\Big(E\big[\delta\gamma^4\big]\Big) \end{cases} \tag{2.7}$$

With this method it is clear that we are using only some partial informations about the input uncertainties, in fact we are using only some statistical moments of the control variable instead of full information available with its PDF, and we will not have anymore the PDF of the propagated uncertainty, but only its approximate mean and variance. Another important point is that the Method of Moments is applicable only for *small* uncertainties, due to the local nature of Taylor expansion approximation.

Two things should be noted here: the first one is that for applying the Method of Moments, *we need the derivatives* of the functional with respect to the control variables affected by uncertainties: in particular we need the gradient for the first-order method, and gradient and Hessian for the second order method. Due to the fact that $j(\gamma) = J(\gamma, W)$, where $W = W(\gamma)$ is solution of the state equation (2.2) we have for the derivative

$$\frac{dj}{d\gamma} = \frac{\partial J}{\partial \gamma} + \frac{\partial J}{\partial W}\frac{\partial W}{\partial \gamma}$$

Since we know the solution $W(\gamma)$ by its numerical values as result of a program (implementing an appropriate method, e.g. fixed point method), it is a mandatory task the use of Automatic

Differentiation tools (like TAPENADE, [Hascoët and Pascual, 2004]) in order to obtain the needed derivatives. The same remarks apply to the computation of the Hessian matrix. In particular we note that the derivatives are computed at the mean value of the control μ_γ, so they are fully deterministic and can be picked out from the expectations in the equations (2.6) or (2.7). In other words we can write

$$\begin{aligned}
E\Big[\big(G\delta\gamma\big)^2\Big] &= \sum_{i,k} G_i G_k E\big[\delta\gamma^{(i)}\delta\gamma^{(k)}\big] = \sum_{i,k} G_i G_k C_{ik} \\
E\big[\delta\gamma^* H\delta\gamma\big] &= \sum_{i,k} H_{ik} E\big[\delta\gamma^{(i)}\delta\gamma^{(k)}\big] = \sum_{i,k} H_{ik} C_{ik} \\
E\big[\big(G\delta\gamma\big)\big(\delta\gamma^* H\delta\gamma\big)\big] &= \sum_{i,k,l} G_l H_{ik} E\big[\delta\gamma^{(i)}\delta\gamma^{(k)}\delta\gamma^{(l)}\big] \\
E\Big[\big(\delta\gamma^* H\delta\gamma\big)^2\Big] &= \sum_{i,k,l,m} H_{ik} H_{lm} E\big[\delta\gamma^{(i)}\delta\gamma^{(k)}\delta\gamma^{(l)}\delta\gamma^{(m)}\big]
\end{aligned} \tag{2.8}$$

where $G_i = \dfrac{dj}{d\gamma^{(i)}}\Big|_{\mu_\gamma}$ are the elements of the gradient, $H_{ik} = \dfrac{d^2 j}{d\gamma^{(i)} d\gamma^{(k)}}\Big|_{\mu_\gamma}$ are the elements of the Hessian matrix and $C_{ik} = E\big[\delta\gamma^{(i)}\delta\gamma^{(k)}\big] = \mathrm{cov}(\gamma^{(i)}, \gamma^{(k)})$ are the elements of the *covariance matrix.* Every expectation term $E[\dots]$ in the equations (2.8), is defined by the statistical model of the uncertainties and could be computed in a preprocessing phase.

For example, for the important case where the uncertainties are random and normally distributed, we have:

$$\begin{aligned}
E\big[\delta\gamma^{(i)}\delta\gamma^{(k)}\delta\gamma^{(l)}\big] &= 0 \\
E\big[\delta\gamma^{(i)}\delta\gamma^{(k)}\delta\gamma^{(l)}\delta\gamma^{(m)}\big] &= C_{ik}C_{lm} + C_{il}C_{km} + C_{im}C_{kl}
\end{aligned}$$

and if these (normal) uncertainties are independents, then holds the relation $C_{ik} = \sigma_i^2 \delta_{ij}$ where $\sigma_i^2 = E\big[\delta\gamma^{(i)}\delta\gamma^{(i)}\big]$ and the equations (2.8) become

$$\begin{aligned}
E\Big[\big(G\delta\gamma\big)^2\Big] &= \sum_i G_i^2 \sigma_i^2 \\
E\big[\delta\gamma^* H\delta\gamma\big] &= \sum_i H_{ii}\sigma_i^2 \\
E\big[\big(G\delta\gamma\big)\big(\delta\gamma^* H\delta\gamma\big)\big] &= 0 \\
E\Big[\big(\delta\gamma^* H\delta\gamma\big)^2\Big] &= \sum_{i,k} \big(H_{ii}H_{kk} + 2H_{ik}^2\big)\sigma_i^2\sigma_k^2
\end{aligned} \tag{2.9}$$

The second comment to do for the Method of Moments, concerns the equation of the variance for second-order moment (2.7): although we have taken into account the term $E\big[\big(\delta\gamma^* H\delta\gamma\big)^2\big]/4$, the error is still of the order of $E\big[\delta\gamma^4\big]$. This is because the other terms of the same kind require the knowledge of order of derivatives higher than the second. In fact, it can be shown (see Appendix B) that the fourth order term in $O(E\big[\delta\gamma^4\big])$ depends on the third derivative of the functional.

From the previous discussion, it is clear that in order to apply the Method of Moments we need to solve only one (expensive) nonlinear system with derivatives (at the mean μ_γ) and then apply the (inexpensive) equations (2.6) or (2.7) where, for the fully nonlinear Monte Carlo approach of the previous section, we need to solve $N \gg 1$ nonlinear systems (2.2).

An application of this method for Aerospace applications can be found in [Putko et al., 2001].

2.2.4 Inexpensive Monte Carlo

This method, developed by [Ghate and Giles, 2006], is based on the idea of adjoint error correction as proposed by [Pierce and Giles, 2000] and it could be viewed at midpoint of Monte Carlo and Method of Moments.

As usual, let $W = W(\gamma)$ the solution of the state equation

$$\Psi(\gamma, W) = 0 \tag{2.10}$$

and $j(\gamma) = J(\gamma, W)$ a smooth functional. The discrete adjoint equation corresponding to the last equation is

$$\left(\frac{\partial \Psi}{\partial W}\right)^* \Pi = \left(\frac{\partial J}{\partial W}\right)^* \tag{2.11}$$

where Π is the adjoint solution. The key here is to perform a Taylor series expansion for the functional *and* for the state equation with respect to the state variables W, and then using the adjoint equation (2.11).

For the functional, the first-order Taylor expansion with respect to the generic state W_0 is

$$J(\gamma, W) = J(\gamma, W_0) + \frac{\partial J}{\partial W}\Big|_{(\gamma, W_0)}(W - W_0) + O\big(||W - W_0||^2\big)$$

Using the adjoint equation (2.11), we have

$$J(\gamma, W) = J(\gamma, W_0) + \left(\Pi^* \frac{\partial \Psi}{\partial W}\right)\Big|_{(\gamma, W_0)}(W - W_0) + O\big(||W - W_0||^2\big) \tag{2.12}$$

A first-order Taylor expansion of the state equation (2.10), gives us

$$0 = \Psi(\gamma, W) = \Psi(\gamma, W_0) + \frac{\partial \Psi}{\partial W}\Big|_{(\gamma, W_0)}(W - W_0) + O\big(||W - W_0||^2\big)$$

Replacing the first-order derivative in the (2.12) with the analogous term in the last equation, we obtain

$$J(\gamma, W) = J(\gamma, W_0) - \big(\Pi^* \Psi\big)\big|_{(\gamma, W_0)} + O\big(||W - W_0||^2\big) \tag{2.13}$$

where the state equation Ψ and the adjoint solution Π^* are both evaluated at the point (γ, W_0). Finally if we use an approximate adjoint solution $\Pi^* = \widetilde{\Pi}^* + O(||\Pi - \widetilde{\Pi}||)$ instead of the exact one, we have

$$J(\gamma, W) = J(\gamma, W_0) - \widetilde{\Pi}^* \Psi(\gamma, W_0) + O\big(||W - W_0||^2, ||W - W_0||\ ||\Pi - \widetilde{\Pi}||\big) \tag{2.14}$$

From the equation (2.11) we can see the adjoint solution Π as a function of the variables γ and W, then, remembering that the state variables depend on the control ($W = W(\gamma)$) through the state equation (2.10) we have

$$\Pi(\gamma, W) = \Pi(\gamma, W(\gamma)) = \pi(\gamma)$$

Thus, if we expand at the first-order the latter equation around the mean value of the control μ_γ (as we did in the previous section for the method of moment), we obtain

$$\Pi(\gamma, W) = \pi(\mu_\gamma) + O(||\delta\gamma||) = \Pi(\mu_\gamma, W(\mu_\gamma)) + O(||\delta\gamma||) \tag{2.15}$$

where $\delta\gamma$ is the stochastic perturbation defined in Sec. 2.2.3 and where $\pi(\mu_\gamma) = \Pi(\mu_\gamma, W(\mu_\gamma))$ is the solution of the adjoint system (2.11) with the derivatives computed at the point $(\mu_\gamma, W(\mu_\gamma))$. It is clear that considering Π as a function of the only variable γ and identifying the approximate adjoint solution $\widetilde{\Pi}$ with $\pi = \Pi(\mu_\gamma, W(\mu_\gamma))$ we have $O(||\Pi - \widetilde{\Pi}||) = O(||\delta\gamma||)$, and the equation (2.14) becomes

$$J(\gamma, W) = J(\gamma, W_0) - \pi^* \Psi(\gamma, W_0) + O\big(||W - W_0||^2, ||W - W_0||\,||\delta\gamma||\big) \tag{2.16}$$

Now it only remains to decide how to choose the approximation W_0. As usual, we perform a first-order Taylor expansion of $W(\gamma)$ around μ_γ

$$W(\gamma) = W(\mu_\gamma) + \frac{dW}{d\gamma}\Big|_{\mu_\gamma} \delta\gamma + O(||\delta\gamma||^2)$$

then, we have two natural options for the choice of W_0

IMC1 $W_0 = W(\mu_\gamma)$ and then

$$J(\gamma, W) = J(\gamma, W(\mu_\gamma)) - \pi^* \Psi(\gamma, W(\mu_\gamma)) + O\big(||\delta\gamma||^2\big) \tag{2.17}$$

IMC2 $W_0 = W(\mu_\gamma) + \frac{dW}{d\gamma}\Big|_{\mu_\gamma} \delta\gamma$ and then

$$J(\gamma, W) = J(\gamma, W_0) - \pi^* \Psi(\gamma, W_0) + O\big(||\delta\gamma||^3\big) \tag{2.18}$$

where the first approach has an overall leading error of second-order, while the second approach has an overall leading error of third order.

Respect to the fully nonlinear Monte Carlo, where has to be solved one nonlinear system $\Psi(\gamma, W) = 0$ for each sample point $\gamma_i = \mu_\gamma + \delta\gamma_i$, in the Inexpensive Monte Carlo we evaluate the residual $\Psi(\gamma_i, W_0)$ for each sampling point, resulting in a computationally less expensive method.

To be clear, the algorithm is summarized as follows:

- choose the mean control μ_γ and solve the nonlinear system $\Psi(\mu_\gamma, W) = 0$
- solve the adjoint linear system $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\mu_\gamma, W(\mu_\gamma))}\right)^* \pi = \left(\frac{\partial J}{\partial W}\Big|_{(\mu_\gamma, W(\mu_\gamma))}\right)^*$

- (if we use the IMC2) compute the matrix $\left.\dfrac{\partial W}{\partial \gamma}\right|_{\mu_\gamma}$ (this is could be done solving n linear systems, where $\gamma \in \mathbb{R}^n$)
- for each $i = 1, \dots, N_{mc}$
  - construct the sampling point $\gamma_i = \mu_\gamma + \delta\gamma_i$
  - (if we use the IMC1) set $W_{0,i} = W(\mu_\gamma)$
  - (if we use the IMC2) compute the extrapolation of the state variables $W_{0,i} = W(\mu_\gamma) + \left.\dfrac{\partial W}{\partial \gamma}\right|_{\mu_\gamma} \delta\gamma_i$
  - compute $\Psi(\gamma_i, W_{0,i})$
  - compute the value $j(\gamma_i)$ using the equation (2.17) for IMC1 or the equation (2.18) for IMC2

Note that the relationship with the Method of Moments (equations (2.6)–(2.7)) can be established. Indeed, in eqs. (2.17)–(2.18) there are two terms (J and Ψ) as function of γ that could be expanded up to the first-order without increase the error order, i.e.

$$\begin{aligned} J(\gamma, W) &= J(\mu_\gamma, W(\mu_\gamma)) + \left.\frac{\partial J}{\partial \gamma}\right|_{(\mu_\gamma, W(\mu_\gamma))} \delta\gamma \; - \pi^* \Psi(\gamma, W(\mu_\gamma)) + O\big(||\delta\gamma||^2\big) \\ &= j(\mu_\gamma) + \left.\frac{\partial J}{\partial \gamma}\right|_{(\mu_\gamma, W(\mu_\gamma))} \delta\gamma - \pi^* \left.\frac{\partial \Psi}{\partial \gamma}\right|_{(\mu_\gamma, W(\mu_\gamma))} \delta\gamma + O\big(||\delta\gamma||^2\big) \end{aligned} \tag{2.19}$$

where we used the property $\Psi(\mu_\gamma, W(\mu_\gamma)) = 0$. Taking the expectation (and remembering that $E[\delta\gamma] = 0$) we have

$$\mu_j = j(\mu_\gamma) + O\Big(E\big[\delta\gamma^2\big]\Big) \tag{2.20}$$

i.e. the same equation for the mean of the functional obtained with the first-order moment method (2.6).

2.3 Robust design

Simulation-based design optimization in aerodynamics has been an active research topic for the last years and is now applied to industrial problems. For a long time, optimization exercises were carried out neglecting uncertainty. However, application of such deterministic methods can lead to unexpected performance losses and, in practice, to unacceptable results. Indeed, the prescribed optimized design is subject to inherent geometrical variations due to manufacturing tolerances. Moreover, as we have seen in Section 2.1, operating conditions (such as Mach number or angle of attack) are subject to variability and random fluctuations. As consequence of such geometrical and operational uncertainties, the fitness of the optimal design predicted by CFD is usually not obtained in practice. To overcome this difficulty, robust-based design methods are developed in order to minimize the performance loss due to everyday fluctuations.

Remark 2.1. In order to specify the different approaches used for geometrical/design uncertainties and operative condition fluctuation, we refer them with a different notation with respect to the previous sections. In the following the geometrical/design variables will be denoted by γ, while the variables describing the operative conditions will be denoted by α.

Robust design can be regarded as an optimization approach which tries to account for uncertainties, generally through the construction of a robust counterparts j^R of the original performance or objective functional j.

For example in [Su and Renaud, 1997], the authors use for their robust optimization strategy, a sensitivity information for the objective functional

$$j^R(\gamma) = j(\gamma) + \varepsilon \sqrt{\sum_i \left(\frac{dj}{d\gamma^{(i)}} \sigma_i \right)^2}$$

that corresponds to the functional $j^R(\gamma) = j(\gamma) + \varepsilon\sigma_j$ when a first-order moment method is used for the approximation of the variance σ_j^2 (and the independence between the uncertain variables is assumed). Moreover, if we assume that $\sigma_i = \sigma$ for each i (and this is reasonable for geometrical uncertainties that arise from manifacturing tolerances), the above expression writes as

$$j^R(\gamma) = j(\gamma) + \varepsilon\sigma \sqrt{\sum_i \left(\frac{dj}{d\gamma^{(i)}} \right)^2} = j(\gamma) + \varepsilon\sigma || \frac{dj}{d\gamma} ||. \tag{2.21}$$

Therefore, if we are interested to build a gradient-based algorithm for optimization, we need to evaluate the gradient of the robust functional j^R, namely

$$\left(\frac{dj^R}{d\gamma} \right)^* = \left(\frac{dj}{d\gamma} \right)^* + \varepsilon\sigma \frac{H \left(\frac{dj}{d\gamma} \right)^*}{|| \frac{dj}{d\gamma} ||} \tag{2.22}$$

where H is the Hessian matrix of the functional j with respect to the variable γ. It is clear that to build the robust functional j^R we need to know the gradient $\left(\frac{dj}{d\gamma}\right)^*$ of the original functional j. Moreover, to compute the gradient (2.22) we need to evaluate the Hessian-by-vector multiplication $H\left(\frac{dj}{d\gamma}\right)^*$ (an algorithm for this purpose is given by the Lemma 3.1 in Section 3.6.2).

Another possibility to achieve robustness is to adopt the statistical approach according to the Von Neumann-Morgenstern decision theory [Bandemer, 2006; Huyse et al., 2002], in which the best choice in presence of uncertainty is to select the design which leads to the best expected fitness. This is commonly known as the Maximum Expected Value (MEV) criterion.

As we have seen, a deterministic simulation-based shape optimization problem consists in minimizing (or maximizing) a functional $J\colon (\gamma, W) \mapsto J(\gamma, W)$, which depends on the shape Γ (parametrized by the design variables γ) and on the state variable $W\colon (\gamma, \alpha) \mapsto W(\gamma, \alpha)$ which is implicitly defined through the state equation $\Psi(\gamma, \alpha, W) = 0$. Here the symbol α denotes the variables that define the operational conditions, e.g. the Mach number or the angle of attack. In this context, the usual design problem for the functional J (and for a given operational condition α_0) writes as

$$\begin{cases} \text{find } \gamma^* \text{ such that } j(\gamma^*) = \min_{\gamma} j(\gamma) \\ \text{where } j(\gamma) = J(\gamma, W) \text{ subject to } \Psi(\gamma, \alpha_0, W) = 0 \end{cases}$$

The non-deterministic MEV approach in the case of geometrical uncertainties becomes

$$\begin{cases} \text{find } \gamma^* \text{ such that } \mu_j(\gamma^*) = \min_{\bar{\gamma}} \mu_j(\bar{\gamma}) \\ \text{where } \mu_j(\bar{\gamma}) = \displaystyle\int J(\gamma, W) f_{\bar{\gamma}}(\gamma)\, d\gamma \ \text{ subject to } \Psi(\gamma, \alpha_0, W) = 0 \end{cases} \tag{2.23}$$

and for the operational uncertainty writes as

$$\begin{cases} \text{find } \gamma^* \text{ such that } \mu_j(\gamma^*) = \min_{\gamma} \mu_j(\gamma) \\ \text{where } \mu_j(\gamma) = \displaystyle\int J(\gamma, W) f_{\bar{\alpha}}(\alpha)\, d\alpha \ \text{ subject to } \Psi(\gamma, \alpha, W) = 0 \end{cases} \tag{2.24}$$

where $f_{\bar{\gamma}}$, $f_{\bar{\alpha}}$ are the Probability Density Function (PDF) for the uncertain variables γ, α with expected values $\bar{\gamma}$, $\bar{\alpha}$ respectively. With these definitions, the integral in (2.23)–(2.24) is the expectation value for the functional $J(\gamma, W)$ with respect to a probability measure for the uncertain variable.

The robust design problem is now considered within a rigorous statistical framework. This allows to take into account the random fluctuations of the fitness in the optimization problem, but also take care about the probability of occurrence of the events, via the PDFs. However, problems (2.23)–(2.24) does not address the variability of the fitness. For engineering problems, one also would like to select a design for which the fitness is not subject to a large variations. Then, a second criterion is often joined to the MEV criterion, that relies on the minimization of the variance of the functional

$$\begin{cases} \sigma_j^2(\bar{\gamma}) = \displaystyle\int J^2(\gamma, W) f_{\bar{\gamma}}(\gamma)\, d\gamma - \mu_j^2(\bar{\gamma}) \\ \text{where } \mu_j(\bar{\gamma}) = \displaystyle\int J(\gamma, W) f_{\bar{\gamma}}(\gamma)\, d\gamma \ \text{ subject to } \Psi(\gamma, \alpha_0, W) = 0 \end{cases} \tag{2.25}$$

for the problem (2.23) or

$$\begin{cases} \sigma_j^2(\gamma) = \displaystyle\int J^2(\gamma, W) f_{\bar{\alpha}}(\alpha)\, d\alpha - \mu_j^2(\gamma) \\ \text{where } \mu_j(\gamma) = \displaystyle\int J(\gamma, W) f_{\bar{\alpha}}(\alpha)\, d\alpha \ \text{ subject to } \Psi(\gamma, \alpha, W) = 0 \end{cases} \tag{2.26}$$

for the problem (2.24).

The problem of finding a design γ^* that minimizes both the mean value and the variance might not have any solution at all (in fact they can represent conflicting goals [Beyer and Sendhoff, 2007]), so we are forced to decide about the trade-off between the expected fitness and the expected fitness variation, choosing for example between the different optimal designs in the Pareto frontier, or defining a robust functional that is the convex combination of the mean and the standard deviation (i.e. the square root of the variance), namely $j^R(\gamma) = (1-\beta)\mu_j(\gamma) + \beta\sigma_j(\gamma)$, with $0 \leq \beta \leq 1$.

The practical difficulty with formulations (2.23)–(2.25) or (2.24)–(2.26) is that the integrations under the probability measure ($f_{\bar{\gamma}}$ or $f_{\bar{\alpha}}$) that define the mean and the variance of the objective function, are required in each of the optimization steps (for each design). Since in CFD simulations the nonlinear equation $\Psi = 0$ is computationally expensive to solve (and thus the evaluation of the objective function J), the use of a brute–force numerical integration method or Monte-Carlo analysis would become prohibitively expensive. Therefore, we need to introduce some techniques that permit us to have an an approximate expectation value and variance of the functional.

A possible approach for the above integration of is with the use of *metamodels* [Duvigneau, 2007], in which the functional $J(\gamma, W)$ in (2.24) is approximated with a function that depends only on the uncertain variables and whose evaluations are inexpensive.

Another possible approach for the integration of (2.23)–(2.24) and/or (2.25)–(2.26) is through the application of the Methods of Moment presented in Section 2.2.3 (see also [Huyse et al., 2002; Beyer and Sendhoff, 2007]). In other words, we perform a Taylor series expansion of the functional $J(\gamma, W)$ inside the above integrals around the mean value of the uncertain variable (that could be the design variable γ or the operative condition α, in fact we need to remember that $W\colon (\gamma,\alpha) \mapsto W(\gamma,\alpha)$ is implicitly defined by the state equation $\Psi(\gamma,\alpha,W)$). Therefore, we can compute the integrals that define the mean and variance of the objective function j, i.e. μ_j and σ_j^2, as Taylor expansion of the objective around the (deterministic) mean value of the uncertain variable: for this purpose we need to have first- and second-order derivatives of the functional j. As final remark, we want to emphasize the fact that using this approach we need to solve the nonlinear state equation $\Psi = 0$ only one time, resulting in a less expensive method with respect to the Monte-Carlo integration.

Chapter 3

First- and Second-Order Derivatives with Automatic Differentiation

3.1 Introduction to sensitivity analysis

Sensitivity analysis consists of a set of tools that can be utilized in the context of optimization, optimal design, uncertainty analysis, robust design, or simply system analysis to assess the influence of *control parameters* γ (input) on the state of the system (output) [Stanley and Stewart, 2002]. In all these contexts, and regardless the goal that has to be accomplished, the common ingredients are the presence of a *functional* j (output) that depends on some control parameters and possibly subject to satisfy some *constraints* (see Figure 3.1 for an example).

In this sense we consider the (discretized) steady Euler equations

$$\Psi(\gamma, W) = 0$$

introduced in Chapter 1, as the equality constraint for a functional (that could be the lift, the drag, or whatever quantity of interest)

$$j\colon \gamma \mapsto j(\gamma) := J(\gamma, W) \in \mathbb{R}$$

where $\gamma \in \mathbb{R}^n$ and $W = W(\gamma) \in \mathbb{R}^N$. In other words, the state variable $W\colon \gamma \mapsto W(\gamma)$ is an implicit function of the control γ through the equation $\Psi = 0$. To be more specific regarding the control γ, its definition depends on the problem at hand: if we are dealing with optimal shape design problems (Chapter 4) , γ could be a parametrization of the physical wall boundary Γ_B that we want to optimize (see Figure 1.1); if we are interested in the study of the effect of the operational conditions on the functional, γ could be for example the Mach number or the angle of attack (Chapter 2), etc.

Then, the sensitivities can be used like building blocks of different methods and techniques developed in order to solve a specific problem: for example they can be used for gradient-based algorithms in order obtain a local extrema of the functional (e.g. multilevel optimization and optimal design, Chapter 4), or used to give some informations on the output when the control is affected by a lack of knowledge (uncertainty analysis and robust design, Chapter 2). Again, the sensitivities can be used in order to improve the accuracy of the functional itself using the adjoint-correction approach by [Pierce and Giles, 2004, 2000] (in Chapter 5 we develop a strategy to compute the gradient of this corrected functional) or used for defect correction techniques [Barrett et al., 1988; Koren, 1988; Skeel, 1981; Stetter, 1978] or for mesh adaptation approach [Becker et al., 2000; Becker and Rannacher, 2001; Venditti and Darmofal, 2002], etc.

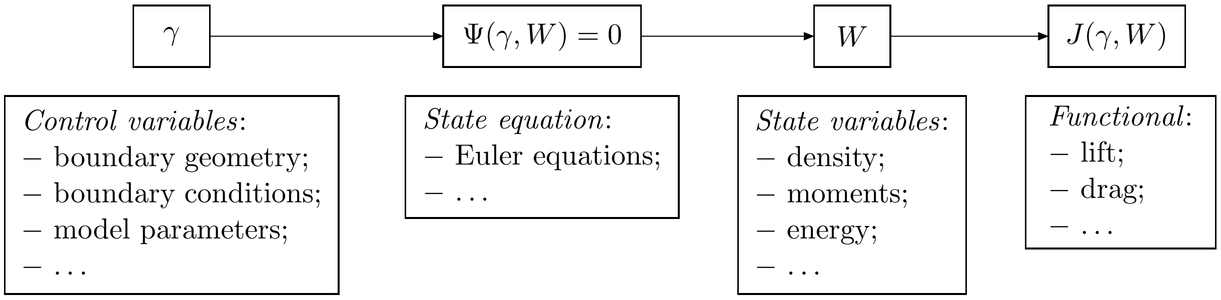


Figure 3.1: Definitions and relations between the building blocks for an optimal control problem. Starting from the definition of the control variable γ (input) we compute the state variable W solving the equation $\Psi(\gamma, W) = 0$ (that could be the Euler equations or any other mathematical model for the problem at hand). Then, the control γ and the state W are inputs for the evaluation of a given functional J.

Regardless the applications of the sensitivities, the tough question is: "How to obtain them?". From a general point of view and for a given problem at hand, maybe the most straightforward way is to go back to the equations that led to the program used to solve them: in other words we suppose that a certain result is defined by some equations, and the program solve these equations to obtain this result. Then one can write a new set of equations, whose solutions are the derivatives of the initial result (this task can be performed by computers using *symbolic differentiation programs*, like Maple[1] or Mathematica[2]). Consequently, one can write a new program that solves these new equations for the desired derivatives. This is mathematically very sound, but it probably is the most expensive way, since it implies the discretization of a new set equations, then we need to write a new program. We all know how difficult this is, and how many errors can be done! Moreover, in some cases there are no simple original equations, and only the program to solve them is at hand.

Therefore we are looking for another way, more economical and that possibly uses only the original program and not the underlying equations.

The last requirement is obviously satisfied if we decide to compute sensitivities using *divided differences.* For a given set of program's inputs, u, the program `P` compute a result $\Phi(u) = v$. In the general case, both u and v are vectors, i.e. are composed of several real numbers. Given now a direction δu in the space of the inputs, one can run the program `P` again, on the new input $u + \epsilon\delta u$ (or $u - \epsilon\delta u$), where ϵ is some very small positive number. Then an *approximation*

[1]`http://www.maplesoft.com/`
[2]`http://www.wolfram.com/`

of the directional derivative along δu is computed easily by:

$$\left.\frac{d\Phi}{du}\right|_{u} \delta u \simeq \frac{\Phi(u+\epsilon\delta u)-\Phi(u)}{\epsilon} \qquad \text{or} \qquad \left.\frac{d\Phi}{du}\right|_{u} \delta u \simeq \frac{\Phi(u)-\Phi(u-\epsilon\delta u)}{\epsilon}\ .$$

In order to have a better approximation, we can use the *centered divided differences*, namely:

$$\left.\frac{d\Phi}{du}\right|_{u} \delta u \simeq \frac{\Phi(u+\epsilon\delta u)-\Phi(u-\epsilon\delta u)}{2\epsilon}$$

but this approach cost an additional evaluation of the function Φ (and therefore an additional execution of the program `P`). Of course divided differences give us just an approximation of the derivatives: ideally, the exact derivative is the limit of these formulas, when ϵ tends to zero. But this makes no sense on a real computer, since very small values of ϵ lead to truncation errors, and therefore to erroneous derivatives. This is a serious drawback of divided differences: some tradeoff must be found between truncation errors and approximation errors. Finding the best ϵ requires numerous executions of the program, and even then the computed derivatives are just approximations. To be honest, we have to say that divided differences are the only way to compute derivatives when we have not access to the program source for `P` (e.g. the program is considered like a *black box*): divided differences approach requires to know only the result of the program on a given input and not how the result is obtained.

Another approach to compute derivatives is the *"complex variable method"* [Squire and Trapp, 1998; Newman III et al., 1998]. The idea relies on the Taylor expansion of the function Φ using a *complex* increment $i\epsilon\delta u$, namely

$$\Phi(u+i\epsilon\delta u) = \Phi(u) + i\epsilon\left.\frac{d\Phi}{du}\right|_{u}\delta u + \dots$$

and therefore the derivative is obtained as

$$\left.\frac{d\Phi}{du}\right|_{u} \delta u = \frac{\mathrm{Im}\big[\Phi(u+i\epsilon\delta u)\big]}{\epsilon} + O(\epsilon^2)$$

where the notation $\mathrm{Im}\big[\dots\big]$ denotes the imaginary part of a complex quantity. Thus, evaluating the function numerically with a complex argument, both the (real) function value and its derivative are obtained, without the subtractive cancellation errors present in finite difference approximation. Some work should be done by the user in the program in order to evaluate the function with complex arguments (whereas the original function had real arguments). The drawback of this approach remains that, as for the divided difference method, it gives us only *directional derivatives*, and therefore if Φ is a functional depending on n variables, to compute its *gradient* we need to evaluate the (complex-valued) functional Φ exactly n times.

A more flexible and efficient approach is given by Automatic Differentiation. In mathematics and computer algebra, Automatic Differentiation, or AD (sometimes alternatively called *algorithmic differentiation*), is a method to numerically evaluate the derivative of a function specified by a computer program. Automatic Differentiation solves all of the mentioned problems:

- it does not require the underlying equation but only its program implementation;
- the computed derivatives are *exact* (at machine level representation), i.e. we do not perform any approximation to compute the derivatives;
- it is (reasonably) *fast*: we don't need to perform any additional run of the program to tune parameters;
- some AD tools are able to compute the gradient of a functional at a cost that is independent from the number of input variables (see *Reverse mode* in the following Section 3.2).

AD exploits the fact that any computer program that implements a vector function $v = \Phi(u)$ (generally) can be decomposed into a sequence of elementary assignments, any one of which may be trivially differentiated by a simple table lookup. These elemental partial derivatives, evaluated at a particular argument, are combined in accordance with the chain rule from derivative calculus to form some derivative information for Φ (such as gradients, directional derivatives, the Jacobian matrix, etc.). This process yields *exact* (to numerical accuracy) derivatives. Because the symbolic transformation occurs only at the most basic level, AD avoids the computational problems inherent in complex symbolic computation. Moreover, as we will see, AD permits to obtain the *gradient* of a functional *in a direct way* and its computational cost is *independent from the dimension of the input variable*: this property is quite astonishing, in fact the usual way to compute gradients of functionals (e.g. with divided differences) is component-by-component evaluating the derivatives along the directions defined by the canonical basis, and therefore the cost is proportional to the dimension of the input space. The main drawback of AD approach is that it requires to know the *source* of the program `P` and needs a certain amount of work by the final user to perform a correct differentiation.

The rest of the chapter is devoted to the study of different approaches to compute first- and second-order derivatives of constrained functionals using Automatic Differentiation, and it refers in particular to the AD tool TAPENADE[3].

3.2 Principles of Automatic Differentiation

Given a program `P` computing a function Φ

$$\begin{aligned}\Phi\colon \mathbb{R}^n &\to \mathbb{R}^m \\ u &\mapsto v\end{aligned} \tag{3.1}$$

we want to build a program that computes the derivatives of Φ. Specifically, we want the derivatives of the *dependent* (i.e. some variables in v) with respect to the *independent* (i.e. some variables in u). For the sake of simplicity, let's assume that the program `P` is a *sequence* of instructions $\mathtt{I}_1, \mathtt{I}_2, \dots, \mathtt{I}_p$ that can be identified with a composition of functions, where each simple istruction $\mathtt{I}_k$ is a function $\phi_k\colon \mathbb{R}^{q_{k-1}} \to \mathbb{R}^{q_k}$. Thus we see $\mathtt{P} : \{\mathtt{I}_1, \mathtt{I}_2, \dots, \mathtt{I}_p\}$ as

$$u \mapsto v = \Phi(u) = \phi_p \circ \phi_{p-1} \circ \cdots \circ \phi_1(u)$$

[3]`http://www-sop.inria.fr/tropics/`

where each ϕ_k is the elementary function implemented by the instruction $\mathtt{I}_k$. In general, the functions implemented by the arithmetic of the programming language are indeed differentiable[4].

Finally, AD simply applies the chain rule to obtain derivatives of Φ

$$\begin{aligned}\left.\frac{\partial\Phi}{\partial u}\right|_u =&(\phi_p' \circ \phi_{p-1} \circ \cdots \circ \phi_1(u))\cdot\\ &\cdot(\phi_{p-1}' \circ \cdots \circ \phi_1(u))\cdot\\ &\cdot \cdots \cdot\\ &\cdot\phi_1'(u)\end{aligned}$$

or in a more compact way

$$\left.\frac{\partial\Phi}{\partial u}\right|_u = \phi_p'(w_{p-1})\phi_{p-1}'(w_{p-2})\cdots\phi_1'(w_0)$$

where $w_0 = u$ and $w_k = \phi_k(w_{k-1})$. The last equation can be mechanically translated back into a sequence of instructions $\mathtt{I}_k'$ and these sequences inserted back into a copy of the control of P, yielding a *differentiated program* P′ [Hascoët and Pascual, 2004].

We note here that if we are not interested in the full Jacobian (that is expensive to compute and store because its computation involves matrix-by-matrix multiplications), we can compute the derivatives along some directions using matrix-by-vector multiplications. In general, given a matrix $A \in \mathbb{R}^{n,m}$ and two (column) vector $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, we can perform the matrix-by-vector multiplication in a twofold manner: *by right*, i.e. computing $b = Ax$, $b \in \mathbb{R}^n$; or *by left*, i.e. computing $c^T = y^T A$, $c \in \mathbb{R}^m$ (and the multiplication can be viewed as the right multiplication using the transpose[5] matrix A^T, namely $c = A^T y$). In order to use the two multiplications, many Automatic Differentiation packages (TAPENADE [Hascoët and Pascual, 2004], OpenAD[6] [Utke, 2004], TAF[7] [Giering et al., 2005], ADOL-C[8] [Griewank et al., 1996]) have two differentiation modes:

- the **Tangent mode** applied to a routine computing Φ produces another routine computing, from u and from an arbitrary direction $\dot{u}$ (of same dimension as u), the derivative in direction $\dot{u}$:
$$u, \dot{u} \mapsto \left.\frac{\partial\Phi}{\partial u}\right|_u \dot{u} = \phi_p'(w_{p-1})\phi_{p-1}'(w_{p-2})\cdots\phi_1'(w_0)\dot{u} \tag{3.2}$$

[4]However in some rare case, this is not true. For example the square root is defined for the null value, and its derivative is not.

Moreover, due to the finite representation of numbers in computers, we could have problem not only for isolated values, but for intervals. In fact, if the derivative of a real function $f(x)$ is not finite at x_0 (i.e. $\lim_{x\to x_0^+} f'(x) = \pm\infty$ or $\lim_{x\to x_0^-} f'(x) = \pm\infty$) we cannot numerically represent the value of the derivative in the interval $[x_0 - \varepsilon^-, x_0 + \varepsilon^+]$ in which the value of $f'(x)$ is greater (smaller) than the maximum (minimum) represententable number.

[5]Due to the discrete approach used for this work, in the following we identify the adjoint with the transpose, namely $A^* = A^T$.

[6]`http://www-unix.mcs.anl.gov/~utke/OpenAD/`

[7]`http://www.fastopt.de/`

[8]`http://www.math.tu-dresden.de/~adol-c/`

In order to use matrix-by-vector multiplications, the obtained routine computes the derivatives in the same order as the initial routine computes the original values (right to left in the equation (3.2)) and its cost is $1 < \alpha_T < 4$ times greater with respect to the cost needed by the original routine to evaluate the function $\Phi(u)$ [Griewank, 2000]. We note that tangent mode delivers only one real number if Φ is a real valued functional.

- the **Reverse mode** when applied to the previous initial routine computing Φ produces a routine which computes, from u and from an arbitrary direction $\bar{\Phi}$ (of same dimension as $v = \Phi(u)$), the following product of same dimension as u:

$$u, \bar{\Phi} \mapsto \left(\frac{\partial \Phi}{\partial u}\bigg|_u \right)^* \bar{\Phi} = \phi_1'^*(w_0)\phi_{p-1}'^*(w_{p-2}) \cdots \phi_p'^*(w_{p-1})\bar{\Phi} \tag{3.3}$$

  For a functional of n variables, the routine produced by the reverse mode delivers n numbers and to compute the *gradient* of the functional, it can be (at least theoretically [Griewank, 2000]) n times more efficient than the tangent mode. In particular, the computation time required to evaluate $\left(\frac{\partial \Phi}{\partial u}\big|_u\right)^* \bar{\Phi}$ is only a small multiple α_R (usually $1 < \alpha_R \lesssim 5$ [Griewank, 2000]) of the run time to evaluate $\Phi(u)$, and *it is independent from the number of the input parameters n*, which can be very large.

  However, we observe in the equation (3.3) that, in order to use matrix-by-vector multiplications, the w_k are required in the *reverse* of their computation order. If the original program *overwrites* a part of w_k, the differentiated program must restore w_p before it is used by $\phi_{k+1}'^*(w_k)$. This is the main drawback of the Reverse mode of AD. The two classical strategies to cope with that are

  – Recompute-All (RA), in which the w_k are recomputed when needed, restarting the original (not-differentiated) program `P` on input w_0 until instruction $\mathtt{I}_k$. This strategy has a quadratic time cost with respect to the total number of run-time instruction p;

  – Store-All (SA), in which the w_k are restored from a stack when needed. This stack is filled (with a PUSH function) during a preliminary run (usually called *forward sweep*) of `P` that additionally stores variables on the stack just before they are overwritten, and then the differentiation take place in a *backward sweep* in which the stored values are taken from the stack (with a POP function) and used for the evaluation of (3.3). This strategy is used by TAPENADE ([Hascoët and Pascual, 2004]), and it has linear memory cost with respect to p.

A simple example of Tangent and Reverse differentiation with TAPENADE is given in Figure 3.2.

Remark 3.1. From the previous definitions, if Φ is a functional (i.e. $m = 1$), the tangent mode differentiation gives us a *directional derivative* while the reverse mode gives us a *gradient* (choosing $\bar{\Phi} = 1$).

To better understand how AD works in practice, we introduce a special notation (similar in some way to the notation in [Griewank et al., 2008]) that will help us for a correct implementation. First of all, we assume that the function

$$\begin{array}{llll} f: & \mathbb{R}^{n_x} \times \mathbb{R}^{n_y} & \to \mathbb{R}^m \\ & (x, y) & \mapsto f(x, y) \end{array}$$

is implemented by the routine `func(f,x,y)`, where `x` and `y` are the variables that should contain the $x \in \mathbb{R}^{n_x}$ and $y \in \mathbb{R}^{n_y}$ respectively, and `f` is the variables where will be stored the result $f(x,y) \in \mathbb{R}^m$ (i.e. `func(f,x,y)` is a "subroutine" in FORTRAN language or a "void function" in C/C++ terminology).

For differentiation, we need to know which are the independent variables (that are input parameters of the function) and which are the dependent variables (output parameters). For this purpose, we use an arrow over any parameter containing the independent variable and an arrow under any parameter containing the dependent variable, thus, if we want to specify that the routine `func(f,x,y)` (implementing the function $f(x,y)$) has `f` as output and `x`, `y` as input we write

$$\texttt{func(}\underset{\downarrow}{\texttt{f}}\texttt{,}\overset{\downarrow}{\texttt{x}}\texttt{,}\overset{\downarrow}{\texttt{y}}\texttt{)} \tag{3.4}$$

Furthermore, if we would specify that the function is evaluated with some specific values $x = x_0$ and $y = y_0$ we'll write

$$\texttt{func(}\underset{f(x_0,y_0)}{\texttt{f}}\texttt{,}\overset{x_0}{\texttt{x}}\texttt{,}\overset{y_0}{\texttt{y}}\texttt{)}$$

where the value written over a parameter means "value taken by the input variable" and the value under a parameter means "value stored in the output variable".

The last step is (to) specify which mode we use for differentiation and study the output generated by the AD tool (TAPENADE in our case [Hascoët and Pascual, 2004]) that performs the differentiation required. First of all, we must keep in account that for each independent variable with respect to which we differentiate, we'll have a corresponding dual variable that will be of the same kind (input) if we use Tangent mode, and of the opposite kind (output) if we use the Reverse mode. The same thing happens for the dependent (output) variables: the dual variables will be output variables in the case of Tangent mode and input variables for Reverse mode.

Using the notation previously introduced, Tangent mode differentiation for the case (3.4) respect all the independent variables `x` and `y` gives us

```
func_d(f,fd,x,xd,y,yd)
```

where the new parameters `fd` (output), `xd` (input) and `yd` (input) are the dual variables of `f`, `x` and `y` (the `d` character after the variables and function name means "*dot*"). If we give at the new input parameters `xd` and `yd` the values $\dot{x} \in \mathbb{R}^{n_x}$ and $\dot{y} \in \mathbb{R}^{n_y}$ respectively, we obtain

$$\texttt{func\_d(}\underset{f(x_0,y_0)}{\texttt{f}}\texttt{,}\underset{\left(\frac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)\dot{x}+\left(\frac{\partial f}{\partial y}\big|_{(x_0,y_0)}\right)\dot{y}}{\texttt{fd}}\texttt{,}\overset{x_0}{\texttt{x}}\texttt{,}\overset{\dot{x}}{\texttt{xd}}\texttt{,}\overset{y_0}{\texttt{y}}\texttt{,}\overset{\dot{y}}{\texttt{yd}}\texttt{)}$$

where the output parameter `fd` contains the value $\dot{f} = \left(\frac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)\dot{x} + \left(\frac{\partial f}{\partial y}\big|_{(x_0,y_0)}\right)\dot{y}$ with $\dot{f} \in \mathbb{R}^m$ (and the derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ are both evaluated at (x_0, y_0)).

Reverse mode differentiation of (3.4) gives us

```
func_b(f,fb,x,xb,y,yb)
```

where now we have a new input parameter `fb` and two new output parameters `xb` and `yb` (the `b` character after the variables and function name means "*bar*"). Storing the value $\bar{f} \in \mathbb{R}^m$ in the parameter `fb`, we obtain

$$\texttt{func\_b(}\underset{f(x_0,y_0)}{\texttt{f}}\texttt{,}\overset{\bar f}{\texttt{fb}}\texttt{,}\overset{x_0}{\texttt{x}}\texttt{,}\underset{\left(\frac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)^* \bar f}{\texttt{xb}}\texttt{,}\overset{y_0}{\texttt{y}}\texttt{,}\underset{\left(\frac{\partial f}{\partial y}\big|_{(x_0,y_0)}\right)^* \bar f}{\texttt{yb}}\texttt{)}$$

i.e. the output parameters `xb` and `yb` will contain the values $\bar{x}_f = \left(\frac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)^* \bar{f}$ and $\bar{y}_f = \left(\frac{\partial f}{\partial y}\big|_{(x_0,y_0)}\right)^* \bar{f}$ respectively (with $\bar{x}_f \in \mathbb{R}^{n_x}$, $\bar{y}_f \in \mathbb{R}^{n_y}$ and the derivatives $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ both evaluated at (x_0, y_0)). The subscript on the output variables stands to remind which is the differentiate function: this will be useful when we need to differentiate (in reverse-mode) routines having some common input variables.

For the case where we are differentiating with respect to only some independent variables (`x` for example) we obtain for Tangent mode

$$\texttt{func\_dx\_d(}\underset{f(x_0,y_0)}{\texttt{f}}\texttt{,}\underset{\dot{f}_x=\left(\frac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)\dot{x}}{\texttt{fd\_x}}\texttt{,}\overset{x_0}{\texttt{x}}\texttt{,}\overset{\dot{x}}{\texttt{xd}}\texttt{,}\overset{y_0}{\texttt{y}}\texttt{)} \tag{3.5}$$

where we use the notation $\dot{f}_x$ = `fd_x` (and the suffix `_dx` in the name of the differentiated routine) to specify that the Tangent mode differentiation is performed only with respect to the independent variable x. In the same manner, we have for the Reverse mode

$$\texttt{func\_dx\_b(}\underset{f(x_0,y_0)}{\texttt{f}}\texttt{,}\overset{\bar f}{\texttt{fb}}\texttt{,}\overset{x_0}{\texttt{x}}\texttt{,}\underset{\left(\frac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)^* \bar f}{\texttt{xb}}\texttt{,}\overset{y_0}{\texttt{y}}\texttt{)}\ . \tag{3.6}$$

Remark 3.2. Before continuing, it is important to introduce a notation that will help us to keep the formalism as simple as possible.

For the case of Tangent mode differentiation, we use a dot over the differentiated (dependent) output if the derivative is performed with respect only one independent variable (i.e. we are computing the partial derivative and not the total one) we use a pedix with the name of the variable we are considering. To be more clear, if $\Psi\colon (\gamma, W) \mapsto \Psi(\gamma, W)$ and $J\colon (\gamma, W) \mapsto J(\gamma, W)$, for Tangent mode differentiation we could have the following possibilities

$$\dot{\Psi}_\gamma = \left(\frac{\partial \Psi}{\partial \gamma}\right)\dot{\gamma} \quad ; \quad \dot{\Psi}_W = \left(\frac{\partial \Psi}{\partial W}\right)\dot{W} \quad ; \quad \dot{\Psi} = \left(\frac{\partial \Psi}{\partial \gamma}\right)\dot{\gamma} + \left(\frac{\partial \Psi}{\partial W}\right)\dot{W}$$

$$\dot{J}_\gamma = \left(\frac{\partial J}{\partial \gamma}\right)\dot{\gamma} \quad ; \quad \dot{J}_W = \left(\frac{\partial J}{\partial W}\right)\dot{W} \quad ; \quad \dot{J} = \left(\frac{\partial J}{\partial \gamma}\right)\dot{\gamma} + \left(\frac{\partial J}{\partial W}\right)\dot{W}\ .$$

The dotted quantities $\dot{\gamma}$ and $\dot{W}$ are the new input variables corresponding to the independent variables we are considering for the differentiation.

In the case of Reverse mode, due to the fact that the additional output variables are always derivatives performed with respect only one single variable, we use a bar over the independent variable we are considering and the name of the dependent variable as pedix. With the same definition for Ψ and J above, we could have

$$\bar{\gamma}_\Psi = \left(\frac{\partial \Psi}{\partial \gamma}\right)^* \bar{\Psi} \quad ; \quad \bar{W}_\Psi = \left(\frac{\partial \Psi}{\partial W}\right)^* \bar{\Psi}$$

$$\bar{\gamma}_J = \left(\frac{\partial J}{\partial \gamma}\right)^* \bar{J} \quad ; \quad \bar{W}_J = \left(\frac{\partial J}{\partial W}\right)^* \bar{J} \ ,$$

where $\bar{\Psi}$ and $\bar{J}$ are the new input quantities corresponding to the dependent variables Ψ and J respectively.

3.3 Matrix-free methods for solving linear systems in the AD context

Before going into the description of the algorithms to compute the first (or second) order derivatives of a constrained functional, we make a little disgression that will be very useful for the computation and that we will use extensively in the sequel.

Suppose that we want to solve a linear system of the kind

$$A\xi = b \quad \text{such that } A = \left(\tfrac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right) \text{ or } A = \left(\tfrac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)^* \tag{3.7}$$

where

$$\begin{array}{rcl} f\colon \mathbb{R}^{n_x} \times \mathbb{R}^{n_y} & \longrightarrow & \mathbb{R}^{n_x} \\ (x,y) & \longmapsto & f(x,y) \end{array} \tag{3.8}$$

is a differentiable function and $b \in \mathbb{R}^{n_x}$ is a known vector (from the previous definition A is a square matrix, namely $A \in \mathbb{R}^{n_x,n_x}$).

A possible strategy is to compute (with AD tools for example) and store the matrix A and then apply one of the many algorithms (direct or iterative) developed for solving linear systems.

However, with the analysis done in Section 3.2 to study how the application of the two AD modes on a routine looks like, we could perform the matrix-by-vector multiplication in (3.7) using the derivatives obtained by AD like in (3.5) or (3.6) and therefore we can solve the linear system (3.7) *without* storing the matrix A. In fact we could use iterative algorithms (like GMRES, see e.g. [Saad, 1996]) that do not need to know the matrix A, but only its effect on a given vector ξ_i, i.e. the matrix-by-vector multiplication $A\xi_i$ (we call such kind of algorithms *"matrix-free methods"*).

In other words, given a general linear solver that implements a matrix-free method, we need to replace any matrix-by-vector occurrence of the kind $A\xi_i$ with the corresponding routine (3.5) or (3.6), depending on the definition of A: we shall use the Tangent mode derivative (3.5) if $A = \left(\frac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)$ and the Reverse mode derivative (3.6) if $A = \left(\frac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)^*$.

Original	**Tangent**	**Reverse**
F: a, b, c $\mapsto$ r	$\dot{F}$: a, $\dot{a}$, b, $\dot{b}$, c, $\dot{c}$ $\mapsto$ r, $\dot{r}$	$\overline{F}$: a, b, c, $\overline{r}$ $\mapsto$ $\overline{a}$, $\overline{b}$, $\overline{c}$
	$\dot{x}$ = 0.0	
x = 2.0	x = 2.0	x = 2.0
	$\dot{r}$ = x*$\dot{a}$	
r = x*a	r = x*a	r = x*a
		PUSH(x)
	$\dot{x}$ += $\dot{c}$	
x += c	x += c	x += c
	$\dot{r}$ += $\dot{x}$*b+x*$\dot{b}$	
r += x*b	r += x*b	r += x*b
		PUSH(x)
	$\dot{x}$ = 0.0	
x = 3.0	x = 3.0	x = 3.0
	$\dot{r}$ += x*$\dot{c}$	
r += x*c	r += x*c	r += x*c
		$\overline{x}$= c*$\overline{r}$
		$\overline{c}$+= x*$\overline{r}$
		POP(x)
		$\overline{x}$= 0.0
		$\overline{x}$= b*$\overline{r}$
		$\overline{b}$+= x*$\overline{r}$
		POP(x)
		$\overline{c}$+= $\overline{x}$
		$\overline{x}$+= a*$\overline{r}$
		$\overline{a}$+= x*$\overline{r}$
		$\overline{x}$= 0.0

Figure 3.2: Tangent and Reverse differentiation on a small code. *Left*: Original code, *middle*: Tangent code, *right*: Reverse code. Tangent-differentiated variables are shown with a dot above, as in $\dot{x}$. Reverse-differentiated variables are shown with a bar above, as in $\overline{x}$. The tangent mode adds differentiate instruction *before* the original one, therefore the original structure of the program is maintained. The reverse mode performs a forward sweep (in which the stack is filled with PUSH functions) then a reverse sweep (in which the variable are taken from the stack with POP functions). The reverse code produced with TAPENADE is actually shorter because of static data-flow analysis: the code in light grey is stripped away, but this has no influence on the demonstration.

In our case, we choose the so-called GMRES-RCI, i.e. the GMRES implementation made by CERFACS [Frayssé et al., 2003] in which a Reverse Communication Interface is implemented [Dongarra et al., 1995]. The primary aim of Reverse Communication is to isolate the matrix-by-vector multiplications that the user supplies on data structures that are most natural to the problem at hand [Dongarra et al., 1995], so it is a natural paradigm when the matrix-by-vector multiplication is performed by a differentiated routine. The typical flow chart for Reverse Communication is given in Figure 3.3 where we note that all the phases of a typical iterative solver (matrix-by-vector multiplication, preconditioning, etc.) are kept separated, and the subroutine does not need to know anything about the data structure of the problem but use only the result of the various phases to drive the algorithm.

To solve the two case of (3.7) we have implemented two versions of the same GMRES-RCI routine with only a difference regarding the way to compute the matrix-by-vector multiplication Ax:

- the first version (that we have called `matrixfree_solve_linearsystem`) performs the matrix-by-vector multiplications using the routine (3.5) and therefore solves the linear system $\left(\frac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)\xi = b$;
- the second version (that we have called `matrixfree_solve_adjointlinearsystem`) performs the matrix-by-vector multiplications using the routine (3.6) and therefore solves the adjoint linear system $\left(\frac{\partial f}{\partial x}\big|_{(x_0,y_0)}\right)^*\xi = b$.

However, for the "matrix-free methods" (and for the iterative methods in general) it is a mandatory task to use an appropriate *preconditioner*, in order to accelerate the velocity of convergence to the solution of the system [Saad, 1996]. The problem here is that to build a preconditioner usually we need to know the matrix A and not only its effect on a vector. This request seems to be in contrast with our purpose of building a matrix-free linear solver, but it is not always the case. As we have seen in the Section 1.2.5, many CFD solvers use an implicit Newton-like algorithm to converge to the steady solution W_h, and this algorithm takes the form of a defect-correction iteration [Barrett et al., 1988; Koren, 1988; Skeel, 1981; Stetter, 1978]

$$\begin{cases} \left[\dfrac{1}{\Delta t^n} + \left(\dfrac{\partial \Psi^{(1)}}{\partial W}\bigg|_{(\gamma,W^n)}\right)\right]\delta W^n = -\Psi^{(2)}(\gamma, W^n) \\ W^{n+1} = W^n + \delta W^n \end{cases} \tag{3.9}$$

in which the Jacobian matrix $A^{(1)} = \left(\frac{\partial \Psi^{(1)}}{\partial W}\big|_{(\gamma,W^n)}\right)$ is referred to the first-order approximation of the state function $\Psi^{(1)}(\gamma, W)$ and is obtained in some way (by exact hand-code differentiation, for example) and stored in the memory. Moreover, due to the fact that a preconditioner is only an approximate inverse of the original matrix that represents the linear system to solve, we can start from the first-order accurate Jacobian matrix $A^{(1)}$, build a preconditioner $\left(P^{(1)}\right)^{-1}$ and then use this preconditioner to solve the second-order accurate linear system

$$\left(\frac{\partial \Psi^{(2)}}{\partial W}\bigg|_{(\gamma,W_h)}\right)\xi = b \tag{3.10}$$

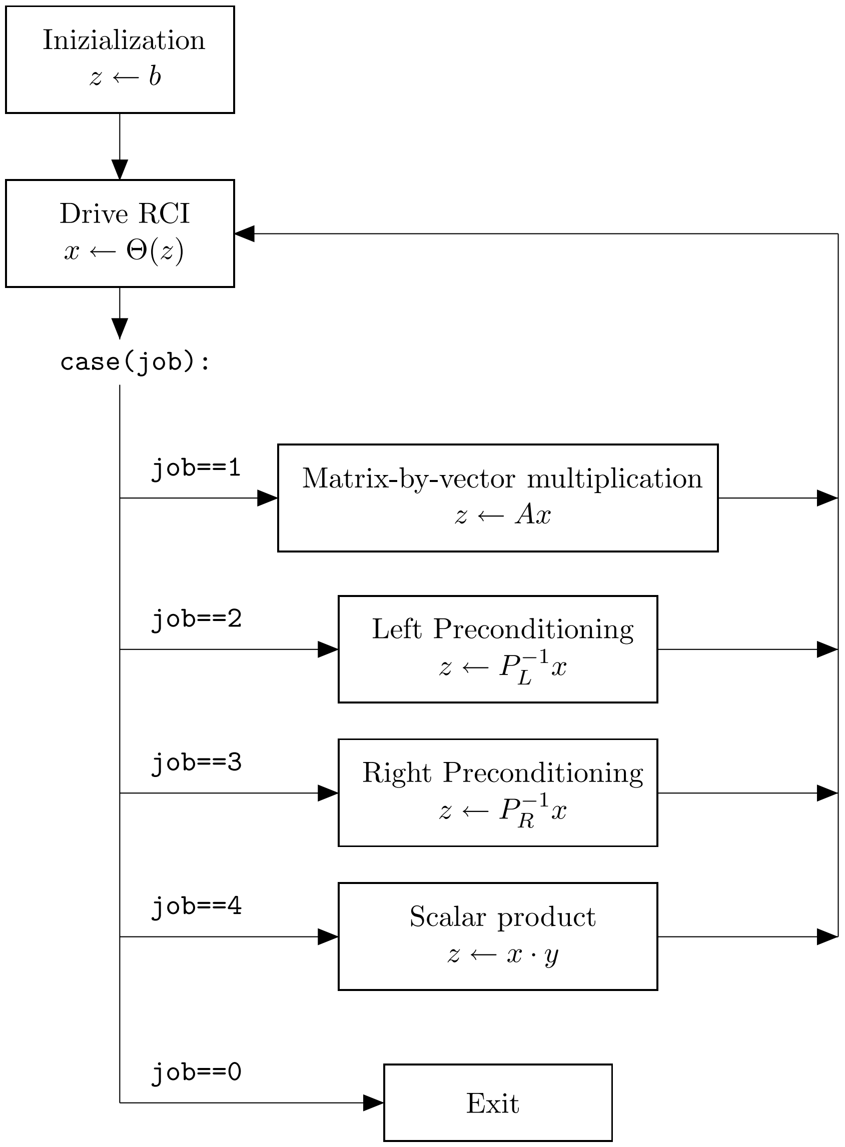


Figure 3.3: The Reverse Communication Interface. The block Drive RCI is the driver for the algorithm and contains all the operations that define the iterative solver (GMRES, etc.) and the test to verify the stopping criteria. P_L and P_R are the left and right-preconditioner respectively and are input arguments for the routine.

with the matrix-free algorithm above, in which the differentiated routine are referred to the second-order approximation. In the case of the adjoint linear system

$$\left(\frac{\partial \Psi^{(2)}}{\partial W}\bigg|_{(\gamma, W_h)} \right)^* \xi = c \tag{3.11}$$

we need to use the preconditioner $\left(P^{(1)*}\right)^{-1}$ built from the transposed matrix $A^{(1)*}$.

Obviously, we expect that the number of iteration needed to solve one of the systems in (3.10)–(3.11) with such first-order preconditioners will be greater that the corrispondent first-order linear system. We have made some numerical tests on with different preconditioners and different dimensions for the Krylov space and the results confirmed our hypothesis (see the Section 6.1 for details).

3.4 Automatic Differentiation of constrained functionals

We are interested in obtaining the first and second derivatives of a functional j depending on the control $\gamma \in \mathbb{R}^n$, and expressed in terms of a state $W(\gamma) \in \mathbb{R}^N$ as follows:

$$j\colon \gamma \mapsto j(\gamma) := J(\gamma, W) \quad \text{with } W\colon \gamma \mapsto W(\gamma) \text{ such that } \Psi(\gamma, W) = 0 \tag{3.12}$$

and where $J\colon \mathbb{R}^n \times \mathbb{R}^N \to \mathbb{R}$.

We observe that $W\colon \gamma \to W(\gamma)$ is a function *implicitly* defined through the state equation $\Psi(\gamma, W(\gamma)) = 0$, and the functional $j(\gamma) := J(\gamma, W(\gamma))$ is evaluated at the solution $W(\gamma)$ for the state equation. In general, the solution for the state equation can be found only numerically, i.e. we have a program that takes a value γ_h for the control and gives us the corresponding state W_h that satisfies the constraint $\Psi(\gamma_h, W_h) = 0$. Therefore our problem is how to compute the gradient $\frac{dj}{d\gamma}\big|_{\gamma_h}$ and the Hessian $\frac{d^2 j}{d\gamma^2}\big|_{\gamma_h}$ at the point γ_h.

We can consider two different points of view:

- *Implicit (brute-force) differentiation:* it consists in differentiating directly the *implicit* function j as a function of the control variable γ. This means that the entire process, involving the solution algorithm for state equation and the evaluation of the functional, is considered to be implemented by the single program `func_implicit(j,gamma)` and it is differentiated as a whole (Fig. 3.4).

- *Differentiation of explicit parts:* the second point of view is to consider the solution algorithm for state equation and the functional evaluation as separated processes, and applying differentiation only to the routines which compute *explicit* functions (that are functions implementing the state residual Ψ and the functional J). A typical structure for a program that performs the evaluation of a functional where the state equation $\Psi(\gamma, W) = 0$ is solved with a fixed-point algorithm is given in the Fig. 3.5, in which the subroutine implementing the state residual is called `state_residuals(psi,gamma,w)` and the routine implementing the evaluation of the functional is called `functional(j,gamma,w)`.

The underlying idea is that an explicit function is implemented by a sequence of arithmetic computations, whereas implicit functions are implemented using solvers and other iterative algorithms. Explicit functions basically have a fixed computational graph therefore the underlying function can be considered continuous and differentiable and the AD theory is well-founded in this case. The computational graph of implicit functions is dynamic (this is the case where we have branches or controls depending on active variables), where a small change of the differentiable input may change completely this computational graph: the function is then only piecewise-continuous and in correspondence to these discontinuity points we are getting out of the framework for which the AD is fully justified, resulting in the possibility of wrong results. AD should be used with extreme care for programs having dynamic computational graph.

In the first approach, differentiating the entire program implementing j can be performed with either Tangent or Reverse mode (see Section 3.2). It directly produces a differentiated program, in a black box manner. The risk is that this program is sometimes not reliable and it often exhibits very poor performance.

To analyze this last issue, we observe that since the program implementing j contains the iterative solver method for the state equation, the differentiated program will contain this solver in differentiated form. Let's assume that we need n_{iter} iterations to obtain the nonlinear solution, and that each iteration costs $(1 + c)$, where we assume an unit runtime cost for the evaluation of the residual $\Psi(\gamma, W)$ and a cost $c > 0$ for the remaining part of the iteration (that contains the algorithm for updating the solution to the next step): the total cost is then $n_{\text{iter}}(1 + c)$.

Tangent mode produces a program that we need to apply n times for computing the entire gradient. The cost is $n(n_{\text{iter}}\alpha_T)(1+c)$ where α_T is the overhead associated with the differentiated code with respect to the original one. One has usually $1 < \alpha_T < 4$, see for example [Griewank, 2000]. Further, the memory requirements will be about twice the memory needed by the original code.

With Reverse mode, we are able to obtain the entire gradient with a single evaluation of the differentiated routine. But, as we have seen in Section 3.2, the Reverse mode with the Store-All (SA) strategy produces a code which involves two successive parts [Hascoët and Pascual, 2004]:

- a *forward sweep* close to original code,
- a *backward sweep* performed in the reverse order of the original code.

The problem is that the *backward sweep* needs data computed in the *forward sweep*, but in the reverse order. In the SA strategy, these data are stored in a stack during the *forward sweep* (using a `PUSH` function) and taken from the stack during the *backward sweep* (using a `POP` function).

The total cost (in terms of CPU time and memory) strongly depends on the strategy applied by the AD tool to solve the problem of making the intermediate values available in reverse order (see Section 3.2). For the case of SA strategy, the CPU cost to evaluate the gradient will be $(n_{\text{iter}}\alpha_R)(1 + c)$ with $1 < \alpha_R < 5$, i.e. α_R times the original code, but there is an additional cost in memory to store values on the stack. This stack size is proportional to n_{iter} and it can quickly exceed the available memory. For a Recompute-All (RA) strategy the memory will be of the same order as the original routine, but the CPU cost will be $(n_{\text{iter}}^2\alpha_R)(1 + c)$, i.e. $(n_{\text{iter}}\alpha_R)$ times the nonlinear solution.

For real large programs, neither SA or RA strategy can usually work (SA requires too much memory and RA requires too much runtime), so we need a special storage/recomputation tradeoff in order to be efficient using *checkpoints* (see [Hascoët and Pascual, 2004]). The idea is to store enough variables (*snapshots*) to be able to restart execution of the backward sweep from a given point, in order to reduce the stack size for the SA strategy or the lenght of recomputation sequence for the RA strategy.

Obviously, the runtime cost of SA strategy with checkpointing will be greater than the pure SA strategy (with the benefit of a smaller stack). However in many cases we can keep this cost reasonably low. For example in the case of iterative processes of fixed lenght n_{iter}, it has been shown [Griewank, 1992] that the runtime cost of the differentiated code is of the order of $\sqrt[s]{n_{\text{iter}}}(n_{\text{iter}}\alpha_R)(1+c)$ and the stack size grows as $\sqrt[s]{n_{\text{iter}}}$ (where s is the number of snapshots available). To the opposite, RA with checkpointing results in a lower runtime (and higher memory requirements) with respect the pure RA.

In many cases, these checkpointing strategies are the only way to go. This is the case for unsteady nonlinear systems, where we have not a steady solution but a time-dependent solution which depends on the initial conditions, and for which we do not know a strategy working without the intermediate-time state variable values. Checkpointing can be applied quasi-automatically by the Automatic Differentiator or applied by hand-coding (see [Tber et al., 2007] for an application of checkpointing to an Oceanographic code).

In contrast to the brute force approach, we consider the case where the iterative algorithm is a fixed point one, e.g. when we have stationary problems (see the Fig. 3.5). In this case we can avoid the differentiation of the iterative algorithms (implemented in the routine `flow_solver(gamma,w)` in Fig. 3.5) that could come from the pseudo-time advancing scheme, but we differentiate only the routines implementing the state residuals computation and the functional evaluation (that we assume not containing any iterative algorithm), and we will use only the final state W_h. This strategy results in a differentiated code that is faster and that does not suffer from reliability problems. Moreover, in the context of the fixed-point algorithms the solution does not depend on the initial guess W^0: therefore in the Reverse mode only the final state variable W_h is necessary for the *backward sweep*, resulting in a smaller stack size and therefore in a lower memory requirement.

For the above reasons, in case of fixed-point algorithms, we recommend the differentiation of explicit parts instead of the implicit (brute-force) approach. Moreover, as we will see in the following sections, with an appropriate design of the interfaces of the routine implementing the state residual Ψ and the functional J, we can provide a framework that frees the user from the complex task of organizing the algorithms needed for the gradient and Hessian evaluation.

We present now in more details this strategy: we go back to the mathematical equations of the constrained functional, then manipulate them to obtain equations for the required derivatives, and from there we deduce our architecture of the differentiated code that avoids the problems above.

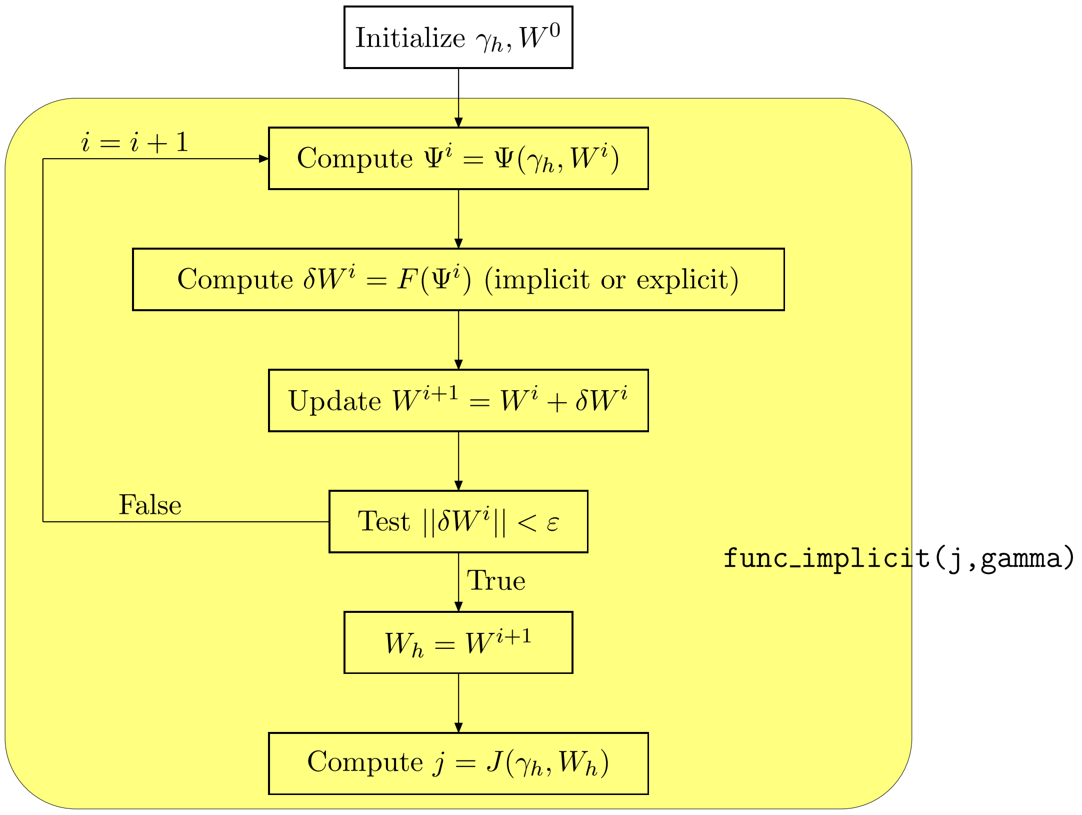

Initialize γ_h, W^0
$i = i + 1$
Compute $\Psi^i = \Psi(\gamma_h, W^i)$
Compute $\delta W^i = F(\Psi^i)$ (implicit or explicit)
Update $W^{i+1} = W^i + \delta W^i$
False
Test $||\delta W^i|| < \varepsilon$
True
$W_h = W^{i+1}$
Compute $j = J(\gamma_h, W_h)$
func_implicit(j,gamma)

Figure 3.4: Typical structure for a program that performs the evaluation of a functional with a fixed-point algorithm. In this case the entire process (solution algorithm for the state equation and the evaluation of the functional) is considered to be implemented by the single program `func_implicit(j,gamma)`: the functional is considered as an implicit function only of the control variable γ, i.e. a function containing iterative algorithms. The application of Automatic Differentiation on this kind of program could give incorrect results, due to the presence of the iterative algorithms for which the AD framework is not fully justified.

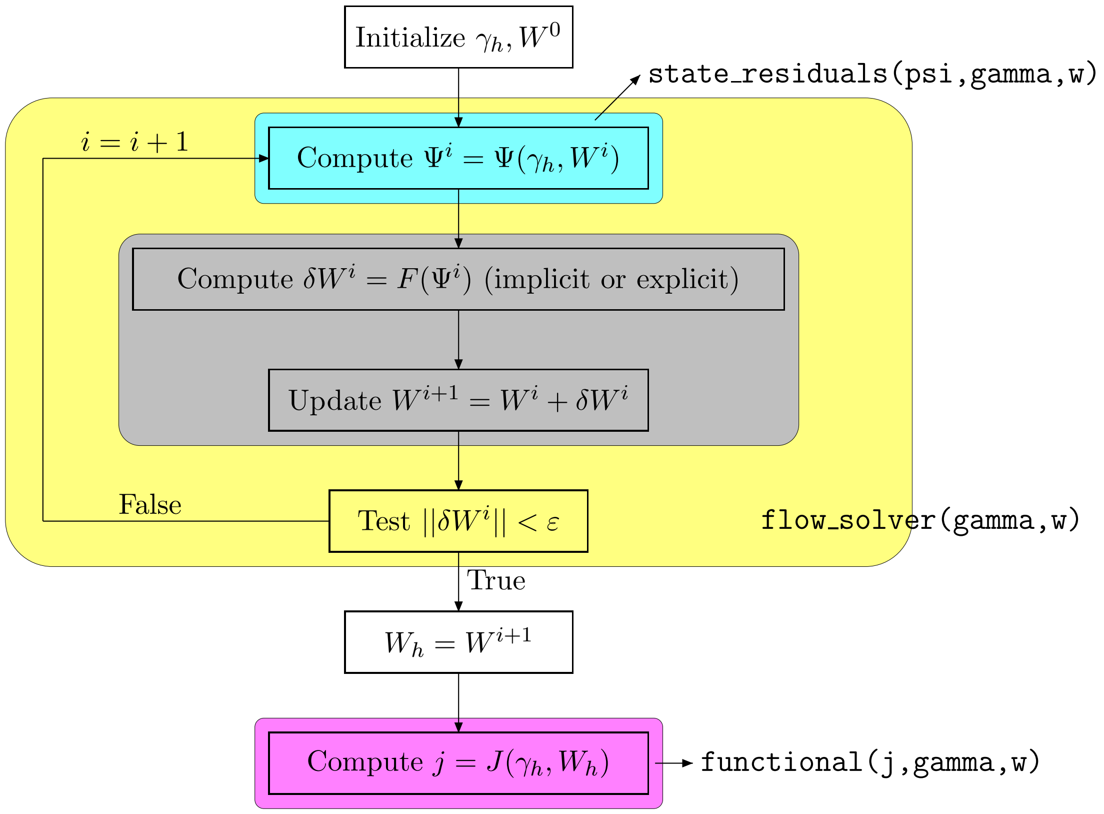

Initialize γ_h, W^0
state_residuals(psi,gamma,w)
$i = i + 1$
Compute $\Psi^i = \Psi(\gamma_h, W^i)$
Compute $\delta W^i = F(\Psi^i)$ (implicit or explicit)
Update $W^{i+1} = W^i + \delta W^i$
False
Test $||\delta W^i|| < \varepsilon$
flow_solver(gamma,w)
True
$W_h = W^{i+1}$
Compute $j = J(\gamma_h, W_h)$
functional(j,gamma,w)

Figure 3.5: Typical structure for a program that performs the evaluation of a functional with a fixed-point algorithm. The iterative flow-solver can be viewed as the composition of two phases: the evaluation of the residual (performed by the subroutine `state_residuals(psi,gamma,w)`) that does not contain any iterative algorithm, and the evolution (with implicit or explicit methods) to the next step. In our strategy we want to avoid the differentiation of iterative algorithms (for which the application of AD is not fully justified), therefore we differentiate only the state residual and the routine `functional(j,gamma,w)` that performs the evaluation of the functional.

3.5 First-order derivative

As we have seen in the previous section, to compute the gradient of a function using AD, we can choose between two modes: Tangent and Reverse mode differentiation. Now we want compute the gradient j' of the constrained functional (3.12) using the differentiation of explicit parts implementing Ψ and J with the two differentiation modes.

Using the chain rule, the differential of the functional j is given in terms of Ψ and J by

$$\frac{dj}{d\gamma} = \frac{\partial J}{\partial \gamma} + \frac{\partial J}{\partial W}\frac{dW}{d\gamma} \tag{3.13}$$

where the derivatives of the state variables $W(\gamma)$, that we remeber is an *implicit function*, could be obtained through the differentiating the state equation $\Psi(\gamma, W) = 0$, namely:

$$\frac{\partial \Psi}{\partial \gamma} + \frac{\partial \Psi}{\partial W}\frac{dW}{d\gamma} = 0 \tag{3.14}$$

and therefore

$$\frac{dW}{d\gamma} = -\left(\frac{\partial \Psi}{\partial W}\right)^{-1}\frac{\partial \Psi}{\partial \gamma} \; .$$

3.5.1 Tangent mode differentiation

It consists in computing the Gâteaux-derivatives with respect to each component direction e_i with $i = 1, \dots, n$ ($e_i = (0, \dots 0, 1, 0, \dots, 0)^T$, where 1 is at the i-th component):

$$\frac{dj}{d\gamma_i} = \frac{dj}{d\gamma}e_i = \frac{\partial J}{\partial \gamma}e_i + \frac{\partial J}{\partial W}\frac{dW}{d\gamma}e_i \tag{3.15}$$

where $\frac{dW}{d\gamma}e_i$ is the solution of the linear system:

$$\frac{\partial \Psi}{\partial W}\frac{dW}{d\gamma}e_i = -\frac{\partial \Psi}{\partial \gamma}e_i \; . \tag{3.16}$$

In order to get the gradient, (3.16) must be solved and (3.15) has to be evaluated at the point (γ_h, W_h) for each vector e_i of the canonical basis, i.e. n times and the main cost is due to the solution of n linearised N-dimensional systems.

If we choose to solve the single system (3.16) with the iterative matrix-free method presented in Section 3.3, and the solution is obtained after $n_{\text{iter},T}$ steps, the total cost will be of the order of $\alpha_T n_{\text{iter},T}$, i.e. $n_{\text{iter},T}$ evaluation of the matrix-by-vector operation $\frac{\partial \Psi}{\partial W}x$, where we assume that each evaluation costs α_T times the evaluation of the state residual $\Psi(\gamma, W)$ (and the cost of the state residual is taken as reference equal to 1). Therefore, the cost of the full gradient will be $n\alpha_T n_{\text{iter},T}$.

3.5.2 Reverse mode differentiation

If we perform the trasposition of (3.13) and using the (3.14), the complete gradient can be expressed by the equation

$$\left(\frac{dj}{d\gamma}\right)^* = \left(\frac{\partial J}{\partial\gamma}\right)^* - \left(\frac{\partial\Psi}{\partial\gamma}\right)^* \Pi \tag{3.17}$$

where $\Pi\colon (\gamma, W) \mapsto \Pi(\gamma, W)$ (the adjoint state) is the solution of the linear system

$$\left(\frac{\partial\Psi}{\partial W}\right)^* \Pi = \left(\frac{\partial J}{\partial W}\right)^* . \tag{3.18}$$

It is important to note that the above formulation permits us to obtain all the derivatives needed by (3.17)-(3.18) using Reverse mode differentiation of the programs implementing $J(\gamma, W)$ and $\Psi(\gamma, W)$.

To compute the gradient j' with this approach we need to solve only one linearised N-dimensional system: the adjoint system (3.18). If we choose to solve the adjoint system with an iterative matrix-free method, we can apply the same estimate as in the case of the Tangent mode differentiation, but this time the overhead associated with the evaluation of the matrix-by-vector operation $\left(\frac{\partial\Psi}{\partial W}\right)^* x$ with respect to the state residual evaluation will be α_R (and usually $\alpha_R > \alpha_T$) and the number of iterations $n_{\text{iter},R}$ for the convergence of the solution could be different from $n_{\text{iter},T}$ of the previous case (in our experience, the number of iterations needed by GMRES to solve the system $Ax = b$ and $A^T x = c$ are of the same order, see the Section 6.1): therefore the total runtime cost cost for the gradient will be $\alpha_R n_{\text{iter},R}$.

From the previous arguments it clearly appears that, if we need to compute the gradient j' only, the Reverse mode is cheaper in terms of CPU time respect to the Tangent mode if $n \gg 1$. Nevertheless, the Tangent mode algorithm in Section 3.5.1 will be used in the following because it is the basis for the Hessian computation with the Tangent-on-Tangent approach.

3.6 Second-order derivative

In the same manner as the computation of the gradient, for second derivatives we have different possibilities, which are theoretically equivalent, but they differ in the computational cost and the best strategy depends, in the end, on the number n of the control variables for the given functional. Moreover, the best strategy depends on which use of the second-order derivative we need, i.e. the best strategy to obtain the full Hessian matrix could be different from the best strategy to obtain its diagonal part or the multiplication of the Hessian matrix by a vector.

The first method to obtain the second-order differentiation of a constrained functional performs two successive Tangent mode differentiations for both the functional and the state residuals and use the adjoint state to compute each single element in the Hessian matrix [Ghate and Giles, 2007]: we call this approach Tangent-on-Tangent (ToT). The second approach (Tangent-on-Reverse, ToR) performs a Tangent mode differentiation over the gradient (3.17) obtained with Reverse differentiation.

3.6.1 Tangent-on-Tangent approach

This methods was initially investigated by [Sherman et al., 1996] along with various other algorithms, but the publication does not go into the implementation details for a generic fluid dynamic code. Here we present the mathematical background behind the idea and the efficient AD implementation of [Ghate and Giles, 2007] but with a different analysis of the computational cost.

Starting from the i-th element of the gradient (3.15), we perform another differentiation with respect to the variable γ_k obtaining the i-k element of the Hessian matrix

$$\left(\frac{d^2 j}{d\gamma^2}\right)_{i,k} = \frac{d^2 j}{d\gamma_i d\gamma_k} = D^2_{i,k} J + \frac{\partial J}{\partial W}\frac{d^2 W}{d\gamma_i d\gamma_k} \tag{3.19}$$

where

$$D^2_{i,k}J = \frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}e_i\right)e_k + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}e_i\right)\frac{dW}{d\gamma_k} + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}e_k\right)\frac{dW}{d\gamma_i} + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\frac{dW}{d\gamma_i}\right)\frac{dW}{d\gamma_k}\ .$$

Differentiating the equation (3.16) we get

$$D^2_{i,k}\Psi + \frac{\partial \Psi}{\partial W}\frac{d^2 W}{d\gamma_i d\gamma_k} = 0 \tag{3.20}$$

where

$$D^2_{i,k}\Psi = \frac{\partial}{\partial\gamma}\left(\frac{\partial \Psi}{\partial\gamma}e_i\right)e_k + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial\gamma}e_i\right)\frac{dW}{d\gamma_k} + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial\gamma}e_k\right)\frac{dW}{d\gamma_i} + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial W}\frac{dW}{d\gamma_i}\right)\frac{dW}{d\gamma_k}$$

and e_i (e_k) is the usual vector of the canonical basis with 1 at the i-th (k-th) component and zero otherwise. Substituting the second derivatives of the state with respect to the control variables $\frac{d^2W}{d\gamma_i d\gamma_k}$ in equation (3.19) from equation (3.20) we get

$$\begin{aligned}\frac{d^2 j}{d\gamma_i d\gamma_k} &= D^2_{i,k}J - \frac{\partial J}{\partial W}\left(\frac{\partial \Psi}{\partial W}\right)^{-1} D^2_{i,k}\Psi \\ &= D^2_{i,k}J - \Pi^* D^2_{i,k}\Psi\end{aligned} \tag{3.21}$$

where Π is the solution of the adjoint system (3.18). The i-k element of the Hessian matrix $\left(\frac{d^2 j}{d\gamma^2}\big|_{\gamma_h}\right)$ is then obtained evaluating the (3.21) at the point (γ_h, W_h) solution of the state equation $\Psi = 0$, namely

$$\left(\frac{d^2 j}{d\gamma^2}\bigg|_{\gamma_h}\right)_{i,k} = \left(D^2_{i,k}J\right)\big|_{(\gamma_h,W_h)} - \Pi_h^*\left(D^2_{i,k}\Psi\right)\big|_{(\gamma_h,W_h)} \tag{3.22}$$

where $\Pi_h \in \mathbb{R}^N$ is the solution of the adjoint linear system

$$\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^* \Pi_h = \left(\frac{\partial J}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^*\ .$$

The n derivatives $\dfrac{dW}{d\gamma_i}$ in the formulas for $D^2_{i,k}J$ and $D^2_{i,k}\Psi$, should be computed (and stored) solving the linear systems

$$\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)\frac{dW}{d\gamma_i} = -\left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma_h,W_h)}\right)e_i \qquad i = 1,\ldots,n$$

and this task can be performed using the routine `matrixfree_solver_linearsystem` as in Section 3.3. We assume the number of iterations needed for the iterative linear solver to the convergence of the solution to be $n_{\text{iter},T}$, and each iteration calls a tangent-differentiated routine (namely `state_residuals_dw_d`) implementing the matrix-by-vector multiplication $\frac{\partial\Psi}{\partial W}x$ whose cost is α_T times the cost of the original routine implementing the evaluation of the state residuals $\Psi(\gamma, W)$.

Implementation. Now the question is: how can we obtain the quantities $\left(D^2_{i,k}\Psi\right)\big|_{(\gamma_h,W_h)}$ and $\left(D^2_{i,k}J\right)\big|_{(\gamma_h,W_h)}$ in (3.22) using Automatic Differentiation? As we will see soon, if we perform two successive Tangent-mode differentiation of the routine implementing Ψ (J) we will able to compute $\left(D^2_{i,k}\Psi\right)\big|_{(\gamma_h,W_h)}$ (or the same quantity relative to J) with a single invocation of the resulting double-differentiated routine. Let us suppose that the subroutine computing the state residual $\Psi(\gamma, W)$ is `state_residuals(psi,gamma,w)`, where the input variables are `gamma` and `w`, and the output variable is `psi`.

$$\texttt{state\_residuals}(\underset{\downarrow}{\texttt{psi}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{w}})$$

Automatic Differentiation in Tangent mode with respect to the input variables `gamma` and `w` builds subroutine:

$$\texttt{state\_residuals\_d}(\underset{\downarrow}{\texttt{psi}}, \underset{\downarrow}{\texttt{psid}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{gammad}}, \overset{\downarrow}{\texttt{w}}, \overset{\downarrow}{\texttt{wd}})$$

that has the additional output $\texttt{psid} = \dot{\Psi} = \left(\frac{\partial\Psi}{\partial\gamma}\right)\dot{\gamma} + \left(\frac{\partial\Psi}{\partial W}\right)\dot{W}$, calling $\texttt{gammad} = \dot{\gamma}$ and $\texttt{wd} = \dot{W}$ the additional input variables.

Now we differentiate the routine `state_residuals_d` in tangent mode considering `psid` as the output variable and with respect to `gamma` and `w`, obtaining

$$\texttt{state\_residuals\_d\_d}(\underset{\downarrow}{\texttt{psi}}, \underset{\downarrow}{\texttt{psid}}, \underset{\downarrow}{\texttt{psidd}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{gammad0}}, \overset{\downarrow}{\texttt{gammad}}, \overset{\downarrow}{\texttt{w}}, \overset{\downarrow}{\texttt{wd0}}, \overset{\downarrow}{\texttt{wd}}) \tag{3.23}$$

the additional output of which is

$$\texttt{psidd} = \dot{\dot{\Psi}} = \frac{\partial}{\partial\gamma}\left(\frac{\partial\Psi}{\partial\gamma}\dot{\gamma}\right)\dot{\gamma_0} + \frac{\partial}{\partial W}\left(\frac{\partial\Psi}{\partial\gamma}\dot{\gamma}\right)\dot{W_0} + \frac{\partial}{\partial W}\left(\frac{\partial\Psi}{\partial\gamma}\dot{\gamma_0}\right)\dot{W} + \frac{\partial}{\partial W}\left(\frac{\partial\Psi}{\partial W}\dot{W}\right)\dot{W_0}$$

and where $\texttt{gammad0} = \dot{\gamma_0}$ and $\texttt{wd0} = \dot{W_0}$ are additional input variables.

In order to evaluate the term $D^2_{i,k}\Psi$ at the point (γ_h, W_h) we must call the routine (3.23) with the appropriate arguments, that is:

$$\texttt{state\_residuals\_d\_d}(\underset{\Psi}{\texttt{psi}}, \underset{\dot{\Psi}}{\texttt{psid}}, \underset{\dot{\dot{\Psi}}}{\texttt{psidd}}, \overset{\gamma_h}{\texttt{gamma}}, \overset{e_k}{\texttt{gammad0}}, \overset{e_i}{\texttt{gammad}}, \overset{W_h}{\texttt{w}}, \overset{\frac{dW}{d\gamma_k}}{\texttt{wd0}}, \overset{\frac{dW}{d\gamma_i}}{\texttt{wd}}) \tag{3.24}$$

where the derivative of the state variables with respect to the control $\frac{dW}{d\gamma_i}$ is obtained as solution of the linear system (3.16) and the resulting output variable is

$$\texttt{psidd} = \dot{\dot{\Psi}} = D^2_{i,k}\Psi\big|_{(\gamma_h, W_h)} .$$

Therefore, the Tangent-on-Tangent approach is the application of two successive tangent-mode differentiations: the first differentiation acts on the original routine, the second one acts on the result of the first differentiation.

The same argument applies to the evaluation of the term $D^2_{i,k}J$. In this case, we perform a Tangent-on-Tangent derivative of the routine

$$\texttt{functional}(\underset{\downarrow}{\texttt{j}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{w}})$$

and we get

$$\texttt{functional\_d\_d}(\underset{J}{\texttt{j}}, \underset{\dot{J}}{\texttt{jd}}, \underset{\dot{\dot{J}}}{\texttt{jdd}}, \overset{\gamma_h}{\texttt{gamma}}, \overset{e_k}{\texttt{gammad0}}, \overset{e_i}{\texttt{gammad}}, \overset{W_h}{\texttt{w}}, \overset{\frac{dW}{d\gamma_k}}{\texttt{wd0}}, \overset{\frac{dW}{d\gamma_i}}{\texttt{wd}}) \tag{3.25}$$

where the resulting $\texttt{jdd} = \dot{\dot{J}}$ is the value of $\left(D^2_{i,k}J\right)\big|_{(\gamma_h, W_h)}$.

It is useful to note that the n derivatives of the state with respect to the control $\frac{dW}{d\gamma_i}$ must be evaluated and stored *before* any evaluation of $D^2_{i,k}J$ or $D^2_{i,k}\Psi$. If the number of state variables N and/or the number of control variables n are high, the previous strategy could be not applicable. One possible solution for this problem could be to store the vectors $\frac{dW}{d\gamma_i}$ on the hard-disk instead of keeping them into the RAM, but this strategy could have negative impact on the performance of the computation due to the I/O overhead.

Description of the algorithm for the Hessian matrix with the ToT approach. The algorithm to compute the Hessian matrix with the ToT approach can be summarized as follow:

1. compute the state W_h such that $\Psi(\gamma_h, W_h) = 0$;
2. compute $\bar{W}_J = \left(\left.\dfrac{\partial J}{\partial W}\right|_{(\gamma_h, W_h)}\right)^*$
3. compute the adjoint state Π_h solving the linear system $\left(\left.\dfrac{\partial \Psi}{\partial W}\right|_{(\gamma_h, W_h)}\right)^* \Pi_h = \bar{W}_J$;
4. for each element of the canonical basis e_i, $i = 1, \ldots, n$

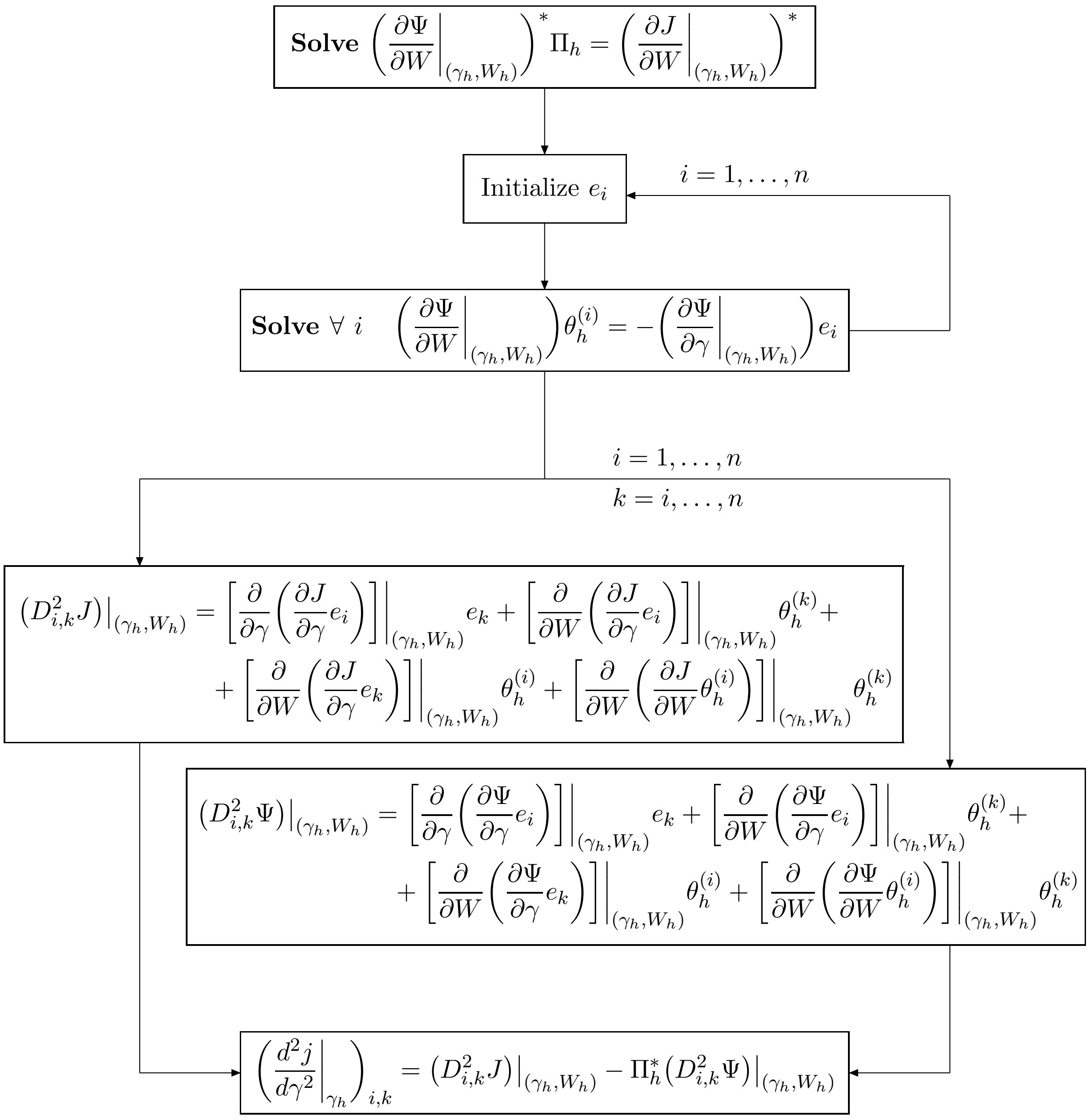

Solve $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h, W_h)}\right)^* \Pi_h = \left(\frac{\partial J}{\partial W}\Big|_{(\gamma_h, W_h)}\right)^*$
Initialize e_i
$i = 1, \dots, n$
Solve $\forall\ i$ $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h, W_h)}\right)\theta_h^{(i)} = -\left(\frac{\partial \Psi}{\partial \gamma}\Big|_{(\gamma_h, W_h)}\right)e_i$
$i = 1, \dots, n$
$k = i, \dots, n$
$(D^2_{i,k} J)\big|_{(\gamma_h, W_h)} = \left[\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial \gamma} e_i\right)\right]\Big|_{(\gamma_h, W_h)} e_k + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma} e_i\right)\right]\Big|_{(\gamma_h, W_h)} \theta_h^{(k)} + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma} e_k\right)\right]\Big|_{(\gamma_h, W_h)} \theta_h^{(i)} + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W} \theta_h^{(i)}\right)\right]\Big|_{(\gamma_h, W_h)} \theta_h^{(k)}$
$(D^2_{i,k} \Psi)\big|_{(\gamma_h, W_h)} = \left[\frac{\partial}{\partial \gamma}\left(\frac{\partial \Psi}{\partial \gamma} e_i\right)\right]\Big|_{(\gamma_h, W_h)} e_k + \left[\frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial \gamma} e_i\right)\right]\Big|_{(\gamma_h, W_h)} \theta_h^{(k)} + \left[\frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial \gamma} e_k\right)\right]\Big|_{(\gamma_h, W_h)} \theta_h^{(i)} + \left[\frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial W} \theta_h^{(i)}\right)\right]\Big|_{(\gamma_h, W_h)} \theta_h^{(k)}$
$\left(\frac{d^2 j}{d\gamma^2}\Big|_{\gamma_h}\right)_{i,k} = (D^2_{i,k} J)\big|_{(\gamma_h, W_h)} - \Pi_h^* (D^2_{i,k} \Psi)\big|_{(\gamma_h, W_h)}$

Figure 3.6: Tangent-on-Tangent algorithm for the full Hessian matrix.

   (a) compute $\dot{\Psi}_\gamma^{(i)} = \left(\dfrac{\partial \Psi}{\partial \gamma}\Big|_{(\gamma_h, W_h)} \right) e_i$

   (b) compute (and store) the vector $\theta_h^{(i)}$ solution of the linear system

$$\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h, W_h)} \right) \theta_h^{(i)} = -\dot{\Psi}_\gamma^{(i)} \tag{3.26}$$

5. for $i = 1, \dots, n$ and $k = i, \dots, n$

   (a) compute $\left(D_{i,k}^2 \Psi\right)\big|_{(\gamma_h, W_h)}$ using the subroutine 3.24;

   (b) compute $\left(D_{i,k}^2 J\right)\big|_{(\gamma_h, W_h)}$ using the subroutine 3.25;

   (c) compute $\left(\dfrac{d^2 j}{d\gamma^2}\Big|_{\gamma_h} \right)_{i,k} = \left(D_{i,k}^2 J\right)\big|_{(\gamma_h, W_h)} - \Pi_h^* \left(D_{i,k}^2 \Psi\right)\big|_{(\gamma_h, W_h)}$.

From equation (3.22) we see that the ToT approach gives us a single element i-k of the Hessian matrix at time, then using the symmetry property of the Hessian we can compute the full $n \times n$ matrix applying $\frac{n(n+1)}{2}$ time the steps 5a–5c in the algorithm above.

For each element i-k we need to know the vectors $\theta_h^{(i)} = \frac{dW}{d\gamma_i}$ and $\theta_h^{(k)} = \frac{dW}{d\gamma_k}$ obtained solving the linear system (3.26) whose cost is $\alpha_T n_{\text{iter,T}}$ (for simplicity we assume that the number of iterations needed to solve the linear system is independent from the right hand side: some numerical experiments are shown in Section 6.1). These linear systems could be solved using the matrix-free algorithm `matrixfree_solve_linearsystem` (see Section 3.3).

Moreover, the quantity $\left(D_{i,k}^2 \Psi\right)\big|_{(\gamma_h, W_h)}$ (step 5a) can be obtained with a single invocation of the differentiated-twice subroutine (3.24) and its cost is α_T^2 times the cost of the evaluation of the residual $\Psi(\gamma, W)$ (that it is assumed to be unitary). The analogous quantity relative to the functional $\left(D_{i,k}^2 J\right)\big|_{(\gamma_h, W_h)}$ (step 5b) can be obtained with a single invocation of the differentiated-twice subroutine (3.25) and its cost is negligible respect to (3.24), being negligible the cost to evaluate the subroutine `functional(j,gamma,w)` respect to `state_residuals(psi,gamma,w)`.

Therefore, assuming the adjoint state Π_h to be available, the evaluation of the full Hessian with the ToT approach costs

$$n\alpha_T \left[n_{\text{iter},T} + \frac{(n+1)}{2} \alpha_T \right]$$

and we note that the cost is quadratic respect to the dimension of the control variables but, if the we have $n_{\text{iter},T} \gg n$ the main contribution could be from the cost to solve the n linear systems (3.26). Therefore, if the dimension of the control variables n is small, the cost is dominated by the solution of the linear systems, otherwise (and assuming the number of iterations n_{iter} to be independent from n) the main cost is due to the differentiated-twice subroutines.

Another important thing to note is the fact that with ToT we can compute the diagonal of the Hessian without computing the extra-diagonal values, due to the fact that the Hessian is built element-by-element: this fact results in a cost for the entire diagonal of

$$n\alpha_T \left[n_{\text{iter},T} + \alpha_T \right]$$

(i.e. one linear system (3.26) and one evaluation of the differentiated-twice routines in the steps 5a-5c for each element of the diagonal).

Remark 3.3. If we want to evaluate the multiplication of the Hessian by a vector $\delta \in \mathbb{R}^n$, we can evaluate the resulting vector element-by-element, using the Tangent differentiation of derivative (3.15) along the direction δ instead of e_k. This results in the algorithm in Figure 3.7 where we have the single loop over $i = 1, \ldots, n$ and where the derivative of the state $\frac{dW}{d\gamma_k} = \frac{dW}{d\gamma} e_k$ is substituted with $\frac{dW}{d\gamma}\delta$. This last quantity can be obtained from the computed $\frac{dW}{d\gamma_k}$ using the fact that the vector δ can be considered as linear combination of vector e_k of the canonical basis. Thus, the resulting cost for the Hessian-by-vector evaluation is

$$n\alpha_T \left[n_{\text{iter},T} + \alpha_T \right] .$$

3.6.2 Tangent-on-Reverse approach

This consists in the direct derivation in any direction e_i, $i = 1, \ldots, n$ of the (non-scalar) function:

$$\left(\frac{dj}{d\gamma}\right)^* = \left(\frac{\partial J}{\partial \gamma}\right)^* - \left(\frac{\partial \Psi}{\partial \gamma}\right)^* \Pi$$

where $W\colon \gamma \mapsto W(\gamma)$ such that $\Psi(\gamma, W) = 0$ and $\Pi\colon (\gamma, W) \mapsto \Pi(\gamma, W)$ is the adjoint state defined as

$$\Pi = \left(\frac{\partial \Psi}{\partial W}\right)^{-*} \left(\frac{\partial J}{\partial W}\right)^* .$$

To build the algorithm to compute the Hessian in the present context we need the following

Lemma 3.1 (Hessian-by-vector). *Let $\gamma_h \in \mathbb{R}^n$ and $W_h \in \mathbb{R}^N$ such that $\Psi(\gamma_h, W_h) = 0$ and let*

$$\begin{array}{rcl} j\colon \mathbb{R}^n & \longrightarrow & \mathbb{R} \\ \gamma & \longmapsto & j(\gamma) := J(\gamma, W) \end{array}$$

then the projection of the Hessian $\left(\left.\frac{d^2 j}{d\gamma^2}\right|_{\gamma_h}\right) \in \mathbb{R}^{n \times n}$ along a direction $\delta \in \mathbb{R}^n$ is given by

$$\boxed{\begin{aligned} \left(\left.\frac{d^2 j}{d\gamma^2}\right|_{\gamma_h}\right)\delta = {} & \left.\left[\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial \gamma}\right)^*\right]\right|_{(\gamma_h, W_h)} \delta + \left.\left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma}\right)^*\right]\right|_{(\gamma_h, W_h)} \theta_h + \\ & - \left.\left[\frac{\partial}{\partial \gamma}\left(\left(\frac{\partial \Psi}{\partial \gamma}\right)^* \Pi_h\right)\right]\right|_{(\gamma_h, W_h)} \delta - \left.\left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial \gamma}\right)^* \Pi_h\right)\right]\right|_{(\gamma_h, W_h)} \theta_h + \\ & - \left(\left.\frac{\partial \Psi}{\partial \gamma}\right|_{(\gamma_h, W_h)}\right)^* \lambda_h \end{aligned}}$$

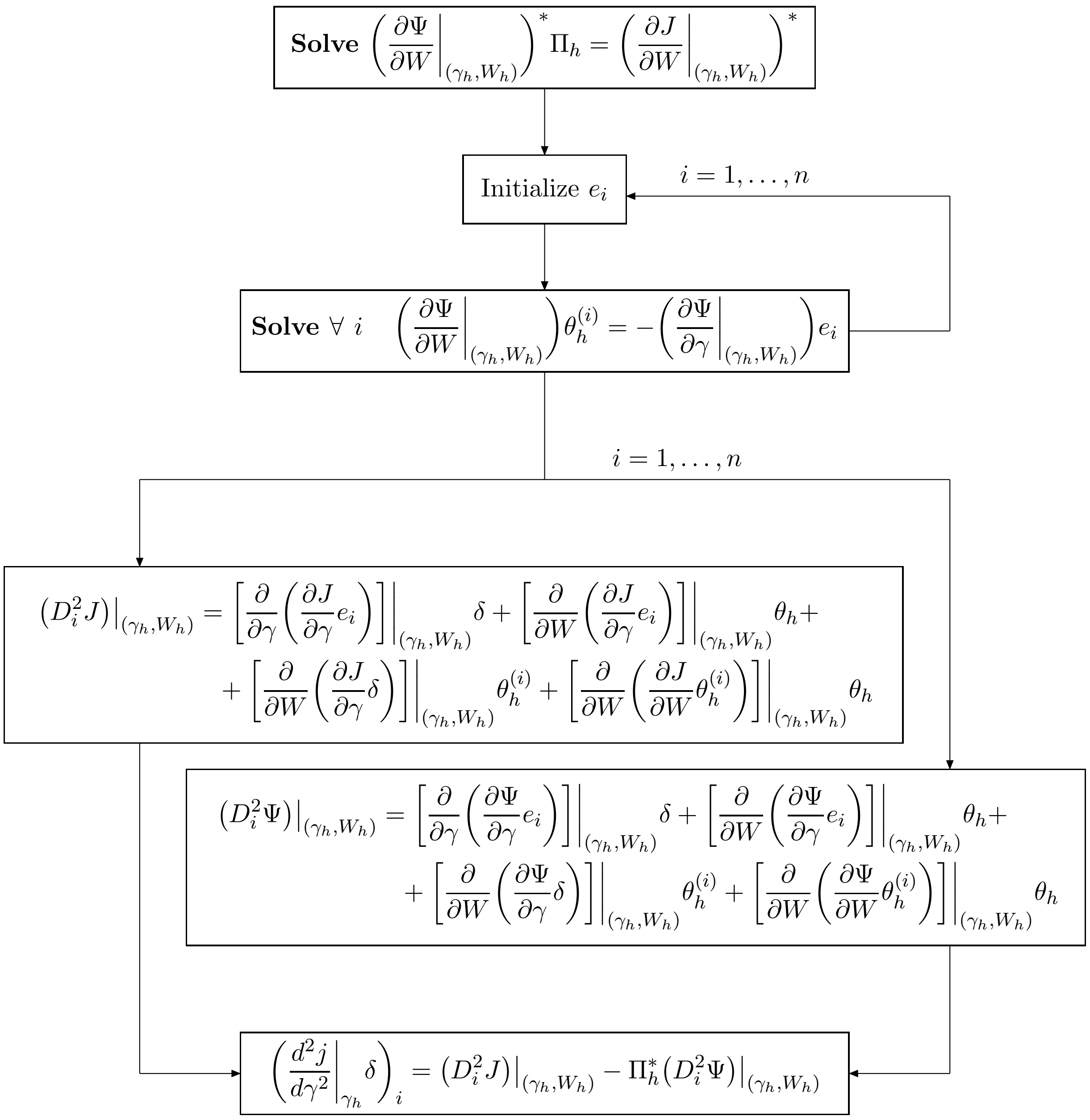

Solve $\left(\frac{\partial\Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^{*}\Pi_h = \left(\frac{\partial J}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^{*}$
Initialize e_i
$i = 1,\dots,n$
Solve $\forall\ i$ $\left(\frac{\partial\Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)\theta_h^{(i)} = -\left(\frac{\partial\Psi}{\partial\gamma}\Big|_{(\gamma_h,W_h)}\right)e_i$
$i = 1,\dots,n$
$(D_i^2 J)\big|_{(\gamma_h,W_h)} = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}e_i\right)\right]\Big|_{(\gamma_h,W_h)}\delta + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}e_i\right)\right]\Big|_{(\gamma_h,W_h)}\theta_h + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}\delta\right)\right]\Big|_{(\gamma_h,W_h)}\theta_h^{(i)} + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\theta_h^{(i)}\right)\right]\Big|_{(\gamma_h,W_h)}\theta_h$
$(D_i^2 \Psi)\big|_{(\gamma_h,W_h)} = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial \Psi}{\partial\gamma}e_i\right)\right]\Big|_{(\gamma_h,W_h)}\delta + \left[\frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial\gamma}e_i\right)\right]\Big|_{(\gamma_h,W_h)}\theta_h + \left[\frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial\gamma}\delta\right)\right]\Big|_{(\gamma_h,W_h)}\theta_h^{(i)} + \left[\frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial W}\theta_h^{(i)}\right)\right]\Big|_{(\gamma_h,W_h)}\theta_h$
$\left(\frac{d^2 j}{d\gamma^2}\Big|_{\gamma_h}\delta\right)_i = (D_i^2 J)\big|_{(\gamma_h,W_h)} - \Pi_h^{*}(D_i^2\Psi)\big|_{(\gamma_h,W_h)}$

Figure 3.7: Tangent-on-Tangent algorithm for the Hessian-by-vector multiplication. The vector θ_h is the solution of the linear system $\left(\frac{\partial\Psi}{\partial W}\right)\theta_h = -\left(\frac{\partial\Psi}{\partial\gamma}\right)\delta$ and it could be obtained as linear combination of the vector $\theta_h^{(i)}$, namely $\theta_h = \sum_i^n \delta_i\theta_h^{(i)}$ where δ_i is the i-th component of the vector δ.

where $\Pi_h, \theta_h, \lambda_h \in \mathbb{R}^N$ *satisfy*

$$\begin{cases} \left(\left.\dfrac{\partial \Psi}{\partial W}\right|_{(\gamma_h, W_h)}\right)^* \Pi_h = \left(\left.\dfrac{\partial J}{\partial W}\right|_{(\gamma_h, W_h)}\right)^* \\ \left(\left.\dfrac{\partial \Psi}{\partial W}\right|_{(\gamma_h, W_h)}\right)\theta_h = -\left(\left.\dfrac{\partial \Psi}{\partial \gamma}\right|_{(\gamma_h, W_h)}\right)\delta \\ \left(\left.\dfrac{\partial \Psi}{\partial W}\right|_{(\gamma_h, W_h)}\right)^* \lambda_h = \dfrac{\partial}{\partial \gamma}\left(\dfrac{\partial J}{\partial W}\right)^* \delta + \dfrac{\partial}{\partial W}\left(\dfrac{\partial J}{\partial W}\right)^* \theta_h + \\ \qquad\qquad - \dfrac{\partial}{\partial \gamma}\left[\left(\dfrac{\partial \Psi}{\partial W}\right)^* \Pi_h\right]\delta - \dfrac{\partial}{\partial W}\left[\left(\dfrac{\partial \Psi}{\partial W}\right)^* \Pi_h\right]\theta_h \ . \end{cases}$$

Proof. First of all, we note that the tangent derivative along the direction δ of a (n-dimensional) function $j\colon \gamma \mapsto j(\gamma) := J(\gamma, W)$ subject to $\Psi(\gamma, W) = 0$, is given by

$$\begin{aligned} \frac{dj}{d\gamma}\delta &= \frac{\partial J}{\partial \gamma}\delta + \frac{\partial J}{\partial W}\frac{dW}{d\gamma}\delta \\ &= \frac{\partial J}{\partial \gamma}\delta - \frac{\partial J}{\partial W}\left(\frac{\partial \Psi}{\partial W}\right)^{-1}\frac{\partial \Psi}{\partial \gamma}\delta \\ &= \frac{\partial J}{\partial \gamma}\delta + \frac{\partial J}{\partial W}\theta \end{aligned}$$

where $\theta\colon (\gamma, W) \mapsto \theta(\gamma, W)$ is the solution of the linear system

$$\frac{\partial \Psi}{\partial W}\theta = -\frac{\partial \Psi}{\partial \gamma}\delta \ .$$

Now we can perform the tangent derivative (along the direction δ) of $\left(\frac{dj}{d\gamma}\right)^*$

$$\left(\frac{d^2 j}{d\gamma^2}\right)\delta = \frac{d}{d\gamma}\left(\frac{dj}{d\gamma}\right)^*\delta = \frac{d}{d\gamma}\left(\frac{\partial J}{\partial \gamma}\right)^*\delta - \frac{d}{d\gamma}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^*\Pi\right]\delta \tag{3.27}$$

The first term is

$$\frac{d}{d\gamma}\left(\frac{\partial J}{\partial \gamma}\right)^*\delta = \frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial \gamma}\right)^*\delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma}\right)^*\theta \tag{3.28}$$

while the second one is

$$\frac{d}{d\gamma}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^*\Pi\right]\delta = \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial \gamma}\right)^*\Pi\right]\delta + \left(\frac{\partial \Psi}{\partial \gamma}\right)^*\frac{d\Pi}{d\gamma}\delta \ . \tag{3.29}$$

Performing a tangent derivative of the adjoint equation $\left(\frac{\partial \Psi}{\partial W}\right)^*\Pi - \left(\frac{\partial J}{\partial W}\right)^* = 0$ along the direction δ we obtain

$$\begin{aligned} 0 &= \frac{d}{d\gamma}\left[\left(\frac{\partial \Psi}{\partial W}\right)^*\Pi - \left(\frac{\partial J}{\partial W}\right)^*\right]\delta = \\ &= \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial W}\right)^*\Pi\right]\delta + \left(\frac{\partial \Psi}{\partial W}\right)^*\frac{d\Pi}{d\gamma}\delta - \frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^*\delta - \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^*\theta \end{aligned}$$

and therefore

$$\frac{d\Pi}{d\gamma}\delta = \left(\frac{\partial \Psi}{\partial W}\right)^{-*}\left\{\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^{*}\delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^{*}\theta - \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial W}\right)^{*}\Pi\right]\delta\right\}\ . \tag{3.30}$$

Putting the (3.28)–(3.30) into the (3.27) we obtain the projection of the Hessian $\frac{d^2 j}{d\gamma^2}$ along a generic direction δ

$$\begin{aligned}\left(\frac{d^2 j}{d\gamma^2}\right)\delta &= \frac{d}{d\gamma}\left(\frac{\partial J}{\partial \gamma}\right)^{*}\delta - \frac{d}{d\gamma}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^{*}\Pi\right]\delta = \\ &= \frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial \gamma}\right)^{*}\delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma}\right)^{*}\theta - \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial \gamma}\right)^{*}\Pi\right]\delta + \\ &\quad - \left(\frac{\partial \Psi}{\partial \gamma}\right)^{*}\left(\frac{\partial \Psi}{\partial W}\right)^{-*}\left\{\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^{*}\delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^{*}\theta - \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial W}\right)^{*}\Pi\right]\delta\right\}\ .\end{aligned}$$

Evaluating this last expression at the point (γ_h, W_h) and remembering that for $x = \{\gamma, W\}$ we have

$$\left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial x}\right)^{*}\Pi\right]\bigg|_{(\gamma_h,W_h)} = \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial x}\right)^{*}\Pi_h\right]\bigg|_{(\gamma_h,W_h)} = \left[\frac{d}{d\gamma}\left(\left(\frac{\partial \Psi}{\partial x}\right)^{*}\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)}$$

we obtain the multiplication of the Hessian matrix j'' by the vector δ

$$\begin{aligned}\left(\frac{d^2 j}{d\gamma^2}\bigg|_{\gamma_h}\right)\delta &= \left[\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial \gamma}\right)^{*}\right]\bigg|_{(\gamma_h,W_h)}\delta + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma}\right)^{*}\right]\bigg|_{(\gamma_h,W_h)}\theta_h + \\ &\quad - \left[\frac{\partial}{\partial \gamma}\left(\left(\frac{\partial \Psi}{\partial \gamma}\right)^{*}\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)}\delta - \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial \gamma}\right)^{*}\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)}\theta_h + \\ &\quad - \left(\frac{\partial \Psi}{\partial \gamma}\bigg|_{(\gamma_h,W_h)}\right)^{*}\lambda_h\end{aligned}$$

where θ_h is the solution of the linear system

$$\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h))}\right)\theta_h = -\left(\frac{\partial \Psi}{\partial \gamma}\bigg|_{(\gamma_h,W_h))}\right)\delta$$

and $\lambda_h = \left(\frac{d\Pi}{d\gamma}\bigg|_{(\gamma_h,W_h)}\right)\delta$ can be computed solving the linear system

$$\begin{aligned}\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{*}\lambda_h &= \left[\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^{*}\right]\bigg|_{(\gamma_h,W_h)}\delta + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^{*}\right]\bigg|_{(\gamma_h,W_h)}\theta_h + \\ &\quad - \left[\frac{\partial}{\partial \gamma}\left(\left(\frac{\partial \Psi}{\partial W}\right)^{*}\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)}\delta - \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial W}\right)^{*}\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)}\theta_h\ .\end{aligned}$$

□

If we apply the Lemma 3.1 using $\delta = e_i$ (where $e_i = (0, \dots, 1, \dots, 0)^T$ is the vector having the only not-zero value at the i-th position, i.e. the i-th element of the canonical basis of $\mathbb{R}^n$), it means that we are computing the i-th column (and, by symmetry, the i-th row) of the Hessian, obtaining

$$\boxed{\begin{aligned}\left(\frac{d^2 j}{d\gamma^2}\bigg|_{\gamma_h}\right) e_i = &\left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}\right)^*\right]\bigg|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}\right)^*\right]\bigg|_{(\gamma_h,W_h)} \theta_h^{(i)} + \\ &- \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial\Psi}{\partial\gamma}\right)^*\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)} e_i - \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial\Psi}{\partial\gamma}\right)^*\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)} \theta_h^{(i)} + \\ &- \left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma_h,W_h)}\right)^* \lambda_h^{(i)}\end{aligned}} \tag{3.31}$$

Then, to compute the full Hessian, we need to apply the Hessian-by-vector multiplication (3.31) to each component of the canonical basis of $\mathbb{R}^n$.

For each $i = 1, \dots, n$, the equation (3.31) needs the adjoint state Π_h, solution of the adjoint system

$$\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^* \Pi_h = \left(\frac{\partial J}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^* \tag{3.32}$$

and the arrays $\theta_h^{(i)}$, $\lambda_h^{(i)}$ solutions of the linear systems:

$$\left\{\begin{aligned}&\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)\theta_h^{(i)} = -\left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma_h,W_h)}\right) e_i \\ &\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^* \lambda_h^{(i)} = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial W}\right)^*\right]\bigg|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^*\right]\bigg|_{(\gamma_h,W_h)} \theta_h^{(i)} + \\ &\qquad - \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial\Psi}{\partial W}\right)^*\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)} e_i - \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial\Psi}{\partial W}\right)^*\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)} \theta_h^{(i)}\end{aligned}\right. \tag{3.33}$$

where all the derivatives in the equations (3.31)–(3.33) are evalued at the final state W_h. Moreover, the second linear system in (3.33) is of the same type of the adjoint system (3.32) but with a different right hand side, so we can use the same matrix-free algorithm and the same preconditioner (but with different right hand sides) for both equations.

Implementation of the Tangent-on-Reverse derivatives As we have done in Section 3.6.1 for ToT approach, let us suppose that the subroutine computing the state residual $\Psi(\gamma, W)$ is `state_residuals(psi,gamma,w)`, where the input variables are `gamma` and `w`, and the output variable is `psi`.

$$\mathtt{state\_residuals}(\underset{\downarrow}{\mathtt{psi}}, \overset{\downarrow}{\mathtt{gamma}}, \overset{\downarrow}{\mathtt{w}})$$

If we perform a differentiation in reverse mode with respect to the input variables `gamma` and `w` we have

$$\texttt{state\_residuals\_b}(\underset{\downarrow}{\texttt{psi}}, \overset{\downarrow}{\texttt{psib}}, \overset{\downarrow}{\texttt{gamma}}, \underset{\downarrow}{\texttt{gammab}}, \overset{\downarrow}{\texttt{w}}, \underset{\downarrow}{\texttt{wb}})$$

where $\texttt{gammab} = \bar{\gamma}_\Psi = \left(\frac{\partial\Psi}{\partial\gamma}\right)^*\bar{\Psi}$ and $\texttt{wb} = \bar{W}_\Psi = \left(\frac{\partial\Psi}{\partial W}\right)^*\bar{\Psi}$ are the new output variables and calling $\texttt{psib} = \bar{\Psi}$ the additional input variable.

Now we differentiate the routine `state_residuals_b` in tangent mode (with respect to the same input variables `gamma` and `w`) considering `gammab` and `wb` as output variables, obtaining

$$\texttt{state\_residuals\_b\_d}(\underset{\downarrow}{\texttt{psi}}, \overset{\downarrow}{\texttt{psib}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{gammad}}, \underset{\downarrow}{\texttt{gammab}}, \underset{\downarrow}{\texttt{gammabd}}, \overset{\downarrow}{\texttt{w}}, \overset{\downarrow}{\texttt{wd}}, \underset{\downarrow}{\texttt{wb}}, \underset{\downarrow}{\texttt{wbd}}) \tag{3.34}$$

where $\texttt{gammad} = \dot{\gamma}$, $\texttt{wd} = \dot{W}$ and the second-order derivatives are in the variables

$$\begin{aligned}
\texttt{gammabd} = \dot{\bar{\gamma}}_\Psi &= \frac{\partial}{\partial\gamma}\left[\left(\frac{\partial\Psi}{\partial\gamma}\right)^*\bar{\Psi}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial\Psi}{\partial\gamma}\right)^*\bar{\Psi}\right]\dot{W} \\
\texttt{wbd} = \dot{\bar{W}}_\Psi &= \frac{\partial}{\partial\gamma}\left[\left(\frac{\partial\Psi}{\partial W}\right)^*\bar{\Psi}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial\Psi}{\partial W}\right)^*\bar{\Psi}\right]\dot{W}\ .
\end{aligned}$$

In order to solve the equations equations (3.31)–(3.33) we should call the routine (3.34) with the right arguments:

$$\texttt{state\_residuals\_b\_d}(\underset{\Psi}{\texttt{psi}}, \overset{\Pi_h}{\texttt{psib}}, \overset{\gamma_h}{\texttt{gamma}}, \overset{e_i}{\texttt{gammad}}, \underset{\left(\frac{\partial\Psi}{\partial\gamma}\right)^*\Pi_h}{\texttt{gammab}}, \underset{\dot{\bar{\gamma}}_\Psi}{\texttt{gammabd}}, \overset{W_h}{\texttt{w}}, \overset{\theta_h^{(i)}}{\texttt{wd}}, \underset{\left(\frac{\partial\Psi}{\partial W}\right)^*\Pi_h}{\texttt{wb}}, \underset{\dot{\bar{W}}_\Psi}{\texttt{wbd}}) \tag{3.35}$$

where

$$\begin{aligned}
\dot{\bar{\gamma}}_\Psi &= \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial\Psi}{\partial\gamma}\right)^*\Pi_h\right)\right]\Bigg|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial\Psi}{\partial\gamma}\right)^*\Pi_h\right)\right]\Bigg|_{(\gamma_h,W_h)} \theta_h^{(i)} \\
\dot{\bar{W}}_\Psi &= \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial\Psi}{\partial W}\right)^*\Pi_h\right)\right]\Bigg|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial\Psi}{\partial W}\right)^*\Pi_h\right)\right]\Bigg|_{(\gamma_h,W_h)} \theta_h^{(i)}\ .
\end{aligned}$$

If the subroutine computing the functional $J(\gamma, W)$ is

$$\texttt{functional}(\underset{\downarrow}{\texttt{j}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{w}})$$

performing a reverse mode differentiation with respect to `gamma` and `w`

$$\texttt{functional\_b}(\underset{\downarrow}{\texttt{j}}, \overset{\downarrow}{\texttt{jb}}, \overset{\downarrow}{\texttt{gamma}}, \underset{\downarrow}{\texttt{gammab}}, \overset{\downarrow}{\texttt{w}}, \underset{\downarrow}{\texttt{wb}})$$

and then a tangent mode differentiation of the output variables `gammab` and `wb` with respect to `gamma` and `w`

$$\texttt{functional\_b\_d}(\underset{\downarrow}{\texttt{j}}, \overset{\downarrow}{\texttt{jb}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{gammad}}, \underset{\downarrow}{\texttt{gammab}}, \underset{\downarrow}{\texttt{gammabd}}, \overset{\downarrow}{\texttt{w}}, \overset{\downarrow}{\texttt{wd}}, \underset{\downarrow}{\texttt{wb}}, \underset{\downarrow}{\texttt{wbd}}) \tag{3.36}$$

where `jb` $= \bar{J}$, `gammad` $= \dot{\gamma}$, `wd` $= \dot{W}$ and

$$\begin{aligned}\texttt{gammabd} = \dot{\bar{\gamma}}_J &= \frac{\partial}{\partial\gamma}\left[\left(\frac{\partial J}{\partial\gamma}\right)^* \bar{J}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial J}{\partial\gamma}\right)^* \bar{J}\right]\dot{W} \\ \texttt{wbd} = \dot{\bar{W}}_J &= \frac{\partial}{\partial\gamma}\left[\left(\frac{\partial J}{\partial W}\right)^* \bar{J}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial J}{\partial W}\right)^* \bar{J}\right]\dot{W}\ .\end{aligned}$$

As usual, we call the routine above with the right arguments to obtain needed quantities in the equations (3.31) and (3.33), i.e.

$$\texttt{functional\_b\_d}(\underset{J}{\texttt{j}}, \overset{1.0}{\texttt{jb}}, \overset{\gamma_h}{\texttt{gamma}}, \overset{e_i}{\texttt{gammad}}, \underset{\left(\frac{\partial J}{\partial\gamma}\right)^*}{\texttt{gammab}}, \underset{\dot{\bar{\gamma}}_J}{\texttt{gammabd}}, \overset{W_h}{\texttt{w}}, \overset{\theta_h^{(i)}}{\texttt{wd}}, \underset{\left(\frac{\partial J}{\partial W}\right)^*}{\texttt{wb}}, \underset{\dot{\bar{W}}_J}{\texttt{wbd}}) \tag{3.37}$$

where

$$\begin{aligned}\dot{\bar{\gamma}}_J &= \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}\right)^*\right]\Bigg|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}\right)^*\right]\Bigg|_{(\gamma_h,W_h)} \theta_h^{(i)} \\ \dot{\bar{W}}_J &= \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial W}\right)^*\right]\Bigg|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^*\right]\Bigg|_{(\gamma_h,W_h)} \theta_h^{(i)}\ .\end{aligned}$$

Description of the algorithm for the Hessian matrix with the ToR approach. The algorithm to compute the Hessian matrix wit the ToR approach can be summarized as follow:

1. compute the state W_h such that $\Psi(\gamma_h, W_h) = 0$;
2. compute $\bar{W}_J = \left(\frac{\partial J}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^*$
3. compute the adjoint state Π_h solving the linear system $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^* \Pi_h = \bar{W}_J$;
4. for each element of the canonical basis e_i, $i = 1, \ldots, n$:
   (a) compute $\dot{\Psi}_\gamma = \left(\frac{\partial \Psi}{\partial \gamma}\Big|_{(\gamma_h,W_h)}\right) e_i$;
   (b) compute the vector $\theta_h^{(i)}$ solving the linear system $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)\theta_h^{(i)} = -\dot{\Psi}_\gamma$;
   (c) compute $\dot{\bar{W}}_J = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial W}\right)^*\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^*\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)}$;
   (d) compute $\dot{\bar{\gamma}}_J = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}\right)^*\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}\right)^*\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)}$;
   (e) compute $\dot{\bar{W}}_\Psi = \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial \Psi}{\partial W}\right)^*\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial W}\right)^*\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)}$;

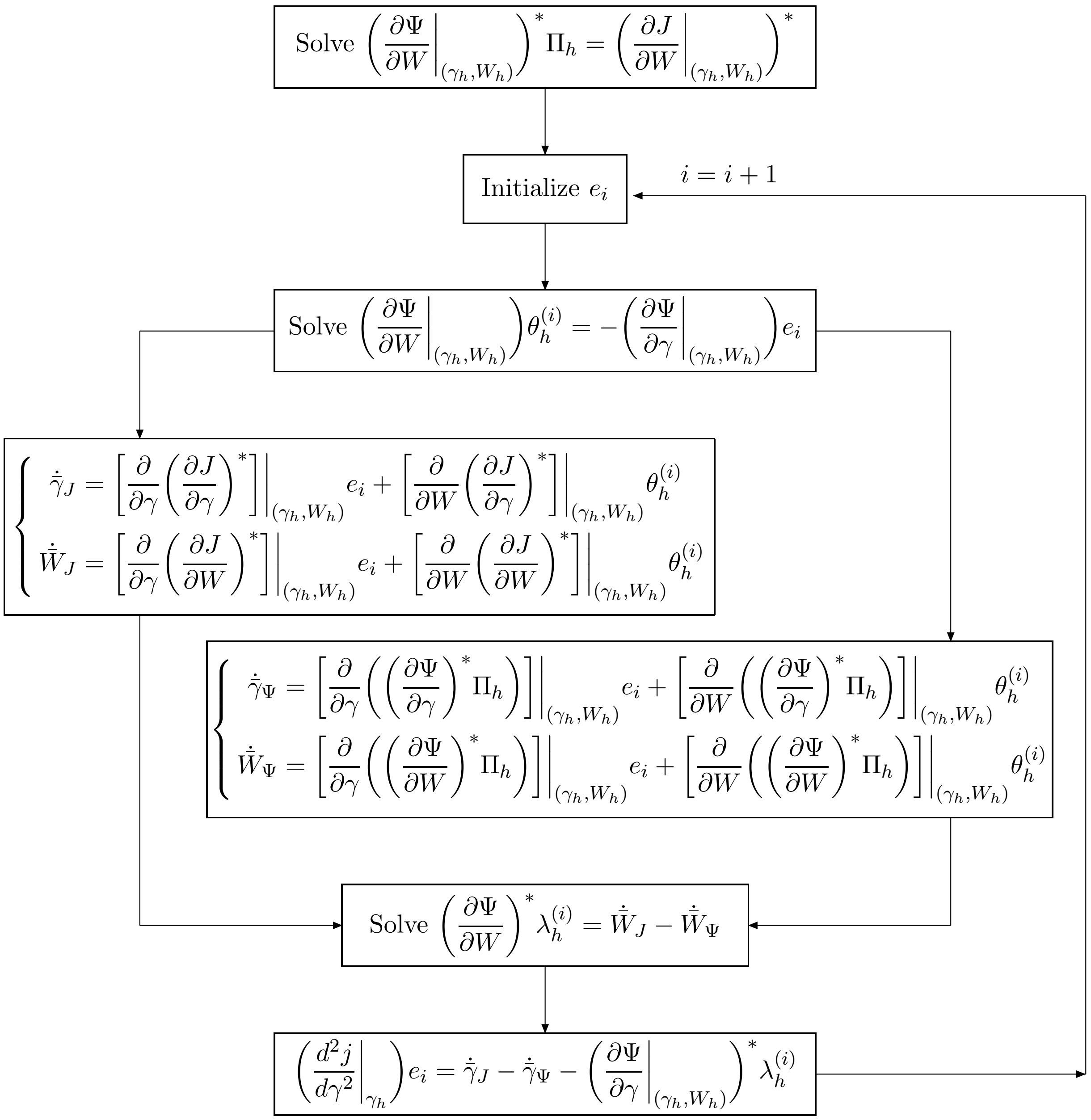

Solve $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^* \Pi_h = \left(\frac{\partial J}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^*$
Initialize e_i
$i = i + 1$
Solve $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)\theta_h^{(i)} = -\left(\frac{\partial \Psi}{\partial \gamma}\Big|_{(\gamma_h,W_h)}\right)e_i$
$\begin{cases} \dot{\bar{\gamma}}_J = \left[\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial \gamma}\right)^*\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma}\right)^*\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)} \\ \dot{\bar{W}}_J = \left[\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^*\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^*\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)} \end{cases}$
$\begin{cases} \dot{\bar{\gamma}}_\Psi = \left[\frac{\partial}{\partial \gamma}\left(\left(\frac{\partial \Psi}{\partial \gamma}\right)^*\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial \gamma}\right)^*\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)} \\ \dot{\bar{W}}_\Psi = \left[\frac{\partial}{\partial \gamma}\left(\left(\frac{\partial \Psi}{\partial W}\right)^*\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial W}\right)^*\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)} \end{cases}$
Solve $\left(\frac{\partial \Psi}{\partial W}\right)^* \lambda_h^{(i)} = \dot{\bar{W}}_J - \dot{\bar{W}}_\Psi$
$\left(\frac{d^2 j}{d\gamma^2}\Big|_{\gamma_h}\right)e_i = \dot{\bar{\gamma}}_J - \dot{\bar{\gamma}}_\Psi - \left(\frac{\partial \Psi}{\partial \gamma}\Big|_{(\gamma_h,W_h)}\right)^* \lambda_h^{(i)}$

Figure 3.8: Tangent-on-Reverse algorithm

(f) compute $\dot{\bar{\gamma}}_\Psi = \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial\Psi}{\partial\gamma}\right)^*\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial\Psi}{\partial\gamma}\right)^*\Pi_h\right)\right]\bigg|_{(\gamma_h,W_h)} \theta_h^{(i)}$;

(g) compute the vector $\lambda_h^{(i)}$ solving the linear system

$$\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^*\lambda_h^{(i)} = \dot{\bar{W}}_J - \dot{\bar{W}}_\Psi \ ;$$

(h) compute $\bar{\gamma}_\Psi = \left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma_h,W_h)}\right)\lambda_h^{(i)}$;

(i) compute the i-th column (or row) of the Hessian matrix:

$$\left(\frac{dj}{d\gamma}\bigg|_{\gamma_h}\right)e_i = \dot{\bar{\gamma}}_J - \dot{\bar{\gamma}}_\Psi - \bar{\gamma}_\Psi \ .$$

From the previous algorithm we see that for each column of the Hessian matrix we need to solve 2 linear systems:

- one is to compute the vector $\theta_h^{(i)}$ (step 4b) and can be solved with the matrix-free algorithm `matrixfree_solve_linearsystem` (see Section 3.3);
- the other is to compute the vector $\lambda_h^{(i)}$ (step 4g) and can be solved with the matrix-free algorithm `matrixfree_solve_adjointlinearsystem` (see Section 3.3).

Moreover, the quantities $\dot{\bar{W}}_J$, $\dot{\bar{\gamma}}_J$ (steps 4c and 4d) could be obtained with a single invocation of the twice-differentiated subroutine (3.37); while the quantities $\dot{\bar{W}}_\Psi$, $\dot{\bar{\gamma}}_\Psi$ (steps 4e and 4f) could be obtained with a single invocation of the differentiated-twice subroutine (3.35).

The computational cost for a single Hessian-by-vector multiplication, evaluated with the Tangent-on-Reverse approach, can be estimated with the same assumptions made in Section 3.6.1, and is due to the following contributions:

- $\alpha_T n_{\text{iter},T}$ for computing the derivatives of the state variables respect to the control $\theta_h^{(i)} = \frac{dW}{d\gamma_i}$ (step 4b), where $n_{\text{iter},T}$ is the number of iterations needed by the matrix-free algorithm to solve the linear system;
- $\alpha_R\alpha_T$ for evaluating the quantities $\dot{\bar{W}}_\Psi$, $\dot{\bar{\gamma}}_\Psi$ (steps 4e and 4f) with the single invocation of the subroutine (3.35);
- $\alpha_R n_{\text{iter},R}$ for computing the vector $\lambda_h^{(i)}$ (step 4g), where $n_{\text{iter},R}$ is the number of iterations needed by the matrix-free algorithm to solve adjoint systems.

Therefore the full Hessian evaluation with ToR costs

$$n\alpha_T\big(n_{\text{iter},T} + \alpha_R + \frac{\alpha_R}{\alpha_T}n_{\text{iter},R}\big)$$

and we note that the major contribution is due to the solution of the linear systems, usually being the number of iterations to the convergence $\gg \alpha_R$.

Another important thing to note is the fact that with ToR we cannot compute the diagonal of the Hessian without computing the extra-diagonal values, due to the fact that the Hessian is built column-by-column (or, by symmetry, row-by-row) using the Lemma 3.1 on the elements of the canonical basis.

As minor remark, ToR approach for the full Hessian does not need to store the derivatives of the state variables respect to the control $\theta_h^{(i)} = \frac{dW}{d\gamma_i}$ for all $i = 1, \dots, n$, but it can use the same memory locations for the various $\theta_h^{(i)}$, resulting in a memory saving and in a serialization of the algorithm, while using a different location for each vector results in an easily parallelizable algorithm (each Hessian-by-vector multiplication is independent from the others, so each evaluation can be run in parallel).

3.6.3 Comparison between ToT and ToR

At this point, the natural question arising from the previous analysis is about the choice of the method that is less expensive for a given problem. The cost to evaluate the full Hessian, its diagonal part and the Hessian-by-vector multiplication for the two different strategies is given in the Table 3.1, where we do not take into account the cost to solve the state equation $\Psi = 0$ and to solve the adjoint system (3.18).

	Hessian (full)	Hessian (diagonal)	Hessian-by-vector
ToT	$n\alpha_T\left[n_{\text{iter},T} + \frac{(n+1)}{2}\alpha_T\right]$	$n\alpha_T\left[n_{\text{iter},T} + \alpha_T\right]$	$n\alpha_T\left[n_{\text{iter},T} + \alpha_T\right]$
ToR	$n\alpha_T\left(n_{\text{iter},T} + \alpha_R + \frac{\alpha_R}{\alpha_T} n_{\text{iter},R}\right)$	—	$\alpha_T\left(n_{\text{iter},T} + \alpha_R + \frac{\alpha_R}{\alpha_T} n_{\text{iter},R}\right)$

Table 3.1: ToT and ToR comparison. Computational cost for the evaluation of the full $n \times n$ Hessian matrix, only its diagonal part and the Hessian-by-vector multiplication. α_T, α_R are the overheads associated with the tangent- and reverse-mode differentiation for the subroutine implementing the evaluation of the state residual. $n_{\text{iter},T}$, $n_{\text{iter},R}$ are the number of iterations needed for the matrix-free algorithm to solve the tangent and adjoint linear system, respectively. The values in the table do not take into account the runtime cost to compute the adjoint state Π_h, that is assumed to be available. The cost to compute the adjoint state Π_h as solution of the adjoint linear system (3.18) can be estimated as $\alpha_R n_{\text{iter},R}$.

From the algorithms in Sections 3.6.1 and 3.6.2 we note that the two approaches to evaluate the full Hessian share a common part, namely the computation of the derivatives of the state variables respect to the control $\frac{dW}{d\gamma_i}$ $(i = 1, \dots, n)$ as solution of the linear system

$$\left(\left.\frac{\partial \Psi}{\partial W}\right|_{(\gamma_h, W_h)}\right)\frac{dW}{d\gamma_i} = -\left(\left.\frac{\partial \Psi}{\partial \gamma}\right|_{(\gamma_h, W_h)}\right)e_i \ .$$

This cost appears in Table 3.1 as the $n\alpha_T n_{\text{iter},T}$ term, therefore the characteristic cost grows as $\frac{n(n+1)}{2}\alpha_T^2$ for the ToT approach and $n\alpha_R\big(\alpha_T + n_{\text{iter},R}\big)$ for ToR. Thus we can say that, *using a single strategy* to compute the full Hessian, ToT has a lower computational cost with respect to ToR if

$$n < 2\frac{\alpha_R}{\alpha_T}\big(1 + \frac{n_{\text{iter},R}}{\alpha_T}\big) - 1.$$

Therefore ToT is cheaper than ToR if the dimension n of the control γ is small. This last result, can be used to build better strategy (i.e. less time-consuming) for the full Hessian *using ToT and ToR for the evaluation of different parts of the Hessian.*

Let us consider $\bar{n}$, a given number not directly dependent on n, the key idea is to use, when the Hessian dimension is larger then $\bar{n}$ ToT to build the upper triangular part of the Hessian until the $\bar{n}$-th column and then evaluate the remaining $n - \bar{n}$ columns with ToR (using the Hessian-by-vector multiplication). On the other hand, only ToT is used when the Hessian dimension is smaller then $\bar{n}$. The cost of this mixed strategy can be evaluated as

$$\begin{cases} n\alpha_T n_{\text{iter},T} + \dfrac{n(n+1)}{2}\alpha_T^2 & \text{for } n \le \bar{n} \\[2ex] n\alpha_T n_{\text{iter},T} + \dfrac{\bar{n}(\bar{n}+1)}{2}\alpha_T^2 + (n-\bar{n})\alpha_R\big(\alpha_T + n_{\text{iter},R}\big) & \text{for } n > \bar{n} \end{cases}$$

where optimal value for $\bar{n}$ is found to be $\bar{n} = \dfrac{\alpha_R}{\alpha_T}\big(1 + \dfrac{n_{\text{iter},R}}{\alpha_T}\big)$. For a given problem we can assume that the values α_T, α_R can be obtained with not too much effort in a preprocessing phase using program profiling. Much more difficult could be the estimate of $n_{\text{iter},R}$, the number of iterations needed to solve the adjoint linear system, in fact this number depends on many factors: the dimension of the problem itself, the dimension of the Krylov space, the kind of preconditioner used, etc. A comparison for the cost of the full Hessian using different strategies is given in Figure 3.9 where we assumed $n_{\text{iter},R} = n_{\text{iter},T} = 300$ for the number of iterations needed to solve the linear systems (direct and adjoint) and $\alpha_T = 2$, $\alpha_R = 4$ for the overhead associated with the Tangent- and Reverse-differentiation (obtaining the corresponding treshold value for the mixed strategy $\bar{n} = 302$).

3.7 Stack management issue for ToR approach

The ToT strategy is very simple to implement and very well managed by TAPENADE, in fact a single Tangent-mode differentiation adds only extra code that is executed in the same order of the original: this means that the differentiated code is considered as "normal" code for the second differentiation and therefore this strategy could be used without any difficulty to obtain higher-order derivatives.

To the opposite, Tangent-on-Reverse differentiation raises some Automatic Differentiation issues because of the specific structure of reverse differentiated programs. As we saw in Section 3.2, programs differentiated in reverse with the Store-All approach make heavy use of `PUSH` and `POP` primitives to store and retrieve intermediate values. In comparison, the Recompute-All

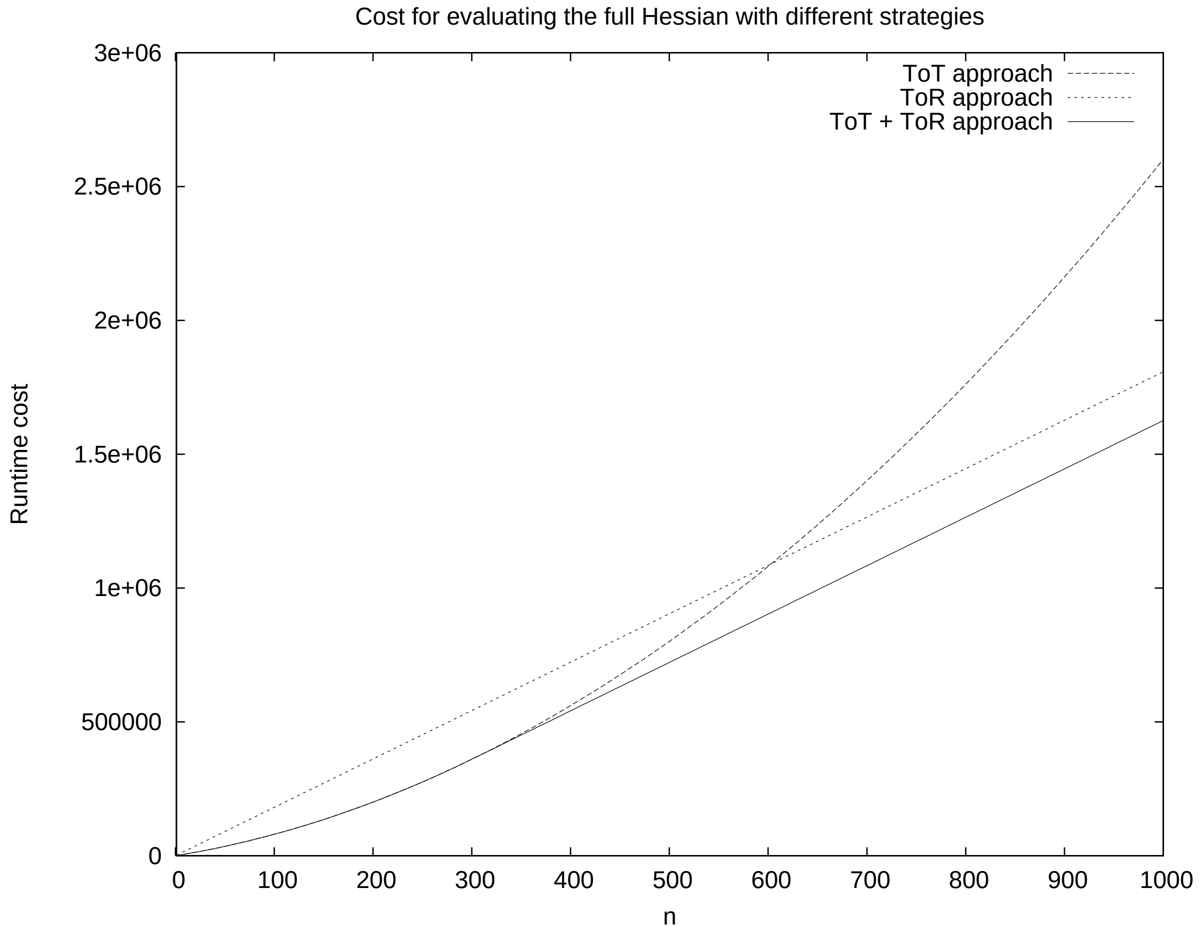


Figure 3.9: Comparison for the cost of the full Hessian using different strategies as a function of n, the dimension of the control variable γ. We assumed $n_{\text{iter},R} = n_{\text{iter},T} = 300$ for the number of iterations needed to solve the linear systems (tangent and adjoint) and $\alpha_T = 2$, $\alpha_R = 4$ for the overhead associated with the Tangent- and Reverse-differentiation (obtaining the corresponding treshold value for the mixed strategy $\bar{n} = 302$).

approach does not *a priori* use PUSH and POP's. However both approaches must use checkpointing for nontrivial applications, which means storing and retrieving snapshots, which boils down to PUSH and POP's in the end.

Thus the structure of a reverse-differentiated program is unusual. Indeed we never found a similar structure in the numerical applications that we have differentiated so far. It is true that AD tools claim to be able to differentiate any program, and this claim is motivated by the general theory sketched in Section 3.2 (in which we assume that a program ca be considered as a sequence of instruction that can be identified as a composition of differentiable functions). There exist a few documented limitations to this theory, such as switches in the control-flow that may cause local non-differentiability, but nothing related to stack management. Still, we felt a little anxious when it came to differentiate reverse programs, and rightly so.

Calls to PUSH and POP communicate values through a stack, which is a *hidden global variable.* Moreover, the source code of PUSH and POP belongs to an external library and is not given to the AD tool. The classical way to handle these "black-box" routines in AD tools is to provide the tool with just enough information about how the so-called "activity" propagates across the black-boxes. Activity is a boolean information attached to each occurrence of a variable in the code, which is true when the variable has a nontrivial derivative and false otherwise. During differentiation, the AD tool generates calls to differentiated black-box routines, that the end-user must code by hand in the end. This black-box mechanism is very general, and has been used many times on real codes. For example, this is how calls to MPI communication routines are handled by AD tools. However this mechanism requires extreme care, as we will see. Making this mechanism easier and safer needs further research.

First, we characterize how activity propagates across PUSH and POP. There are two variables involved:

- the first argument of the PUSH (resp. POP) is the program variable stored (resp. retrieved). Let us call it V.
- the hidden stack that keeps all the stored values can be thought of as an array. Let us call it S. Initial stack S is empty and therefore not active a priori.

A PUSH does not change the value nor the activity of V. On the other hand if V is active, then S becomes or remains active. Unfortunately, activity of S is a single boolean that mixes or blurs the activity of all stored values, therefore the effect of a POP can not be described very accurately. We say that a POP never changes the activity of S, and it returns an active V if and only if S is active.

Figure 3.10 illustrates Tangent-on-Reverse differentiation on a small example code, displayed on the left. The reverse differentiated code has the usual two-sweeps structure, with two PUSH/POP pairs because the value of x needs to be restored. The tangent differentiation that follows begins with activity analysis. Examining the reverse code, one can check that x first receives a constant and thus is non active. The first PUSH, having non active arguments, does not need to be differentiated. When x is incremented by c, it becomes active and so the second PUSH makes the stack active. During the reverse sweep, the first POP has an active stack and thus returns an active x and a still active stack. Since the stack is active, the second POP also returns an x which is considered active. After activity analysis, tangent differentiation produces the code in the third column. The important point is that there is now an unexpected PUSH/POP_D pair. In a naive first attempt we implemented PUSH_D and POP_D using the same stack to store the value and its derivative, and the second POP_D causes a segmentation fault because the stack is empty. In more complex examples, the POP_D might even pop the next top of stack which was expected to be read by a subsequent POP.

Original	**Reverse**	**Tangent-on-Reverse**
F : a, b, c $\mapsto$ r	$\overline{F}$: a, b, c, $\bar{\text{r}}$ $\mapsto$ $\bar{\text{a}}$, $\bar{\text{b}}$, $\bar{\text{c}}$	$\dot{\overline{F}}$: a, $\dot{\text{a}}$, b, $\dot{\text{b}}$, c, $\dot{\text{c}}$, $\bar{\text{r}}$, $\dot{\bar{\text{r}}}$ $\mapsto$ $\bar{\text{a}}$, $\dot{\bar{\text{a}}}$, $\bar{\text{b}}$, $\dot{\bar{\text{b}}}$, $\bar{\text{c}}$, $\dot{\bar{\text{c}}}$
		$\dot{\text{x}}$= 0.0
x = 2.0	x = 2.0	x = 2.0
r = x*a	r = x*a	
	PUSH(x)	PUSH(x)
		$\dot{\text{x}}$= $\dot{\text{c}}$
x += c	x += c	x += c
r += x*b	r += x*b	
	PUSH(x)	PUSH_D(x,$\dot{\text{x}}$)
		$\dot{\text{x}}$= 0.0
x = 3.0	x = 3.0	x = 3.0
r += x*c	r += x*c	
	$\bar{\text{x}}$= c*$\bar{\text{r}}$	
		$\dot{\bar{\text{c}}}$+= x*$\dot{\bar{\text{r}}}$
	$\bar{\text{c}}$+= x*$\bar{\text{r}}$	$\bar{\text{c}}$+= x*$\bar{\text{r}}$
	POP(x)	POP_D(x,$\dot{\text{x}}$)
	$\bar{\text{x}}$= 0.0	
		$\dot{\bar{\text{x}}}$= $\dot{\text{b}}$*$\bar{\text{r}}$+b*$\dot{\bar{\text{r}}}$
	$\bar{\text{x}}$= b*$\bar{\text{r}}$	$\bar{\text{x}}$= b*$\bar{\text{r}}$
		$\dot{\bar{\text{b}}}$+= $\dot{\text{x}}$*$\bar{\text{r}}$+x*$\dot{\bar{\text{r}}}$
	$\bar{\text{b}}$+= x*$\bar{\text{r}}$	$\bar{\text{b}}$+= x*$\bar{\text{r}}$
	POP(x)	POP_D(x,$\dot{\text{x}}$)
		$\dot{\bar{\text{c}}}$+= $\dot{\bar{\text{x}}}$
	$\bar{\text{c}}$+= $\bar{\text{x}}$	$\bar{\text{c}}$+= $\bar{\text{x}}$
	$\bar{\text{x}}$+= a*$\bar{\text{r}}$	
		$\dot{\bar{\text{a}}}$+= $\dot{\text{x}}$*$\bar{\text{r}}$+x*$\dot{\bar{\text{r}}}$
	$\bar{\text{a}}$+= x*$\bar{\text{r}}$	$\bar{\text{a}}$+= x*$\bar{\text{r}}$
	$\bar{\text{x}}$= 0.0	

Figure 3.10: Tangent-on-Reverse differentiation on a small code. *Left*: Original code, *middle*: Reverse code, *right*: ToR code. Reverse-differentiated variables are shown with a bar above, as in $\bar{\text{x}}$. Tangent-differentiated variables are shown with a dot above, as in $\dot{\text{x}}$ and $\dot{\bar{\text{x}}}$. The reverse code produced with TAPENADE is actually shorter because of static data-flow analysis: the code in light grey is stripped away, but this has no influence on the demonstration.

What has gone wrong? Could it be an error in the tangent differentiation model? To check that, we considered an equivalent reversed program, with the source of PUSH and POP made explicit, using a simple array to store the values. Then the Tangent-on-Reverse program works fine. We notice that it has produced a differentiated array to store differentiated values. Thus, the problem does not lie with the differentiation model, but rather with the hand-written code for `PUSH_D` and `POP_D` that use the same stack for `x` and $\dot{\texttt{x}}$. With two different stacks, the second `POP_D` finds an empty stack and returns 0.0, which is consistent with the fact that this `x` is actually non active. Equivalently, we can simply decide that if at any time activity actually propagates across the hidden communication variable, then the communication variable is active right from the beginning. This would turn the first `PUSH` into a `PUSH_D`. This is what we did for this work, and the Tangent-on-Reverse strategy now produces correct results.

A much more elegant solution would be to keep the information on matching `PUSH`/`POP` pairs in the Reverse code. So we would know when a `POP` returns a non-active variable. Pairs `PUSH_D`/`POP_D` would be reserved for really active variables, therefore reducing the stack size for the Tangent-on-Reverse strategy. We are considering to implement this feature in TAPENADE.

Table 3.2 illustrates these two solutions. Top of the table corresponds to the wrong result that TAPENADE returns if the stack is not activated. The middle solution is what we obtain when activating the stack. The bottom solution is the desidered better solution if TAPENADE was able to keep track of matching `PUSH`/`POP`.

3.8 . . . putting ToT and ToR into the practice

In Sections 3.6.1-3.6.2 we have presented two approaches to compute the Hessian and Hessian-related quantities (like its diagonal part or Hessian-by-vector multiplication) of a constrained functional, and at first sight it could appear very complicated to put in practice for codes of industrial complexity-level. This is not (entirely) true.

From the equations involved in the algorithms, we note that all the quantities used in both approaches are combination of derivatives of the state residual Ψ and the functional J, therefore we can implement the algorithms (ToT , ToR, mixed ToT/ToR, matrix-free methods for solving linear systems, routines for validation, etc.) referring to the subroutines for the *generic functions* (and *a priori* not defined yet) $\Psi(\gamma, W)$ and $J(\gamma, W)$ that we have called **`state_residual(psi,gamma,w)`** and **`functional(j,gamma,w)`** respectively. In this way we are considering the framework as composed by two parts: one is relative to the implementation of the algorithms and the other is relative to the implementation of the differentiated routines (see Figure 3.11). The algorithms side is *independent* from the definition of Ψ and J, so we can build (and compile) a library containing all the algorithms above, and when a specific problem has to be solved, the user has to take care of the derivatives side only, i.e. implement the correct residual and the functional and then compute (with an AD tool) the corresponding differentiated routines.

This approach results in a very flexible scheme and in a rapid application to a given problem: the final user must only

- provide its definitions for the functional and the state residual (in our experience this is the most difficult task, due to the fact that the state residual usually must be extracted from bad-written existing codes);

Reverse		Tangent-on-Reverse
`...`		`...`
`PUSH(a)`		`PUSH(a)`
`...`		`...`
`PUSH(b)`		`PUSH_D(b,bd)`
`...`	$\xrightarrow{\text{Tangent-mode differentiation}}$	`...`
`POP(b)`		`POP_D(b,bd)`
`...`		`...`
`POP(a)`		`POP_D(a,ad)`
`...`		`...`

Reverse		Tangent-on-Reverse
`...`		`...`
`PUSH(a)`		`PUSH_D(a,ad)`
`...`		`...`
`PUSH(b)`		`PUSH_D(b,bd)`
`...`	$\xrightarrow{\text{Tangent-mode differentiation}}$	`...`
`POP(b)`		`POP_D(b,bd)`
`...`		`...`
`POP(a)`		`POP_D(a,ad)`
`...`		`...`

Reverse		Tangent-on-Reverse
`...`		`...`
`PUSH(a)`		`PUSH(a)`
`...`		`...`
`PUSH(b)`		`PUSH_D(b,bd)`
`...`	$\xrightarrow{\text{Tangent-mode differentiation}}$	`...`
`POP(b)`		`POP_D(b,bd)`
`...`		`...`
`POP(a)`		`POP(a)`
`...`		`...`

Table 3.2: Stack management problem for ToR with TAPENADE. Assuming that `b` (in the Reverse-differentiated program) is an active variable (and `a` is inactive) for the Tangent-mode differentiation, the correct ToR program is shown at the bottom, while on top is what we get from the current version of TAPENADE (2.2.3-r1955) if we do not activate the initial stack: note the `PUSH/POP_D` pair on the variable `a`. In the middle is shown the straight (and not-so-elegant) solution that we used: we activate the stack from the beginning, storing the unneded variable `ad` in order to have a correct matching between `PUSH_D` and `POP_D` functions.

- run two scripts to drive the differentiation tool TAPENADE (one script is devoted to the first-order differentiation, the other one to the second-order differentiation);
- compile the resulting differentiated subroutines;
- link them with the library containing the algorithm for the derivatives;
- use the API within to compute the quantities of interests (i.e. gradient, Hessian, etc.);

where the three steps in the middle could be accomplished automatically using `Makefile`.

Following this framework, the user has only to respect one constraint: the subroutines that implement the residual and the functional evaluation must have the same interface used in the library containing the algorithms (in fact the library will refer to the differentiated routine by their interface).

Moreover, this constraint affects the name of the variables: in our examples, the user must use the name `gamma` for the control variables and `w` for the state variables, that are the names we used for the variables in the interface of the subroutines `state_residual` and `functional` referred by the library. The possible complication with this strategy is when the implementation for the state residual (or for the functional) uses the dependent and independent variables in global space (i.e. `COMMON` blocks in FORTRAN programs): when it happens, a possible solution is to copy the state and control variables (that are stored in the global space) to the corresponding `w` and `gamma` variables just *before* the invocation of the differentiated routines, and the inverse assignment (from the local `w` and `gamma` to the global state and control variables) as first instruction inside `state_residual(psi,gamma,W)` and `functional(J,gamma,W)`.

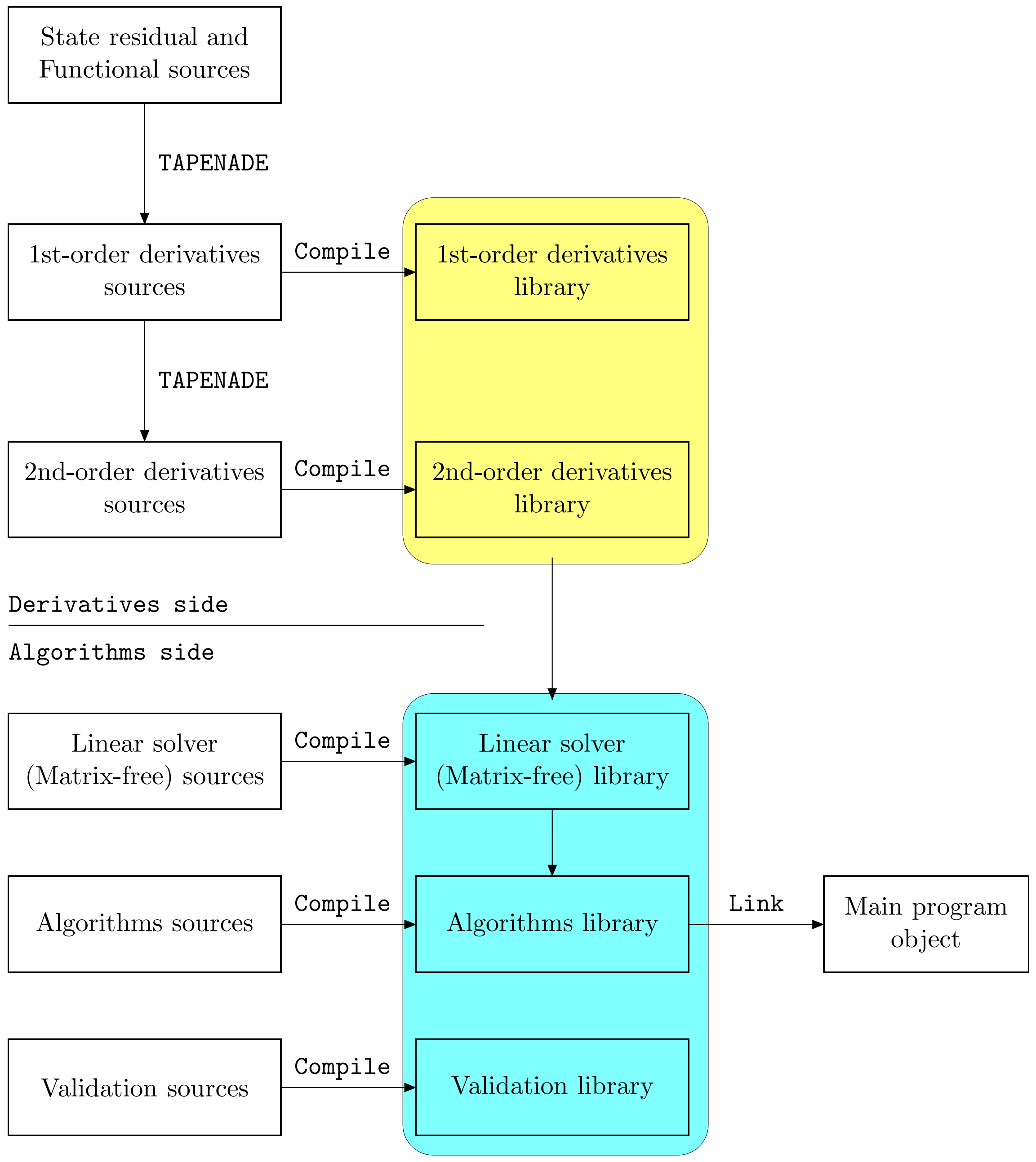

State residual and Functional sources
TAPENADE
1st-order derivatives sources
Compile
1st-order derivatives library
TAPENADE
2nd-order derivatives sources
Compile
2nd-order derivatives library
Derivatives side
Algorithms side
Linear solver (Matrix-free) sources
Compile
Linear solver (Matrix-free) library
Algorithms sources
Compile
Algorithms library
Link
Main program object
Validation sources
Compile
Validation library

Figure 3.11: Framework for the implementation of the first and second-order derivatives. For each new definition for functional or the state residual, only the upper part (Derivatives side) is re-evaluated: the algorithms side is unchanged. To do that, we are forced to use a fixed interface for the subroutines implementing the functional and the state residual.

3.9 TAPENADE commands

In this section we give the TAPENADE commands to perform the different differentations needed by the algorithms (ToT and ToR) presented in Sections 3.6.1-3.6.2. See the TAPENADE user's guide [Hascoët and Pascual, 2004] for extra details.

- `tapenade -d -root state_residuals -outvars "psi" -vars "gamma w" -difffuncname _d -fixinterface`

Result	`state_residuals_d(psi,psid,gamma,gammad,w,wd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{psid} = \dot{\Psi} = \dfrac{\partial \Psi}{\partial \gamma}\dot{\gamma} + \dfrac{\partial \Psi}{\partial W}\dot{W} \end{cases}$

- `tapenade -d -root state_residuals -outvars "psi" -vars "gamma" -difffuncname _dgamma_d -fixinterface`

Result	`state_residuals_dgamma_d(psi,psid,gamma,gammad,w)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{psid} = \dot{\Psi}_{\gamma} = \dfrac{\partial \Psi}{\partial \gamma}\dot{\gamma} \end{cases}$

- `tapenade -d -root state_residuals -outvars "psi" -vars "w" -difffuncname _dw_d -fixinterface`

Result	`state_residuals_dw_d(psi,psid,gamma,w,wd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{psid} = \dot{\Psi}_W = \dfrac{\partial \Psi}{\partial W}\dot{W} \end{cases}$

- `tapenade -b -root state_residuals -outvars "psi" -vars "gamma w" -difffuncname _b -fixinterface`

Result	`state_residuals_b(psi,psib,gamma,gammab,w)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{psib} = \bar{\Psi} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{gammab} = \bar{\gamma}_\Psi = \left(\dfrac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi} \\ \texttt{wb} = \bar{W}_\Psi = \left(\dfrac{\partial \Psi}{\partial W}\right)^T \bar{\Psi} \end{cases}$

- `tapenade -b -root state_residuals -outvars "psi" -vars "gamma" -difffuncname _dgamma_b -fixinterface`

Result	`state_residuals_dgamma_b(psi,psib,gamma,gammab,w,wb)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{psib} = \bar{\Psi} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{gammab} = \bar{\gamma}_\Psi = \left(\dfrac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi} \end{cases}$

- `tapenade -b -root state_residuals -outvars "psi" -vars "w" -difffuncname _dw_b -fixinterface`

Result	`state_residuals_dw_b(psi,psib,gamma,gammab,w)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{psib} = \bar{\Psi} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{wb} = \bar{W}_\Psi = \left(\dfrac{\partial \Psi}{\partial W}\right)^T \bar{\Psi} \end{cases}$

- `tapenade -d -root state_residual_d -outvars "psid" -vars "gamma w" -difffuncname _d -fixinterface`

Result	`state_residuals_d_d(psi,psid,psidd,gamma,gammad0,gammad,w,wd0,wd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{gammad0} = \dot{\gamma}_0 \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \\ \texttt{wd0} = \dot{W}_0 \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{psid} = \dot{\Psi} = \frac{\partial \Psi}{\partial \gamma}\dot{\gamma} + \frac{\partial \Psi}{\partial W}\dot{W} \\ \texttt{psidd} = \dot{\dot{\Psi}} = \frac{\partial}{\partial \gamma}\left(\frac{\partial \Psi}{\partial \gamma}\dot{\gamma}\right)\dot{\gamma_0} + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial \gamma}\dot{\gamma}\right)\dot{W_0} + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial \gamma}\dot{\gamma_0}\right)\dot{W} + \\ \qquad + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial W}\dot{W}\right)\dot{W_0} \end{cases}$

- `tapenade -d -root state_residual_b -outvars "gammab wb" -vars "gamma w" -difffuncname _d -ext PUSHPOPGeneralLib -extAD PUSHPOPADLib-fixinterface`

Result	`state_residuals_b_d(psi,psib,gamma,gammad,gammab,gammabd,w,wd,wb,wbd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{psib} = \bar{\Psi} \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{gammab} = \bar{\gamma}_\Psi = \left(\frac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi} \\ \texttt{wb} = \bar{W}_\Psi = \left(\frac{\partial \Psi}{\partial W}\right)^T \bar{\Psi} \\ \texttt{gammabd} = \dot{\bar{\gamma}}_\Psi = \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi}\right]\dot{W} \\ \texttt{wbd} = \dot{\bar{W}}_\Psi = \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T \bar{\Psi}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T \bar{\Psi}\right]\dot{W} \end{cases}$

- `tapenade -d -root functional -outvars "j" -vars "gamma w" -difffuncname _d -fixinterface`

Result	`functional_d(j,jd,gamma,gammad,w,wd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{j} = J \\ \texttt{jd} = \dot{J} = \dfrac{\partial J}{\partial \gamma}\dot{\gamma} + \dfrac{\partial J}{\partial W}\dot{W} \end{cases}$

- `tapenade -d -root functional -outvars "j" -vars "gamma" -difffuncname _dgamma_d -fixinterface`

Result	`functional_dgamma_d(j,jd,gamma,gammad,w)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \end{cases}$
Output	$\begin{cases} \texttt{j} = J \\ \texttt{jd} = \dot{J}_\gamma = \dfrac{\partial J}{\partial \gamma}\dot{\gamma} \end{cases}$

- `tapenade -d -root functional -outvars "j" -vars "w" -difffuncname _dw_d -fixinterface`

Result	`functional_dw_d(j,jd,gamma,w,wd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{j} = J \\ \texttt{jd} = \dot{J}_W = \dfrac{\partial J}{\partial W}\dot{W} \end{cases}$

- `tapenade -b -root functional -outvars "j" -vars "gamma w" -difffuncname _b -fixinterface`

Result	`functional_b(j,jb,gamma,gammab,w)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{jb} = \bar{J} \end{cases}$
Output	$\begin{cases} \texttt{j} = J \\ \texttt{gammab} = \bar{\gamma}_J = \left(\dfrac{\partial J}{\partial \gamma}\right)^T \bar{J} \\ \texttt{wb} = \bar{W}_J = \left(\dfrac{\partial J}{\partial W}\right)^T \bar{J} \end{cases}$

- `tapenade -b -root functional -outvars "j" -vars "gamma" -difffuncname _dgamma_b -fixinterface`

Result	`functional_dgamma_b(j,jb,gamma,gammab,w,wb)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{jb} = \bar{J} \end{cases}$
Output	$\begin{cases} \texttt{j} = J \\ \texttt{gammab} = \bar{\gamma}_J = \left(\dfrac{\partial J}{\partial \gamma}\right)^T \bar{J} \end{cases}$

- `tapenade -b -root functional -outvars "j" -vars "w" -difffuncname _dw_b -fixinterface`

Result	`functional_dw_b(j,jb,gamma,gammab,w)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{jb} = \bar{J} \end{cases}$
Output	$\begin{cases} \texttt{j} = J \\ \texttt{wb} = \bar{W}_J = \left(\dfrac{\partial J}{\partial W}\right)^T \bar{J} \end{cases}$

- `tapenade -d -root state_residual_d -outvars "jd" -vars "gamma w" -difffuncname _d -fixinterface`

Result	`functional_d_d(j,jd,jdd,gamma,gammad0,gammad,w,wd0,wd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{gammad0} = \dot{\gamma}_0 \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \\ \texttt{wd0} = \dot{W}_0 \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{j} = J \\ \texttt{jd} = \dot{J} = \dfrac{\partial J}{\partial \gamma}\dot{\gamma} + \dfrac{\partial J}{\partial W}\dot{W} \\ \texttt{jdd} = \ddot{J} = \dfrac{\partial}{\partial \gamma}\left(\dfrac{\partial J}{\partial \gamma}\dot{\gamma}\right)\dot{\gamma}_0 + \dfrac{\partial}{\partial W}\left(\dfrac{\partial J}{\partial \gamma}\dot{\gamma}\right)\dot{W}_0 + \dfrac{\partial}{\partial W}\left(\dfrac{\partial J}{\partial \gamma}\dot{\gamma}_0\right)\dot{W} + \\ \qquad + \dfrac{\partial}{\partial W}\left(\dfrac{\partial J}{\partial W}\dot{W}\right)\dot{W}_0 \end{cases}$

- `tapenade -d -root state_residual_b -outvars "gammab wb" -vars "gamma w" -difffuncname _d -ext PUSHPOPGeneralLib -extAD PUSHPOPADLib-fixinterface`

Result	`functional_b_d(j,jb,gamma,gammad,gammab,gammabd,w,wd,wb,wbd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{jb} = \bar{J} \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{j} = J \\ \texttt{gammab} = \bar{\gamma}_J = \left(\frac{\partial J}{\partial \gamma}\right)^T \bar{J} \\ \texttt{wb} = \bar{W}_J = \left(\frac{\partial J}{\partial W}\right)^T \bar{J} \\ \texttt{gammabd} = \dot{\bar{\gamma}}_J = \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial J}{\partial \gamma}\right)^T \bar{J}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial J}{\partial \gamma}\right)^T \bar{J}\right]\dot{W} \\ \texttt{wbd} = \dot{\bar{W}}_J = \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial J}{\partial W}\right)^T \bar{J}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial J}{\partial W}\right)^T \bar{J}\right]\dot{W} \end{cases}$

3.10 Conclusion

In this chapter we have studied two approaches for computing second-order derivatives of a constrained functional, in which the constraint is assumed to be solved with a fixed-point method. Both approaches rely on the knowledge and availability of the adjoint state Π_h, solution of the adjoint linear system (3.18).

The first approach (Tangent-on-Tangent, [Ghate and Giles, 2007] and Section 3.6.1) permits us to have a single element of the Hessian matrix at time, and its runtime cost is due to two main contributions: the solution of n linear systems (in order to compute the derivative of the state respect to the contol) and the evaluation of $\frac{n(n+1)}{2}$ double-differentiated routines. These facts result in an appealing strategy for problems requiring the full Hessian (or a part of it, like the diagonal) when the dimension of the control space is moderate ($\lesssim 10^3$). The double differentiation required by this approach could be performed in automatic way using AD tools and its implementation is straightforward, due to the fact that in each differentation the computational graph and the program structure remain frozen: any `PUSH/POP` pair is added to the code, resulting in a very robust (and easy to implement) approach to compute the Hessian matrix.

The second approach (Tangent-on-Reverse, Section 3.6.2) builds the Hessian matrix using the multiplication of the Hessian matrix by a vector, and the runtime cost of this Hessian-by-vector multiplication is mainly due to the solution of two linear systems (3.33). Moreover, the runtime cost to obtain the Hessian-by-vector multiplication is independent from the dimension n of the control space. In order to obtain the full Hessian matrix, we need to apply the Hessian-by-vector multiplication to each element of the canonical basis, resulting in a cost that is proportional to the dimension n. The fact that ToR builds the Hessian column-by-column (or, by symmetry, row-by-row) results in the impossibility to obtain separately the diagonal part of the Hessian. Due to the requirement of the solution of an extra linear system for each column, the ToR approach to compute the full Hessian is cheaper than ToT when the dimension of the control space is high ($\gtrsim 10^3$). This approach raises some new AD issues because of the special structure of Reverse-differentiated programs, and the correct Tangent-differentiation of such programs is not perfectly performed yet by TAPENADE (the user needs to make some little adjustments on the ToR-differentiated code).

Exploiting the characteristics of the two approaches above, to obtain the full Hessian we can build a mixed strategy that uses the ToT and ToR strategies to compute different parts of the Hessian matrix. This approach results in a cheaper strategy not only respect to the pure ToR approach, but even for medium-size problems for which the single ToT strategy is preferred respect to ToR.

Using a clever strategy for the definition of the programs interfaces, we can use a separate compilation and late-linking approach, resulting in a easy and rapid implementation of the algorithms above for existing industrial and research codes.

Chapter 4

Multilevel optimization in aerodynamic shape design

4.1 Introduction to multilevel approaches in aerodynamic shape design

Due to the ripeness reached by computational fluid dynamics (CFD) combined with the rapid advances of computational power, research in the field of aerodynamic shape design has experienced a large development in the last years, allowing to deal with more and more complex optimization problems. However, shape optimization for aerodynamic applications remains a costly task since the system of governing flow equations (as, for instance, Euler or Navier-Stokes equations) should be solved, many times during the whole procedure. Thus, even if significant progress have been done for optimization tools and related techniques (as, for instance, the use of adjoint approaches in the context of gradient-based methods), the improvement of optimization algorithm efficiency still appears as an important goal.

On another hand, to deal with complex engineering design optimization, different multi-level or multi-scale approaches, in which the whole problem is decomposed in several simpler sub-problems to be solved in a predetermined sequence, have been developed (see e.g. [Migdalas et al., 1997; Schwabacher and Gelsey, 1998]). Each optimization sub-problem can differ according to objective function, constraints, design space and/or optimization algorithm allowing a better treatment of complex systems (multidisciplinary design, multiple local optima, large-scale system, multi-objective optimization, ...). Alternatively, efficiency can be also increased using various degrees of fidelity, i.e. varying the complexity of the physical modelling and/or the accuracy of the numerical approach (see e.g. [Alexandrov et al., 2001]). Note that a particular case of low fidelity model can be obtained through the use of coarse meshes, in which the flow solution is computed more easily and at a lower cost. For instance, in the field of aerodynamic design optimization, in [Feng and Pulliam, 1995], a reduced Hessian SQP algorithm is combined with a solution refinement, while in [Dadone and Grossman, 2000], a progressive optimization is proposed in which starting from a low accurate computation of the sensitivity derivatives (using coarse mesh and partially converged flow solutions) the degree of accuracy is progressively increased during the optimization process. A similar idea is proposed in [Pironneau and Polak, 2002] (see also Chapter 6 of [Mohammadi and Pironneau, 2001]), in which mesh refinement is combined with approximate gradients in order to speed up the convergence on the finest mesh of the descent algorithm. Methods based on multigrid principles can be the successive step; some works can be found in literature in which multigrid-like techniques have been applied to optimal control problems involving partial differential equations. For instance, the MG/Opt algorithm [Nash, 2000] recursively uses coarse resolution problems (coarse mesh) to generate search directions at a cheaper cost for finer resolution problems. For further examples, we refer, for instance, to [Gelman and Mandel, 1990; Dreyer et al., 2000; Borzi, 2003; Gratton et al., 2004]. In the context of aerodynamic design optimization, in [Kuruvila et al., 1994] a one-shot method, in which the flow and the sensitivities are simultaneously solved, is coupled with a multigrid approach. In this formulation, the design variables corresponding to low-frequencies of the shape are updated on a coarse level (i.e., a coarse mesh) while the other design variables are updated on a finer level. Finally, in [Catalano et al., 2005, 2008], the progressive optimization, introduced in [Dadone and Grossman, 2000], is coupled with a multigrid-aided finite-difference approach, in which the gradient is obtained through finite-difference sensitivities computed using only flow solutions on a coarse mesh.

Another approach, also based on multilevel concepts and ideated for gradient-like methods, has been introduced in [Beux and Dervieux, 1994]. In this preconditioned gradient method, the minimisation is done alternatively on different subsets of control parameters according to multigrid-like cycles. More particularly, using shape grid-point coordinates as design variables, a hierarchical parametrization was defined considering different subsets of parameters extracted from the complete parameterisation, which can be prolongated to the higher level by linear mapping. This approach acts as a smoother and also makes the convergence rate of the gradient-based method low dependent of the number of control parameters. The good behaviour observed in different numerical experiments [Beux, 1994; Beux and Dervieux, 1994], have been also corroborated by a theoretical view point in [Guillard, 1993; Guillard and Marco, 1995]. Note that, contrary to the other approaches based on multigrid concepts, only one computational mesh is employed since the coarseness acts only on the number of design parameters. The increase of efficiency is only related to the faster convergence obtained considering less degrees of freedom and to the improvement on convergence rate typical of multigrid techniques. Different extensions of the original approach have been, successively, proposed: in [Held et al., 2002] the same hierarchical parametrization is associated with a finite-difference/one-shot formulation, in [Marco and Dervieux, 1999] the generalisation to 3D case involving unstructured meshes is done through the use of agglomeration technique while an additive multilevel preconditioner has been also defined in [Koobus et al., 1997; Courty and Dervieux, 2006].

Another multilevel approach based on a family of embedded parametrizations has also been proposed in [Désidéri, 2003] (see also successive works, as e.g. [Abou El Majd et al.; Désidéri, 2007; Abou El Majd et al., 2008]). However, contrary to the method introduced in [Beux and Dervieux, 1994] and its different extensions, this approach is based on a polynomial representation of the shape through the use of Bézier curves, and is not specifically focused on gradient-based methods.

4.2 Optimum shape design problem in aerodynamics

4.2.1 The Optimal shape problem in a fully discrete context

The optimal shape problem consists in minimising a cost functional j with respect to some control variables α, which should characterise the shape. Moreover, for aerodynamic shape optimization, j can not be expressed directly in a explicit way as a function of α since it also depends on the flow variables. Indeed, for each shape configuration, and thus, for each choice of α, a particular flow is obtained by solving the governing equations, i.e typically Euler or Navier-Stokes equations. Note that we consider an optimization in a discrete context in which the optimization algorithm is applied to the problem already fully discretised, i.e. with both the discrete governing equations and the discrete cost functional. Then, the unconstrained optimal shape problem can be written as follows:

$$\text{Find } \ \alpha_{opt} \in \mathbb{R}^p \text{ such that } \ j(\alpha_{opt}) = \min_{\alpha \in \mathbb{R}^p} j(\alpha) \tag{4.1}$$

in which the cost functional j can be defined introducing $J : \mathbb{R}^p \times \mathbb{R}^N \to \mathbb{R}$ such that:

$$\forall \alpha \in \mathbb{R}^p \quad j(\alpha) = J(\alpha, W(\alpha)) \tag{4.2}$$

where α is a discrete set of parameters (see Sec. 4.2.3) while W represents the values of the flow variables at each point of the computational mesh.

Furthermore, the discretised shape is fully determined by the coordinates of the grid-points localised on the shape. Thus, the control variables influence the discrete cost functional only through these coordinates, which can be introduced as intermediate variables. More precisely, let us consider $\mathcal{L} : \mathbb{R}^p \to \mathbb{R}^q$, the operator which, for each set of control parameters, furnishes the corresponding set of shape grid-points coordinates. Then, instead of (4.2) the cost functional can be expressed as follows:

$$\forall \alpha \in \mathbb{R}^p \quad j(\alpha) = \mathcal{I}(\mathcal{L}(\alpha)) = I\big(\mathcal{L}(\alpha), W(\mathcal{L}(\alpha))\big) \tag{4.3}$$

in which $\mathcal{I} : \mathbb{R}^q \to \mathbb{R}$ and $I : \mathbb{R}^q \times \mathbb{R}^N \to \mathbb{R}$ are defined by $j = \mathcal{I} \circ \mathcal{L}$ and $J = I \circ \mathcal{L}$ respectively.

4.2.2 Computation of the sensitivity derivatives

In the present study, an exact hand-coding discrete adjoint approach is used for a 2D-Euler stationary flow solver based on an unstructured finite-volume first-order spatial discretisation and a pseudo-unsteady approach associated with a linearised implicit algorithm. Finite-difference sensitivities through the formulation proposed in [Held et al., 2002] are also considered in Sec. 4.6.

4.2.3 Parametrisations for aerodynamic shape representation

An important ingredient which should be also specified in the optimization process is the representation of the shape, which is defined through the choice of the control parameters. Indeed, the shape parametrization plays a crucial role for the shape optimization since it directly acts on the accuracy of the final solution and on the efficiency of the particular optimization strategy.

The use of shape grid-point coordinates as design variables appears the more natural approach since, in this case, the parameterisation is directly correlated with the explicit representation of the discrete shape. Furthermore, for optimization algorithms based on an exact discrete gradient, the computation of the sensitivity derivatives are simplified since, in this case, (4.2) and (4.3) coincide. Nevertheless, since the geometry is modified by moving individual grid points, non-smooth profiles are often obtained, particularly during intermediate phases of the convergence to optimum (see, e.g. [Beux and Dervieux, 1992]). Moreover, the large number of variables involved in this case has a negative effect on the computational cost (slow convergence and, possibly, large number of cost functional evaluations). As a matter of fact, the multilevel strategy introduced in [Beux and Dervieux, 1994] and described in Sec. 4.3.2 was defined exactly in order to reduce these drawbacks. Note that the lack of shape smoothness, particularly critical for shape grid-points parametrization, can be also linked with a regularity loss of the gradient with respect to the control variables, already verified in the continuous case (see, e.g. [Courty and Dervieux, 2006]). To avoid oscillations, in many works dealing with a shape grid-point parameterisation, a smoothing is applied (see e.g. [Mohammadi and Pironneau, 2001; Reuther and Jameson, 1995]).

On the other hand, a classical approach for the parameterisation of the shape is to use a polynomial representation which permits a compact description of the shape with only few parameters. For instance, Bézier control points, i.e. coefficients in a basis of Bernstein polynomials, can be used as design variables. The Bézier representation has suitable properties at an algorithmic level (efficient recursive algorithms) but also is well adapted to deal with geometric constraints (see, e.g. [Farin, 1990]). Furthermore, the Bézier curves act as a basic tool to define other representations as B-splines and non-uniform rational B-spline (NURBS) more suitable for high-degree polynomial and non smooth geometries respectively.

Another possible choice, frequently considered in the context aerodynamic shape design (typically, for airfoil or wing design), is to represent the shape through a linear combination of given geometric shapes. The control variables are, then, the coefficients in this basis of shape functions. In this case, few parameters are sufficient to obtain a good shape representation, but, on the other hand, the final solution is highly related to the choice of the particular basis. Thus, this representation yields a priori a smaller design space with respect to the parametrizations based on shape grid-points or polynomial control points. The basis is, in general, composed of existing geometric shapes or alternatively, a given base shape and a set of modified shapes obtained from the first one through some perturbation functions, such as the Hicks-Henne analytical functions (see [Hicks and Henne, 1978]). Moreover, in order to improve the completeness of the design space and thus, avoid the presence of nearly linear dependent functions, some authors use orthogonal functions obtained through a Gram-Schmidt orthogonalisation (see, [Kuruvila et al., 1994; Chang et al., 1995; Catalano et al., 2005]) or analytically (e.g., through the use of Chebychev polynomials in [Carpentieri et al., 2007]).

The three kinds of parametrizations, described here, represent a large range of widely used approaches, and, have been considered in this study in the framework of multilevel methods. For a more complete overview of possible shape parametrizations, we refer, as example, to [Samareh, 2001] or [Selmin, 2008].

4.3 Multilevel gradient-based approaches for shape design

4.3.1 Change of Hilbert control space

Multi-level methods in the context of optimum shape design, as initially proposed in [Beux and Dervieux, 1994], are based on a change of control space. More precisely, let us consider the optimization of a differentiable functional $j : U \to \mathbb{R}$ in a Hilbert space U. Then, instead of a direct minimisation of j in U, one can also envisage a minimisation of j in the subset $f(V) \subset U$, in which V is a second Hilbert space and f an application from V to U. It can be formulated, equivalently, as the minimisation of $\mathcal{J} = j \circ f$ in V, i.e.:

$$\text{Find} \ \ \alpha_{opt} \in V \text{ such that } \ \mathcal{J}(\alpha_{opt}) = \min_{\alpha \in V} [j \circ f](\alpha) \tag{4.4}$$

The Fréchet derivative of $\mathcal{J}$ at $\alpha \in V$ can be expressed as follows:

$$\forall h \in V \quad \mathcal{J}'(\alpha)(h) = [j \circ f]'(\alpha)(h) = j'\left(f(\alpha)\right)\big(f'(\alpha)(h)\big)$$

Since $f'(\alpha) \in \mathcal{L}(V, U)$, i.e is a linear continuous application from V to U, the following relation can be also obtained in terms of gradient for any $h \in V$:

$$\begin{aligned} \langle \text{grad}_V \mathcal{J}(\alpha), h \rangle_V &= \langle \text{grad}_U j(\gamma), f'(\alpha)h \rangle_U \\ &= \langle \big(f'(\alpha)\big)^* \, \text{grad}_U j(\gamma), h \rangle_V \end{aligned} \tag{4.5}$$

where $\gamma = f(\alpha)$, $\big(f'(\alpha)\big)^* \in \mathcal{L}(U, V)$ is the adjoint of $f'(\alpha)$ and $\langle ., . \rangle_U$ and $\langle ., . \rangle_V$ are the inner products associated to U and V respectively.

Furthermore, let us consider the particular case in which f affine, and thus, it exists $b \in U$ and $P \in \mathcal{L}(V, U)$ such that $f : \alpha \to P\alpha + b$. Then, since in this case $f'(\alpha) = P$ for any $\alpha \in V$, solving the minimisation problem (4.4) through a gradient descent method corresponds to the following iterative algorithm:

$$\alpha_0 \in V \text{ given}, \qquad \text{for } r \geq 0 \quad \alpha_{r+1} = \alpha_r - \omega_r P^* \text{grad}_U \, j(f(\alpha_r))$$

Nevertheless, applying the operator f permits to go back to space U, and thus, to obtain:

$$f(\alpha_{r+1}) = P\alpha_{r+1} + b = f(\alpha_r) - \omega_r P P^* \text{grad}_U \, j(f(\alpha_r))$$

Thus, considering as initial solution $\gamma_0 = f(\alpha_0)$, the following iterative algorithm is finally defined in U:

$$\text{for } r \geq 0 \quad \gamma_{r+1} = \gamma_r - \omega_r P P^* \text{grad}_U \, j(\gamma_r) \tag{4.6}$$

Lemma 4.1. *Let $\alpha \in V$. If $grad_V \mathcal{J}(\alpha) \neq 0$ then $\exists \bar{\omega} \in \mathbb{R}^+$ such that for $0 < \omega < \bar{\omega}$ holds $\mathcal{J}(\alpha - \omega \, grad_V \mathcal{J}(\alpha)) < \mathcal{J}(\alpha)$. If $grad_V \mathcal{J}(\alpha) = 0$ then $\mathcal{J}(\alpha - \omega \, grad_V \mathcal{J}(\alpha)) = \mathcal{J}(\alpha)$.*

Proof. Let $\text{grad}_V \mathcal{J}(\alpha) \neq 0$. Using the Taylor expansion of $\mathcal{J}(\alpha - \omega \text{grad}_V \mathcal{J})$ around α we obtain

$$\begin{aligned} \mathcal{J}(\alpha - \omega \text{grad}_V \mathcal{J}(\alpha)) &= \mathcal{J}(\alpha) - \omega \left\langle \text{grad}_V \mathcal{J}(\alpha), \text{grad}_V \mathcal{J}(\alpha) \right\rangle_V + o\big(\omega ||\text{grad}_V \mathcal{J}(\alpha)||\big) \\ &= \mathcal{J}(\alpha) - \omega ||\text{grad}_V \mathcal{J}(\alpha)||_V^2 + o\big(\omega ||\text{grad}_V \mathcal{J}(\alpha)||\big) \end{aligned}$$

by the hypotesis on the gradient we have $||\text{grad}_V \mathcal{J}(\alpha)||_V^2 \neq 0$, and therefore, if we choose $\omega > 0$ small enough we have $\mathcal{J}(\alpha - \omega\, \text{grad}_V \mathcal{J}(\alpha)) < \mathcal{J}(\alpha)$. The second part of the Lemma is obvious. □

Theorem 1. *If $\alpha \in V$, $b \in U$, $P \in \mathcal{L}(V,U)$ and $\gamma = P\alpha + b$ then*

$$\exists\, \bar{\omega} \in \mathbb{R}^+ \textit{ such that for } 0 < \omega < \bar{\omega} \textit{ holds } j(\gamma - \omega P P^* \textit{grad}_U j(\gamma))) \leq j(\gamma).$$

Proof. Remembering that $\mathcal{J}(\alpha) = [j \circ f](\alpha)$ and $f\colon \alpha \to P\alpha + b$ (where P is a linear operator), we have

$$\begin{aligned}\mathcal{J}(\alpha - \omega \text{grad}_V \mathcal{J}(\alpha)) &= [j \circ f](\alpha - \omega \text{grad}_V \mathcal{J}(\alpha)) \\ &= j\big(P(\alpha - \omega\, \text{grad}_V \mathcal{J}(\alpha)) + b\big) \\ &= j\big(P\alpha + b - \omega P\, \text{grad}_V \mathcal{J}(\alpha)\big) \\ &= j\big(\gamma - \omega P\, \text{grad}_V \mathcal{J}(\alpha)\big) \\ &= j\big(\gamma - \omega P P^*\, \text{grad}_U j(\gamma)\big)\end{aligned}$$

where for the last term we used the relation (4.5), namely $\text{grad}_V \mathcal{J}(\alpha) = P^*\, \text{grad}_U j(\gamma)$. Using Lemma 4.1 and remembering the equivalence of $\mathcal{J}(\alpha) = j(\gamma)$ the theorem is proved. □

Remark 4.1. Theorem 1 states that $-P\, \text{grad}_V \mathcal{J}(\alpha) = -PP^*\, \text{grad}_U j(\gamma)$ is a *weak descent direction* for $j(\gamma)$, that is $j(\gamma - \omega PP^*\, \text{grad} j(\gamma)) \leq j(\gamma)$ because, without further requirements for P other than linearity and boundedness, there could be $\bar{\gamma}$ such that $P^* \text{grad}_U j(\bar{\gamma}) = 0$.

For Theorem 1 the algorithm in (4.6) is a weak descent method in U, and can be also interpreted as a preconditioned gradient method.

Furthermore, since we are interested to the shape optimization problem (4.1) in a discrete context, $U = \mathbb{R}^p$ while the second Hilbert space is, typically, $V = \mathbb{R}^{\bar{p}}$ with $\bar{p} < p$. Then, the linear operator P and its adjoint P^* are associated to a matrix $M \in \mathbb{R}^{p \times \bar{p}}$ and its transpose respectively, and thus, the algorithm can be simply rewritten as:

$$\text{for } r \geq 0 \qquad \gamma_{r+1} = \gamma_r - \omega_r\, MM^T g_r \tag{4.7}$$

with $g_r = \text{grad}\, j(\gamma_r) \in \mathbb{R}^p$.

Note that MM^T, the matrix which is used as preconditioner for the gradient, is, by construction, a square symmetrical positive semidefinite matrix. This implies that $MM^T g_r$ is a weak descent direction; a result already ensured by Theorem 1 in a more general context.

4.3.2 A hierarchical parametrization based on shape grid-points

An optimization algorithm based on control space change as presented in Sec. 4.3.1 has been initially proposed in [Beux and Dervieux, 1994] for the case of a linear operator, i.e. for $f = P$. In this study, the ordinates of the grid-points, localised on the shape which should be optimised, have been chosen as control variables γ. Then, a set of points, extracted from the complete set of shape grid-points, is considered as sub-parametrization α while a Hermitian cubic interpolation is defined as prolongation operator f.

Moreover, instead of considering a single space V, the cost functional is minimised alternatively on different control subspaces of decreasing dimension. More precisely, a family of embedded sub-parametrizations is considered, in which for each increase of level the number of points is doubled. At a particular level, the prolongation operator is defined by $P^{(l)} = P_{L-1}^{L} \circ \cdots \circ P_{l}^{l+1}$ where L corresponds to the finest level, i.e. to the complete parametrization, while P_i^{i+1} is the cubic interpolation used for the prolongation from level i to the next one. In practice, at each optimization iteration r corresponds a particular level l, and following (4.6), minimising on this coarse level l corresponds to replace the gradient g_r by the descent direction $p_r^{(l)} = P^{(l)}(P^{(l)})^* g_r$. The choice of the particular subspace is determinate by a strategy of level changes similar to multilevel/multigrid strategies used for the resolution of partial differential equations (as, for instance, V-cycles).

Note that the multilevel approach elaborated in [Beux and Dervieux, 1994] is strongly linked to the particular type of parametrization used. Nevertheless, the formulation described in Sec. 4.3.1 is rather more general since, providing that the prolongation operator be affine, any type of design variables and sub-parametrizations can be proposed.

4.3.3 Generalisation to other kinds of parametrization

The multi-level method, as described in the previous sections, is based on the idea of control space change. Moreover, it also need the definition of a family of sub-parametrizations and the corresponding prolongation operators. In Sec. 4.3.2, a set of sub-parametrizations has been found in a natural way considering shape grid-points as control parameters.

More generally, let us, now, consider an optimal shape design associated to α, a generic set of control variables. Nevertheless, as pointed out in Sec. 4.2, the shape grid-points coordinates can be used as intermediate variables. Then, since the cost functional can be expressed as $j = \mathcal{I} \circ \mathcal{L}$, the following relation between the gradients of j and $\mathcal{I}$ can be obtained in a similar way as done to obtain (4.5):

$$\forall \alpha \in \mathbb{R}^p \quad \text{grad}_\alpha j(\alpha) = \left(\mathcal{L}'(\alpha)\right)^* \text{grad}_\gamma \mathcal{I}\left(\mathcal{L}(\alpha)\right)$$

where the subscript α and γ denote, here, that the gradient is in $\mathbb{R}^p$ and $\mathbb{R}^q$ respectively. Thus, the gradient descent method for the minimisation of j with respect to α corresponds to the following iterative algorithm:

$$\alpha_0 \in \mathbb{R}^p \text{ given,} \qquad \text{for } r \geq 0 \quad \alpha_{r+1} = \alpha_r - \omega_r \left(\mathcal{L}'(\alpha_r)\right)^* \text{grad}_\gamma \mathcal{I}(\mathcal{L}(\alpha_r))$$

After the computation of α_{r+1}, the shape grid-points coordinates should be also updated. This updating is done through $\mathcal{L}$, which lies the control variables to the shape grid-points coordinates γ, in the following way:

$$\text{for } r \geq 0, \quad \gamma_{r+1} = \mathcal{L}(\alpha_{r+1}) = \mathcal{L}\Big(\alpha_r - \omega_r \left(\mathcal{L}'(\alpha_r)\right)^* \text{grad}_\gamma \mathcal{I}(\gamma_r)\Big) \tag{4.8}$$

with $\gamma_0 = \mathcal{L}(\alpha_0)$.

Note that, if the shape parametrization is relied to the coordinates of the shape grid-points by an affine application, and thus, in particular, it exists a matrix $M \in \mathbb{R}^{q \times p}$ such that for any

$h \in \mathbb{R}^p$ we have $\left[\mathcal{L}'(\alpha_r)\right](h) = Mh$, then (4.8) can be rewritten as follows:

$$\gamma_{r+1} = \gamma_r - \omega_r M M^T \text{grad}_\gamma \mathcal{I}(\gamma_r)$$

In this case, a descent direction is obtained considering as control variables α as well as γ. From the point of view of an optimization with respect to γ, α acts as a sub-parametrization, and, we exactly recover the formulation of Sec. 4.3.1 taking, here, $f = \mathcal{L}$, $j = \mathcal{I}$ and $\mathcal{J} = j$. Thus, the approach proposed in [Beux and Dervieux, 1994] corresponds to choose as control parameters the coordinates of a subset of the shape grid-points.

On another hand, if it is possible to find an affine application from $\mathbb{R}^{\bar{p}}$ to $\mathbb{R}^p$ (associated to a matrix $D \in \mathbb{R}^{p \times \bar{p}}$), then, as done in Sec. 4.3.1, a preconditioned gradient method can be also defined for the optimization of j with respect to α. It corresponds to the following iterative algorithm:

$$\alpha_0 \in \mathbb{R}^p \text{ given}, \qquad \text{for } r \geq 0 \quad \alpha_{r+1} = \alpha_r - \omega_r D D^T \text{grad}_\alpha j(\alpha_r) \tag{4.9}$$

Thus, two possible ways can be envisaged to generalize the multi-level approach: considering α as control variables as done in (4.9), or alternatively, considering γ as control variables if $\mathcal{L}$ is affine. In this last case, one can also combined the two kinds of preconditioners. Indeed, (4.9) can be expressed in terms of shape grid-points as follows:

$$\gamma_{r+1} = \mathcal{L}\Big(\alpha_r - \omega_r D D^T \left(\mathcal{L}'(\alpha_r)\right)^* \text{grad}_\gamma \mathcal{I}(\gamma_r)\Big)$$

Then, if $\mathcal{L}$ is an affine application, it can be also rewritten as:

$$\gamma_{r+1} = \gamma_r - \omega_r M D (M D)^T \text{grad}_\gamma \mathcal{I}(\gamma_r) \tag{4.10}$$

In this case, the same algorithm can be interpreted as a preconditioned gradient method considering as control variables α (through (4.9)) as well as γ (through (4.10)). In particular, let us consider a parametrization α with $\mathcal{L}$ affine and an adequate set of sub-levels associated to the matrices $D^{(l)}$, $l = 1, \cdots, L$. Then, (4.10) furnishes a practical way to define the different preconditioning matrices for the optimization with respect to γ taking $M^{(l)} = M D^{(l)}$.

Note that, in order to really define a multi-level strategy, one should also introduce a notion of coarseness for the sub-levels. In particular, we should be able to define a family of sub-parametrizations with a decreasing number of parameters. Furthermore, a coarser level should correspond to a representation of the shape with lower frequencies. Finally, it will be also interesting to define, as done for the original formulation [Beux and Dervieux, 1994], a family of embedded parametrizations in which the passage from a given level to the finest one, can be done progressively considering successively the different intermediate sub-levels.

4.4 Examples of alternative multi-level approaches

In the framework of the formulation defined in Sec. 4.3.3, we propose, here, to elaborate alternative multi-level approaches based on some classical shape parametrizations. More precisely, parameters, which can be related by linear or affine application to the set of shape grid points, are individualized, and then, used to define an adequate family of sub-parametrizations. Note that, as done in [Beux and Dervieux, 1994], we consider the case in which the design control acts only on the ordinates of the shape grid points whereas the abscissas $x_0^\Gamma, \cdots, x_m^\Gamma$ are defined by the knowledge of the initial mesh and frozen during the optimization process. In the context of aerodynamic shape optimization, this approach is often chosen, and does not appear so restrictive as long as only slender bodies are considered.

4.4.1 Formulation based on Bézier control points

Shape representation through Bézier curves

A Bézier curve of degree n can be defined as follows:

$$S(t) = (x(t), y(t))^T = \sum_{q=0}^{n} B_q^n(t) S_q \quad \text{with} \;\; t \in [0,1] \tag{4.11}$$

where $S_q = (\mathrm{x}_q, \mathrm{y}_q)^T$ is the q-th Bézier control point while $B_q^n(t)$ corresponds to the q-th Bernstein polynomial of degree n.

Thus, for a given set of parameters $(t_k)_{k=0,m}$ with $t_0 = 0 < t_1 < \cdots < t_m = 1$, the ordinates of the Bézier control points are directly related through a linear application with the ordinates of the shape grid-points, $y_0^\Gamma, \cdots, y_m^\Gamma$. Indeed, applying (4.11) with $y(t_k) = y_k^\Gamma$, we obtain:

$$\begin{cases} y_0^\Gamma = y(0) = \mathrm{y}_0 \\ y_k^\Gamma = y(t_k) = \displaystyle\sum_{q=1}^{n-1} B_q^n(t_k)\mathrm{y}_q + s_n(t_k) \quad \text{for } k = 1, m-1 \\ y_m^\Gamma = y(1) = \mathrm{y}_n \end{cases} \tag{4.12}$$

in which $s_n(t_k) = B_0^n(t_k)\mathrm{y}_0 + B_n^n(t_k)\mathrm{y}_n$.

Nevertheless, geometrical constraints are often imposed at the extremities of the shape which should be optimised. For instance, the shape extremities are in general fixed; it corresponds, here, to freeze y_0^Γ and y_m^Γ. Then, $\alpha = \left(\mathrm{y}_1, \cdots, \mathrm{y}_{n-1}\right)^T$ are related to $\gamma = (y_1^\Gamma, \cdots, y_{m-1}^\Gamma)^T$ by the following affine operator [1]:

$$\begin{array}{rlcll} f: & \mathbb{R}^{n-1} & \longrightarrow & \mathbb{R}^{m-1} & \\ & \alpha & \longmapsto & \gamma = M\alpha + b & b \in \mathbb{R}^{m-1}, M \in \mathbb{R}^{m-1 \times n-1} \end{array}$$

where

$$\begin{cases} M_{kq} = B_q^n(t_k), \quad \text{for } q = 1, n-1 \;\; \text{and} \;\; k = 1, m-1 \\ \\ b_k = s_n(t_k) = (1-t_k)^n y_m^\Gamma + (t_k)^n y_0^\Gamma, \quad \text{for } k = 1, m-1 \end{cases}$$

Thus, according to Sec. 4.3.1, it can be envisaged to define a strategy similar to the original multilevel approach [Beux and Dervieux, 1994]. Indeed, if the ordinates of the shape grid-points

[1]note that we choose, here, to follow the notations introduced in Sec. 4.3.1 of instead of the ones used in Sec. 4.2.1 and 4.3.3, i.e. to call the affine operator f instead of $\mathcal{L}$.

are taken as control variables, each sub-parametrization is chosen as a set of Bézier control points instead of a subset of boundary grid-points. Then, the following descent direction is obtained at iteration r and level l:

$$d_r^{(l)} = M^{(l)}(M^{(l)})^T\, g_r \quad \text{with} \quad M_{ij}^{(l)} = B_j^{n_l}(t_i^{(l)}). \tag{4.13}$$

To fully describe the different sub-parametrizations, at each sub-level $X^{(l)} = (x_0^{(l)}, \cdots, x_{n_l}^{(l)})^T$ with $n_l > n_{l-1}$ and $T^{(l)} = (t_0^{(l)}, \cdots, t_m^{(l)})^T$ should be defined. Furthermore, this definition should be done consistently with (4.11), i.e.:

$$x_k^\Gamma = x(t_k^{(l)}) = \sum_{q=0}^{n_l} B_q^{n_l}(t_k^{(l)})\mathrm{x}_q^{(l)} \quad \text{for} \quad k = 0, \cdots, m \tag{4.14}$$

in which the abscissas of shape grid-points $(\{x_k^\Gamma\}_{k=0,m})$ are given.

Note that the definition of the different sub-parametrizations is not dependent of the particular optimization iteration r, and thus, should be done as a preprocessing.

Sub-parametrizations based on the degree-elevation property

In order to define an adequate parametrization at level l, the classical degree-elevation property of the Bézier curves (see, e.g. [Farin, 1990])), which allows to increase the degree and the number of control points, is used here. Note that the good features of the degree-elevation property have been pointed out, and, already, employed to construct an embedded parametrization in [Désidéri, 2003] .

Given a Bézier curve of degree s associated to the $s+1$ control points $S_q = (\mathrm{x}_q, \mathrm{y}_q)^T$, the same geometrical curve can be also understood as a Bézier curve of degree $s+1$ considering a new set of $s+2$ control points $\bar{S}_q = (\bar{\mathrm{x}}_q, \bar{\mathrm{y}}_q)^T$ obtained from S_q as follows:

$$\bar{S}_q = \frac{q}{s+1} S_{q-1} + \left(1 - \frac{q}{s+1}\right) S_q \quad \text{for } q = 0, \cdots, s+1 \tag{4.15}$$

An interesting feature is that the distribution of the parameters t over the Bézier curve does not change by degree elevation, and thus:

$$\sum_{q=0}^{s} B_q^s(t)\, \mathrm{x}_q = \sum_{q=0}^{s+1} B_q^{s+1}(t)\, \bar{\mathrm{x}}_q \qquad \forall t \in [0,1] \tag{4.16}$$

Consequently, if the parametrization on the coarsest level has been yet defined with $X^{(0)}$ and $T^{(0)}$ consistent, i.e. with (4.14) verified for $l = 0$, then, thanks to (4.16), keeping the parameters t_k unchanged on all the levels, i.e. $T^{(l)} = T^{(0)}$ for all $l > 0$, the consistency is preserved by applying successively the degree-elevation algorithm starting from $X^{(0)}$. More precisely, $X^{(l)}$ is obtained from $X^{(l-1)}$ by applying $n_l - n_{l-1}$ times the degree-elevation algorithm, and thus, we obtain a family of embedded sub-parametrizations with a progressive increase of the number of control points.

Note that, the conditions at the endpoints $\mathrm{x}_0^{(l)} = x_0^\Gamma$ and $\mathrm{x}_n^{(l)} = x_m^\Gamma$ are automatically verified at each level if these relations occur for $l = 0$. Furthermore, additional geometrical constraints are often imposed on the shape: for instance, a vertical tangent at the origin is a standard constraint at the leading edge for airfoil profiles. With a Bézier curve, the derivatives at the endpoints can be easily managed, and in particular, a vertical tangent at the origin can be enforced by the condition $\mathrm{x}_0 = \mathrm{x}_1$. It is easy to see that the degree elevation also preserves this condition. Thus, using the present procedure, the different sub-levels are fully determined by the construction of a consistent coarsest sub-level.

Construction of a consistent coarsest sub-level

The simplest way to construct an initial set of abscissas for Bézier control points is to consider the set reduced to the endpoints, i.e. $\tilde{X}_0 = (x_0^\Gamma, x_m^\Gamma)$. In this case, eq. (3.14) with $n_l = 1$ gives the following expression for the parameters t_k:

$$t_k = \frac{x_k^\Gamma - x_0^\Gamma}{x_m^\Gamma - x_0^\Gamma} \quad \text{for } k = 0, \cdots, m.$$

On another hand, any $\tilde{X}_l$ obtained by applying successively the degree-elevation process starting from $\tilde{X}_0$ yields an uniform distribution of points on the interval $[x_0^\Gamma, x_m^\Gamma]$. Furthermore, if, as for the case of the nozzle inverse problem presented in Sec. 4.8.2 (test-case 1), the $(x_k^\Gamma)_{k=0,m}$ are uniformly distributed, it directly follows that the parameters t_k are also uniformly distributed. Thus, a consistent coarsest level can be easily obtained by simply choosing an uniform distribution for both $X^{(0)}$ and $T^{(0)}$.

Let us, now, consider the case in which a vertical tangent should be imposed at the origin (as for instance, the airfoil inverse problem of Sec. 4.8.2 (test-case 2)). In this case, $\tilde{X}_0 = (x_0^\Gamma, x_0^\Gamma, x_m^\Gamma)^T$ gives the simplest initial set of abscissas for Bézier control points. $\tilde{X}_0$ is associated to the following distribution of the parameters t_k:

$$t_k = \sqrt{\frac{x_k^\Gamma - x_0^\Gamma}{x_m^\Gamma - x_0^\Gamma}} \quad \text{for } k = 0, \cdots, m. \tag{4.17}$$

A consistent coarsest level is then obtained considering $T^{(0)}$ defined by (4.17) and $X^{(0)}$ being any $\tilde{X}_l$ obtained from $\tilde{X}_0$ by applying successively the degree-elevation process.

4.4.2 Formulation based on shape functions basis

Shape representation through analytical shape functions

As pointed out in Sec. 4.2.3, another classical choice of parametrization for aerodynamic shape design is to consider the curve coefficients in some basis of shape functions. Then, given $(x_k^\Gamma)_{k=0,m}$, the shape grid-points abscissas and the particular basis $(f_q)_{q=1,n}$, the basis coefficients $\alpha^{(n)} = (\alpha_1, \cdots, \alpha_n)^T$ can be easily related to the ordinates of the shape grid-points γ by:

$$y(x_k^\Gamma) = \sum_{q=1}^{n} \alpha_q f_q(x_k^\Gamma) \quad \forall k = 1, \cdots, m-1 \tag{4.18}$$

Then, thanks to the linearity between $\alpha^{(n)}$ and γ, it is still possible to define a multilevel strategy as done previously for the Bézier points, i.e. considering γ as control variables. Here, a sub-level is simply obtained considering only a subset of the basis shape functions. More precisely, taking the first l coefficients, i.e., $\alpha^{(l)} = (\alpha_1, \cdots, \alpha_l)^T$, the preconditioning matrix $M^{(l)}(M^{(l)})^T$ to apply in (4.7) is determinate by:

$$M_{ij}^{(l)} = f_j(x_i^\Gamma) \quad \text{for } j = 1, l \text{ and } i = 1, m-1. \tag{4.19}$$

Sub-parametrizations based on orthonormal shape functions

As pointed out in Sec. 4.3.3, the number of freedom degrees considered is not enough to well define the notion of level coarseness. More specifically, to define a coarser level by taking less shape functions as done previously, does not make sense if a hierarchy between the different shape functions can not be established. This difficulty can be avoided by considering orthonormal shape functions since these functions are increasingly oscillatory (see Fig. 4.1), and consequently, each shape function can be arranged with respect to its degree of high-frequency. The interest of using orthonormal functions associated to multigrid approach has been pointed out in [Kuruvila et al., 1994] in which four orthogonal functions have been defined starting from a NACA 0012 airfoil (this set of shape functions has been successively extended to ten by [Chang et al., 1995] in order to represent a supercritical wing). More recently, in [Catalano et al., 2005] a family of orthonormal shape functions based on Bézier curves has been defined and associated to a multigrid-aided finite-difference method.

Let us, here, consider the following family of functions which just corresponds to consider the functions defined by [Kuruvila et al., 1994] for any degree:

$$g_1(x) = \sqrt{x} - x \quad \text{and} \quad g_q = x^{q-1}(1-x) \quad \text{for } q \geq 2$$

Then, the orthonormal shape functions (with respect to $L^2\big([0,1]\big)$ scalar product) $(f_q)_{q=1,n}$ are obtained by applying a classical Gram-Schmidt procedure. Note that, here, the orthonormalisation is not done numerically but analytically through symbolic calculus. In this way, we have access possibly to the exact derivatives, which can be useful, for instance, for imposing some geometrical constraints. The analytical expressions for the orthonormal shape functions (f_q) with $q = 1, \ldots, 9$ are given in Table 4.1.

4.5 Reinterpretation of the new multilevel approaches

4.5.1 Parametrization based on Bézier control points

In the present formulation, even if each sub-parametrization corresponds to the ordinates of a set of Bézier control points, these control points do not explicitly appear in the definition of the descent direction since only the parameters $(t_k)_{k=0,m}$ are directly involved in (4.13). Thus, the knowledge of the particular position of the Bézier control points is not required in the practical algorithm implementation. Nevertheless, an explicitly dependence on these control points can be, also, reintroduced in the formulation.

$$f_1(x) = \sqrt{30}(\sqrt{x} - x)$$

$$f_2(x) = \frac{\sqrt{10}}{3}(-13\sqrt{x} + 27x - 14x^2)$$

$$f_3(x) = \frac{\sqrt{1190}}{51}(35\sqrt{x} - 108x + 154x^2 - 81x^3)$$

$$f_4(x) = \frac{3\sqrt{442}}{221}(-123\sqrt{x} + 500x - 1330x^2 + 1701x^3 - 748x^4)$$

$$f_5(x) = \frac{\sqrt{10582}}{1443}(671\sqrt{x} - 3375x + 14210x^2 - 31752x^3 + 32076x^4 - 11830x^5)$$

$$f_6(x) = \frac{\sqrt{962}}{111}(-221\sqrt{x} + 1323x - 8036x^2 + 27216x^3 - 46332x^4 + 38038x^5 - 11988x^6)$$

$$\begin{aligned} f_7(x) = \frac{\sqrt{78}}{13}(&113\sqrt{x} - 784x + 6468x^2 - 30618x^3 + 76956x^4 - 104104x^5 + 71604x^6 + \\ &- 19635x^7) \end{aligned}$$

$$\begin{aligned} f_8(x) = \frac{\sqrt{90610}}{1599}(&-493\sqrt{x} + 3888x - 41748x^2 + 261954x^3 - 901692x^4 + 1769768x^5 + \\ &- 1975428x^6 + 1166319x^7 - 282568x^8) \end{aligned}$$

$$\begin{aligned} f_9(x) = \frac{\sqrt{157358}}{12423}(&3439\sqrt{x} - 30375x + 411180x^2 - 3293136x^3 + 14779908x^4 + \\ &- 39169130x^5 + 62653500x^6 - 59376240x^7 + 30669496x^8 - 6648642x^9) \end{aligned}$$

Table 4.1: Analytical expression of the orthonormal shape functions f_q with $q = 1, \ldots, 9$.

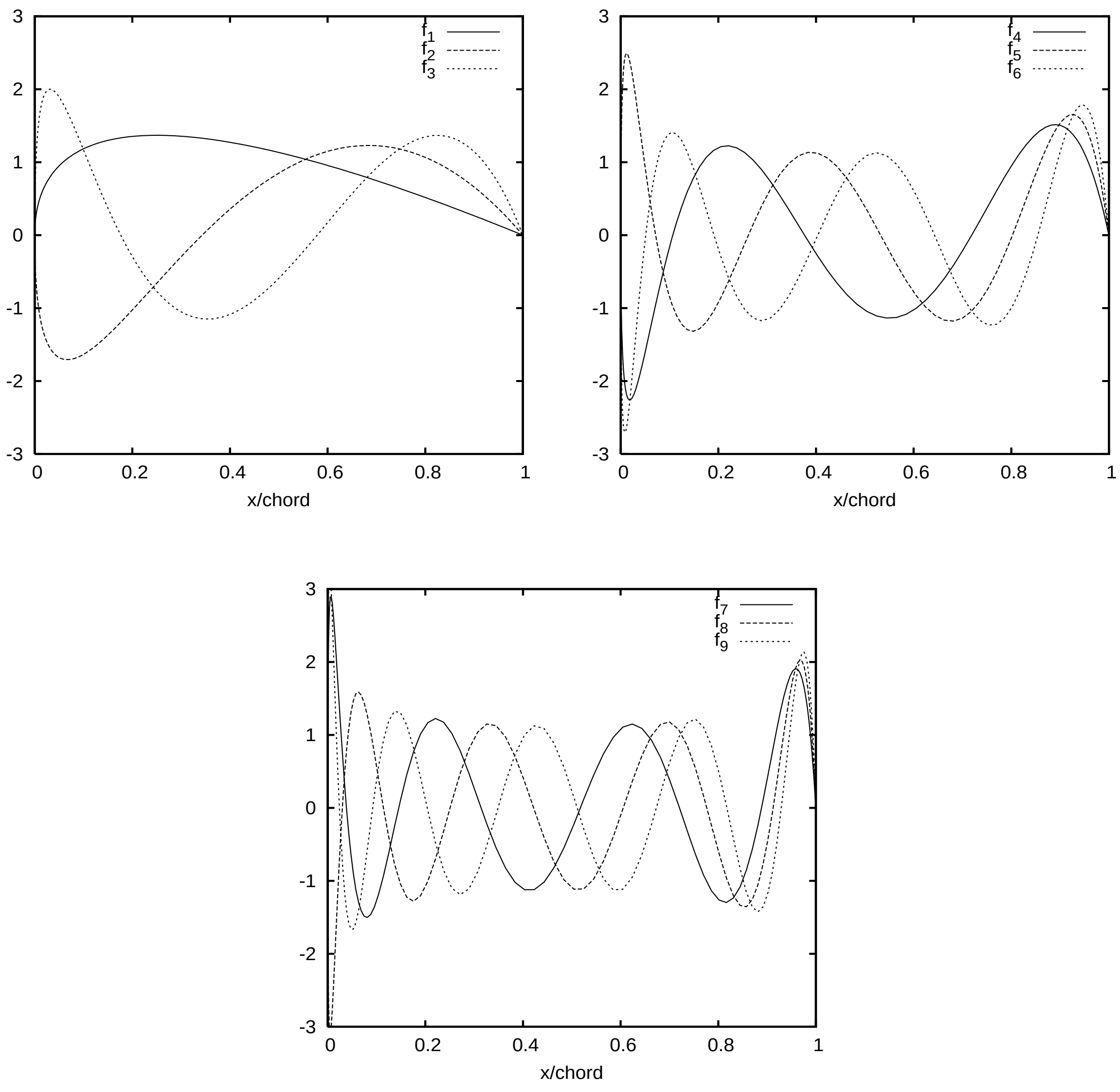


Figure 4.1: Orthonormal shape functions: the 1-2-3 (top-left), the 4-5-6 (top-right) and the 7-8-9 (bottom) basis functions.

Let us consider a particular level l, with $0 \leq l < L$, L corresponding to the finest level of Bézier control points.

$$\text{for } k = 1, m-1 \qquad y_k^\Gamma = \sum_{q=1}^{n_l - 1} B_q^{n_l}(t_k)\, \mathrm{y}_q^{(l)} + b_k^{(l)} = \sum_{q=1}^{n_L - 1} B_q^{n_L}(t_k) \mathrm{y}_q^{(L)} + b_k^{(L)}$$

The last equality is due to the fact that $(\mathrm{y}_q^{(L)})_{q=0,n_L}$ is obtained from $(\mathrm{y}_q^{(l)})_{q=0,n_l}$ by a degree-elevation process, i.e. using iteratively the relation (4.16) for the ordinates. Thus, the prolongation operator at level l, i.e. $f^{(l)} : \alpha \longmapsto \gamma = M^{(l)}\alpha + b^{(l)}$, verifies

$$f^{(l)} = f^{(L)} \circ d_l^L$$

in which $f^{(L)} : \beta \longmapsto \gamma = M^{(L)}\beta + b^{(L)}$ is the operator relating the finest level of Bézier control points to the shape grid-points while d_l^L corresponds to the prolongation operator from level l to level L.

Note that the degree-elevation process is linear, and thus, excluding the endpoints, the application d_s, which furnishes s internal control points from $s-1$ ones, is affine. More precisely, d_s is defined by:

$$\begin{array}{rll} d_s : & \mathbb{R}^{s-1} & \longrightarrow \mathbb{R}^s \\ & \alpha & \longmapsto D_s\alpha + \lambda_s \end{array} \quad \text{with} \quad \left\{ \begin{array}{l} \lambda_s = \dfrac{1}{s+1}(y_0^\Gamma, 0, \cdots, 0, y_m^\Gamma)^T \\[2ex] (D_s)_{ij} = \dfrac{1}{s+1}\big((s+1-i)\delta_{ij} + i\delta_{(i-1)j}\big) \end{array} \right.$$

Consequently, the prolongation operator from level l to the finest level L of Bézier control points can be expressed as $d_l^L = d_{n_L-1} \circ \cdots \circ d_{n_l}$, and thus, is an affine application associated to the matrix $\mathcal{D}_l^L = D_{n_L-1} \cdots D_{n_l+1} D_{n_l}$. In conclusion, at a particular level l, one can directly relate the current set of Bézier control points with the shape grid-points through $f^{(l)}$, or alternatively, apply successively the operators d_l^L and $f^{(L)}$. The two ways are equivalent from a theoretical view point, even if they can differ in the implementation. Thus, the following algorithm can be used instead of eqs. (4.7) and (4.13):

$$\gamma_{r+1} = \gamma_r - \omega_r\, M^{(L)} \mathcal{D}_l^L (\mathcal{D}_l^L)^T (M^{(L)})^T g_r \tag{4.20}$$

The algorithm (4.20) exactly corresponds to (4.10) of Sec. 4.3.3, in which $M = M^{(L)}$ and $D^{(l)} = \mathcal{D}_l^L$. Thus, the approach described in Sec. 4.4.1 can be also interpreted as a multilevel gradient-based method in which the Bézier control points (on the finest level L) are taken as control variables. Then, (4.9) corresponds, in this specific case, to the following iterative algorithm (expressed, here, at iteration $r+1$ and sub-level l):

$$\beta_{r+1} = \beta_r - \omega_r\, \mathcal{D}_l^L (\mathcal{D}_l^L)^T \mathcal{G}_r \tag{4.21}$$

where $\beta = \alpha^{(L)} = (y_1, \cdots, y_{n_L-1})^T$ and $\mathcal{G}_r = (M^{(L)})^T g_r$ is the functional gradient with respect to β at iteration r.

Note that this pointed out a strong analogy with the approach proposed in [Désidéri, 2003] since, in the both algorithms, the degree-elevation property is directly used to prolongate from a coarse level to the complete set of design variables, which are Bézier control points (a more detailed comparison between the two approaches is proposed in Sec. 4.7).

4.5.2 Parametrization based on shape functions

The degree-elevation property, which has been used as prolongation operator for the Bézier-based embedded parametrization, allows to consider the same geometrical curve as an element of a larger space, An analogous operation can be done here; indeed, the curve with coordinates $(\alpha_1, \cdots, \alpha_l)^T$ in the basis $(f_1, \cdots, f_l)^T$ is exactly equal to the curve with coordinates $(\alpha_1, \cdots, \alpha_l, 0, \cdots, 0)^T$ in the basis $(f_1, \cdots, f_n)^T$. This simple way to prolongate an element of $\mathbb{R}^l$ in $\mathbb{R}^n$ corresponds to consider the linear prolongation operator, which is associated with the rectangular matrix $N_l^n \in \mathbb{R}^{n \times l}$ defined by

$$\left(N_l^n\right)_{ij} = \delta_{ij} \quad \text{for} \quad i = 1, n \text{ and } \ j = 1, l$$

Then, taking $\beta = \alpha^{(n)}$ as control variables, a multilevel preconditioned gradient method can be defined as follows:

$$\beta_{r+1} = \beta_r - \omega_r \, N_l^n \left(N_l^n\right)^T \text{grad}_\beta j(\beta_r)$$

and the corresponding updating of the shape grid-points is expressed as:

$$\gamma_{r+1} = \gamma_r - \omega_r \, M^{(n)} N_l^n \left(M^{(n)} N_l^n\right)^T \text{grad}_\gamma j(\gamma_r)$$

in which $M^{(n)}$ is defined by (4.19) with $l = n$.

Thus, as for Bézier-based parametrization, a direct correlation between the different sub-level has been established allowing the definition of an algorithm as (4.10).

4.6 Computation of an approximate gradient

As previously pointed out, an adjoint approach seems to be the more suitable way to compute the sensitivities. However, an approximate gradient, computed using finite-differences sensitivities, remains interesting as far as the exact differentiation is a too complex task or some problems of non-differentiability are present. Note that a multilevel method, as described in this study, is *a priori* independent of the way in which the discrete gradient is computed. Nevertheless, in the case of an approximate gradient, the computational cost can be noticeably reduced by coupling with a multilevel approach in which the finite differences are applied on the coarser levels (see [Beux and Dervieux, 1994]). This multi-level/finite-differences formulation has been used in [Held et al., 2002] coupled with a one-shot approach while, in [de' Michieli Vitturi and Beux, 2006], a multilevel adjoint-free gradient formulation is proposed in which finite differences are used to approximate the flow sensitivities.

Sections 4.3.3 and 4.5 show that there are two possible ways to prolongate from a particular level l to γ, the shape grid-points ordinates. Indeed, it can be directly applied $f^{(l)}$ or, alternatively, used successively the affine operators d_l^L and $f^{(L)}$, i.e. in terms of functional, it signifies that $\mathcal{J}^{(l)} = j \circ f^{(l)}$ can be also expressed as $\mathcal{J}^{(L)} \circ d_l^L$. Consequently, there are also two possible ways to compute its Gâteaux derivative. More precisely, for all h in $\mathbb{R}^{n_l - 1}$, we have:

$$\left[\mathcal{J}^{(l)}\right]'(\alpha)(h) = j'(\gamma)\left(M^{(l)} h\right) = \lim_{\theta \to 0^+} \frac{j\left(\gamma + \theta M^{(l)} h\right) - j(\gamma)}{\theta} \tag{4.22}$$

and

$$\left[\mathcal{J}^{(l)}\right]'(\alpha)(h) = \left(\mathcal{J}^{(L)}\right)'(\beta)\left(\mathcal{D}_l^L h\right) = \lim_{\theta\to 0^+} \frac{\mathcal{J}^{(L)}\left(\beta + \theta\mathcal{D}_l^L h\right) - \mathcal{J}^{(L)}(\beta)}{\theta} \tag{4.23}$$

with $\gamma = M^{(l)}\alpha \in \mathbb{R}^{m-1}$ and $\beta = \mathcal{D}_l^L\alpha \in \mathbb{R}^{n_L-1}$ respectively.

Consequently, two different algorithms can be proposed in the context of an approximate gradient. The first one is obtained considering the ordinates of the shape grid-points as design variables, and can be expressed as follows:

$$\gamma_{r+1} = \gamma_r - \omega_r M^{(l)}\ \tilde{g}_r^{(l)}(\epsilon) \tag{4.24}$$

in which $\tilde{g}_r^{(l)}(\epsilon)$ is an approximation of $\left(M^{(l)}\right)^T g_r$ obtained using (4.22), and thus, is defined by:

$$\forall i = 1, \cdots, n_l - 1 \qquad \left(\tilde{g}_r^{(l)}(\epsilon)\right)_i = \frac{1}{\epsilon}\left[j\left(\gamma_r + \epsilon M^{(l)} e_i\right) - j(\gamma_r)\right]$$

in which e_i is the i-th element of the canonical basis of $\mathbb{R}^{n_l-1}$ and ϵ a small given parameter.

On another hand, from (4.21) the following algorithm is obtained if the ordinates of the Bézier control points[2] are directly used as design variables:

$$\beta_{r+1} = \beta_r - \omega_r \mathcal{D}_l^L\ \tilde{\mathcal{G}}_r^{(l)}(\epsilon) \tag{4.25}$$

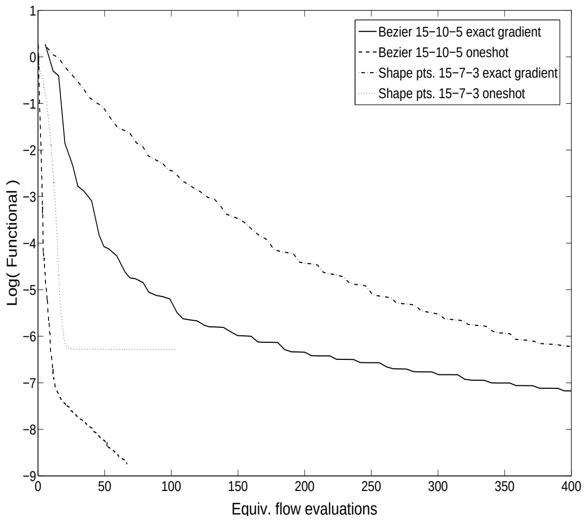
in which $\tilde{\mathcal{G}}_r^{(l)}(\epsilon)$ approximates $\left(\mathcal{D}_l^L\right)^T \mathcal{G}_r$ using (4.23), and thus:

$$\forall i = 1, \cdots, n_l - 1 \qquad \left(\tilde{\mathcal{G}}_r^{(l)}(\epsilon)\right)_i = \frac{1}{\epsilon}\left[\mathcal{J}^{(L)}\left(\beta_r + \epsilon\mathcal{D}_l^L e_i\right) - \mathcal{J}^{(L)}(\beta_r)\right]$$

Since $M^{(l)} = M^{(L)}\mathcal{D}_l^L$, applying the affine operator $f^{(L)}$ to (4.25) one can easily recover algorithm (4.24) in which $\gamma_r = f^{(L)}(\beta_r)$. Thus, according to what happens for an exact gradient computation, the two algorithms are equivalent, and moreover, in the both cases, at level l, only $n_l - 1$ evaluations of the cost function are needed.

In order to illustrate the interest of the multilevel approximate gradient formulation, some results presented in [Martinelli and Beux, 2006] are reported in Fig. 4.2. For an inverse problem similar to the one presented in Sec. 4.8.2, V-cycle three-levels approaches are applied both for the adjoint method and the finite-differences one associated with a one-shot approach (as described in [Held et al., 2002]). The good behaviour of the one-shot approach without adjoint, already obtained in [Held et al., 2002], is also confirmed for the Bézier-based parametrization. Moreover, the same improvement obtained by using the Bézier parametrization instead of the shape grid-points is observed with the multilevel/finite-differences one-shot approach.

[2]note that the notations are, here, coherent with the ones used in Sec. 4.3.1 and successively in Sec. 4.5.1. Obviously, the results of this section are also true for the case of a parametrization based on shape functions as described in sections 4.4.2 and 4.5.2.



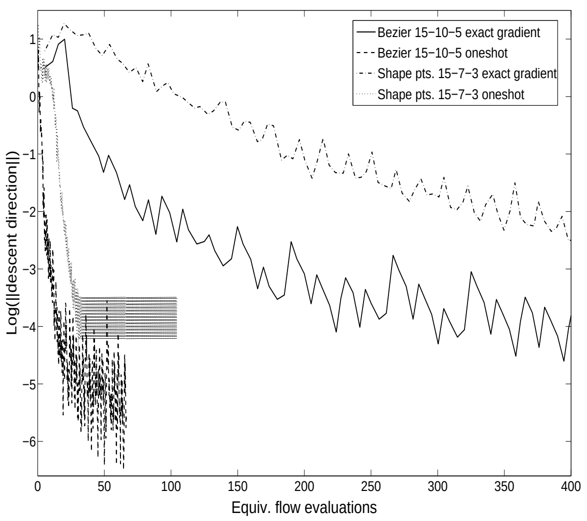
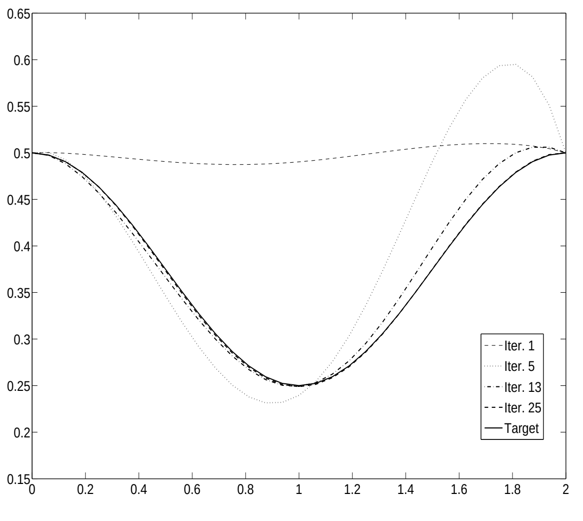


Figure 4.2: Convergence histories for $j(\gamma_r)$ (left) and $d_r^{(l)}$ (right) (shape grid-points and Bézier-based parametrizations): comparison between the exact adjoint gradient computation and the gradient approximated by divided differences associated with a one-shot approach.



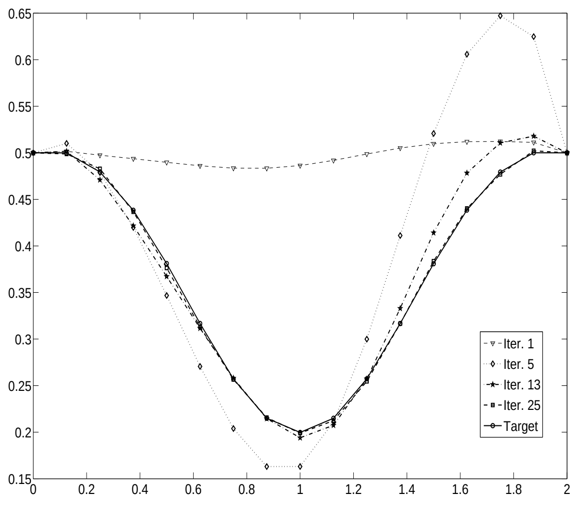


Figure 4.3: One-shot/multilevel approach using Bézier parametrization: successive shapes (left) and Bézier control points (right). Comparison between the target solution and solutions after 1, 2, 4 and 7 V-cycles (i.e., after 1, 5, 13 and 25 optimization iterations).

4.7 Reinterpretation of the approach proposed in [Désidéri, 2003]

As previously pointed out, the present Bézier-based multilevel approach has strong similarities with the approach proposed by [Désidéri, 2003]. This approach has been defined independently of the specific optimization algorithm, and thus, is *a priori* gradient-free. Nevertheless, in order to try to better see the analogies, let us consider a two-level algorithm, in which a steepest-descent-like method is used both for the finest level and for the sub-level (see [Désidéri, 2007; Abou El Majd et al., 2008]). It consists in alternate a relaxation phase on the upper-level of Bézier parametrization (few shape optimization iterations) with a coarse-level correction phase, in which the shape perturbation is parametrised on a coarse Bézier sub-parametrization. Then, the cycle c of the algorithm can be rewritten with the notations used in the present study as:

1. Upper level: given $\beta_0^{(c)} \in \mathbb{R}^{n_L-1}$ from the previous cycle, iterate:
$$\text{for } \ r=1, r_L \qquad \beta_r^{(c)} = \beta_{r-1}^{(c)} - \tilde{\omega}_r \text{grad}_\beta(\beta_{r-1}^{(c)}) \tag{4.26}$$
2. Solve the following minimisation problem on the coarser level:
$$\text{Find } \ \bar{\delta}_\alpha^{(c)} = Arg \min_{\delta_\alpha} \mathcal{I}^{(c)}(\delta_\alpha) \tag{4.27}$$
in which $\mathcal{I}^{(c)}(\delta_\alpha) = \mathcal{J}^{(L)}\left(\beta_{r_L}^{(c)} + \tilde{d}_l^L(\delta_\alpha)\right)$ for $\delta_\alpha \in \mathbb{R}^{n_l-1}$.
3. A new cycle can be, then, computed starting from $\beta_0^{(c+1)} = \beta_{r_L}^{(c)} + \tilde{d}_l^L\big(\bar{\delta}_\alpha^{(c)}\big)$ on the upper level.

$\tilde{d}_l^L$ is the prolongation operator from level l to L based on degree-elevation property, which depends on the particular choice of $T^{(l)}$ and $X^{(l)}$. Furthermore, since we are working, here, on shape perturbations, the conditions at the endpoints are $(\delta_\alpha)_0 = 0$ and $(\delta_\alpha)_{n_l} = 0$, and consequently, $\tilde{d}_l^L$ is linear, i.e. $\tilde{d}_l^L(\delta_\alpha) = \tilde{\mathcal{D}}_l^L \delta_\alpha$.

To solve (4.27) and obtain an approximation of $\bar{\delta}_\alpha^{(c)}$, s_l ($s_l \gg r_L$) iterations of a steepest-descent method are performed starting from a given initial solution $(\delta_\alpha)_0$:

$$s=1, s_l \quad (\delta_\alpha)_s = (\delta_\alpha)_{s-1} - \tilde{\omega}_{s-1}\, \text{grad}_{\delta_\alpha} \mathcal{I}^{(c)}\big((\delta_\alpha)_{s-1}\big)$$

Furthermore, let us define $\varphi_{r_L}^{(c)}$ by $\varphi_{r_L}^{(c)}(\delta_\alpha) = \beta_{r_L}^{(c)} + \tilde{\mathcal{D}}_l^L \delta_\alpha$, since $\mathcal{I} = \mathcal{J}^{(L)} \circ \varphi_{r_L}^{(c)}$ with $\varphi_{r_L}^{(c)}$ affine, we have:

$$\text{grad}_{\delta_\alpha} \mathcal{I}(\delta_\alpha) = (\tilde{\mathcal{D}}_l^L)^T \text{grad}_\beta \mathcal{J}^{(L)}\big(\varphi_{r_L}^{(c)}(\delta_\alpha)\big).$$

Consequently, the Bézier control points on the finer level are obtained at the end of the cycle by:

$$\begin{aligned} \beta_0^{(c+1)} &= \beta_{r_L}^{(c)} + \tilde{\mathcal{D}}_l^L \big(\delta_\alpha\big)_{s_l} \\ &= \beta_{r_L}^{(c)} - \tilde{\mathcal{D}}_l^L \big(\tilde{\mathcal{D}}_l^L\big)^T \sum_{i=0}^{s_l-1} \tilde{\omega}_i\, \text{grad}_\beta \mathcal{J}^{(L)}\left(\beta_{r_L}^{(c)} + \tilde{\mathcal{D}}_l^L (\delta_\alpha)_i\right) - \tilde{\mathcal{D}}_l^L\, (\delta_\alpha)_0 \end{aligned}$$

Let us consider, now, the approach defined in sections 4.4.1 and 4.5.1, associated with a two-level algorithm with r_l and s_l iterations for the upper and coarser levels respectively. From (4.21), the following expression is obtained for the Bézier control points on the finest level are obtained at the end of the cycle c:

$$\bar{\beta}_0^{(c+1)} = \beta_{[r_0(c)+s_l]} = \beta_{r_0(c)} - \mathcal{D}_l^L \big(\mathcal{D}_l^L\big)^T \sum_{i=0}^{s_l-1} \omega_i \, \mathcal{G}_{[r_0(c)+i]}$$

with $r_0(c) = (c-1)(r_L + s_l) + r_L$.

Proposition 4.1. *With an adequate choice of the parameters which characterize the two-level algorithms, the two strategies coincide at the end of each cycle, i.e.*

$$\beta_0^{(c+1)} = \bar{\beta}_0^{(c+1)} \qquad \textit{for each cycle} \;\; c$$

Proof. Let us consider that the same choice of family of embedded sub-parametrizations, i.e. the same choice of $T^{(l)}$ and $X^{(l)}$ at each sub-level, is done, and thus, in particular $\tilde{\mathcal{D}}_l^L = \mathcal{D}_l^L$.

- If the two algorithms start from the same shape configuration, then $\beta_0^{(1)} = \bar{\beta}_0^{(1)}$.
- Let us suppose, now, that $\beta_0^{(c)} = \bar{\beta}_0^{(c)}$ for $c \geq 1$.
  If the same 1D search strategy (i.e., the same scalar parameter ω) is employed, the iterative algorithms on the upper level, i.e. (4.21) and (4.26), coincide, and thus, we have that $\beta_{r_0(c)} = \beta_{r_L}^{(c)}$. Moreover, choosing $(\delta_\alpha)_0 = 0$ and supposing that $\tilde{\omega}_i = \omega_{[r_0(c)+i]}$ at each iteration on the lower level, we obtain that

  $$\beta_{[r_0(c)+i]} = \varphi_{r_L}^{(c)}\big((\delta_\alpha)_i\big) \qquad \text{for} \;\; i = 0, s_l$$

  and consequently

  $$\beta_0^{(c+1)} = \bar{\beta}_0^{(c+1)}$$

□

4.8 Numerical experiments

4.8.1 Parametrization and shape representations

Since in a multi-level approach, different levels of coarseness are involved, it should be interesting to evaluate the capability of shape representation of each level. For instance, let us investigate on the accuracy in which a cambered RAE2822 profile can be represented by a parametrization based on Bézier control points. At a level l, the Bézier curvefit, at the discrete least-squares sense, is obtained minimising with respect to $\alpha^{(l)} = \big(\mathrm{y}_1, \cdots, \mathrm{y}_{n_l-1}\big)^T$ the Euclidean norm in $\mathbb{R}^{m-1}$ of the residual $Res\big(\alpha^{(l)}\big) = \big[M^{(l)}\alpha^{(l)} - b^{(l)}\big] - \gamma^{target}$. On another hand, for the parametrization defined in Sec. 4.4.2, since we consider an orthonormal basis, the reconstruction of a given shape f, known at $x_0^\Gamma, \cdots, x_m^\Gamma$, on a sub-level l is obtained as $\sum_i^l \langle f, f_i \rangle \, f_i$ in which $\langle ., . \rangle$ is an adequate discrete approximation of the $L^2\big([x_0^\Gamma, x_m^\Gamma]\big)$ scalar product.

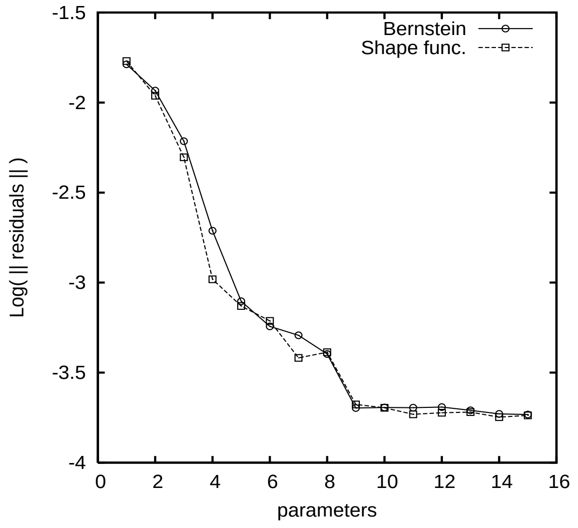


Figure 4.4: Curvefit error ($\log \|Res((\alpha^{(l)}))\|_\infty$) with respect to the number of parameters for the two kinds of parametrizations.

The curvefit error with respect to the number of parameters[3] is plotted in Fig. 4.4. A very similar behaviour is observed for the two kinds of parametrization. A rather good representation of the RAE2822 profile is already furnished with few degrees of freedom (as, also, shown in Fig. 4.5 with the plot of profiles for small number of parameters) whereas for more than 8 parameters a plateau is obtained with an error value of about $2\ 10^{-4}$. Obviously, the capability of shape representation is strongly correlated to the choice of the particular configuration (e.g. for the case of a symmetric NACA0012 profile, a Bézier curvefit error minor than 10^{-8} is obtained). Note that, an increase of the number of Bézier control points amplifies the irregularity of the control polygon (see Fig. 4.6). Consequently, a very small variation in the solution space (shape) can correspond to a large variation in the design space (Bézier control points). This depends on the particular shape (with the NACA0012, very regular control polygons are obtained), but also, on the particular construction of the sub-parametrizations, i.e. on the choice of $T^{(l)}$ and $X^{(l)}$. This problem has been already pointed out in [Tang and Désidéri, 2002; Abou El Majd et al.], in which a procedure of parametrization adaptation is also proposed which acts dynamically during the optimization procedure.

4.8.2 Numerical experiments on 2D inverse problems

Test-case 1: a 2D nozzle inverse problem

The first test-case, already used for the multilevel approach associated to shape grid-point coordinates parametrization [Beux, 1994; Beux and Dervieux, 1992, 1994; Held et al., 2002], is a

[3]Two Bézier curves are considered: one for the upper side and one for the lower one. Thus, the number of variables really involved is two times more. In particular, 30 parameters are globally used on the finest level of parametrization.

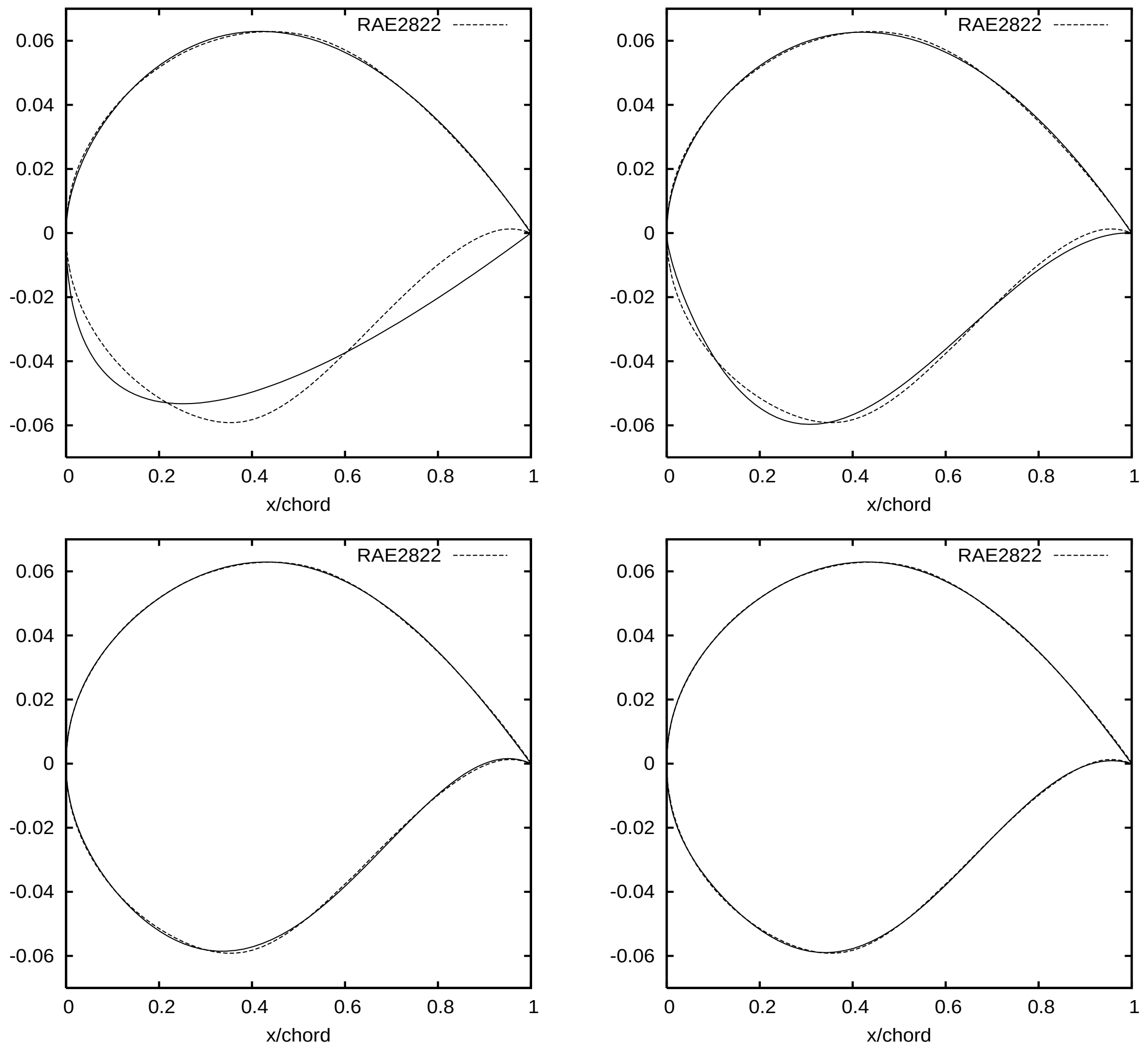


Figure 4.5: Curvefits of RAE2822 profile by orthonormal shape functions: using 2 (top-left), 3 (top-right), 4 (bottom-left) and 5 (bottom-right) parameters respectively.

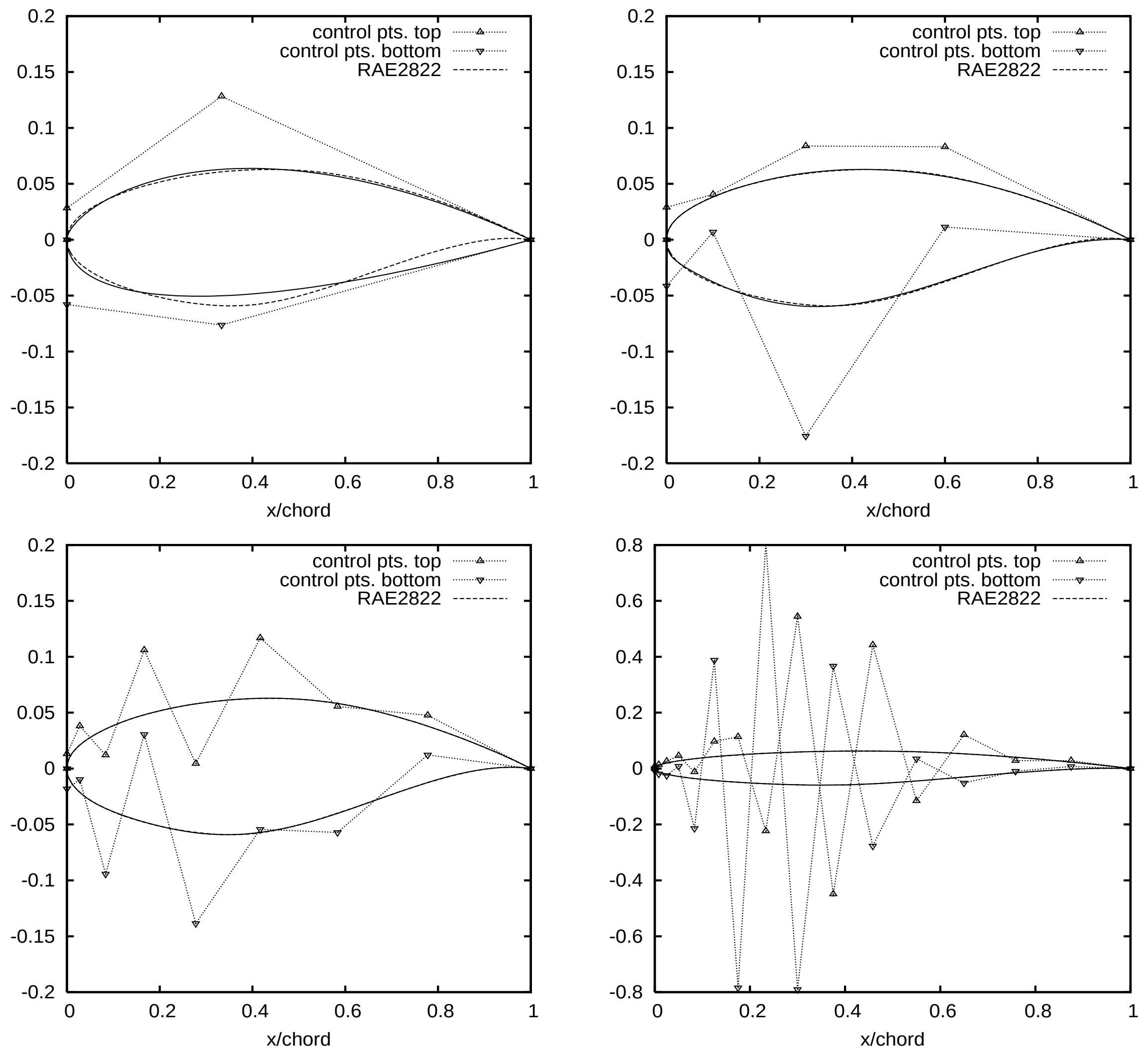


Figure 4.6: RAE2822 curvefits with 2 (top-left), 4 (top-right), 8 (bottom-left) and 15 (bottom-right) Bézier control points: profiles and corresponding control polygons.

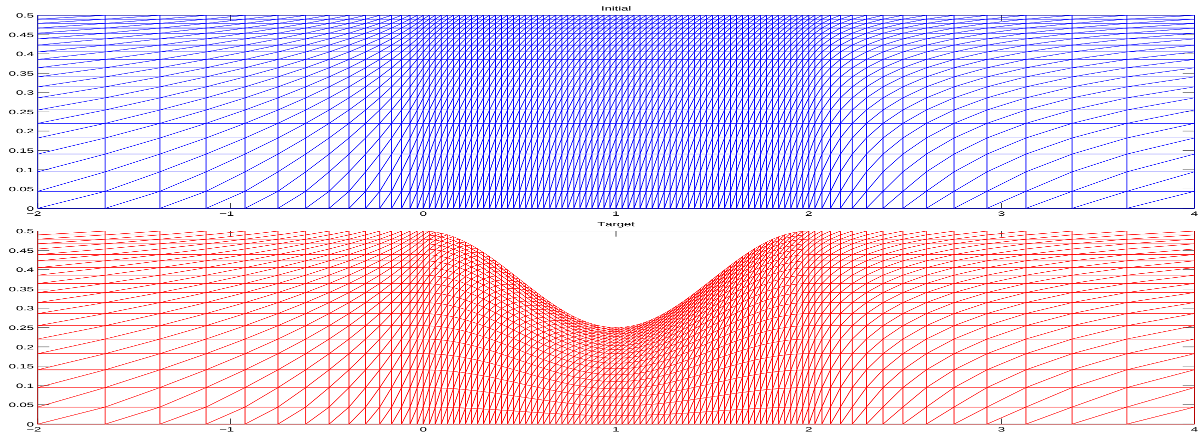
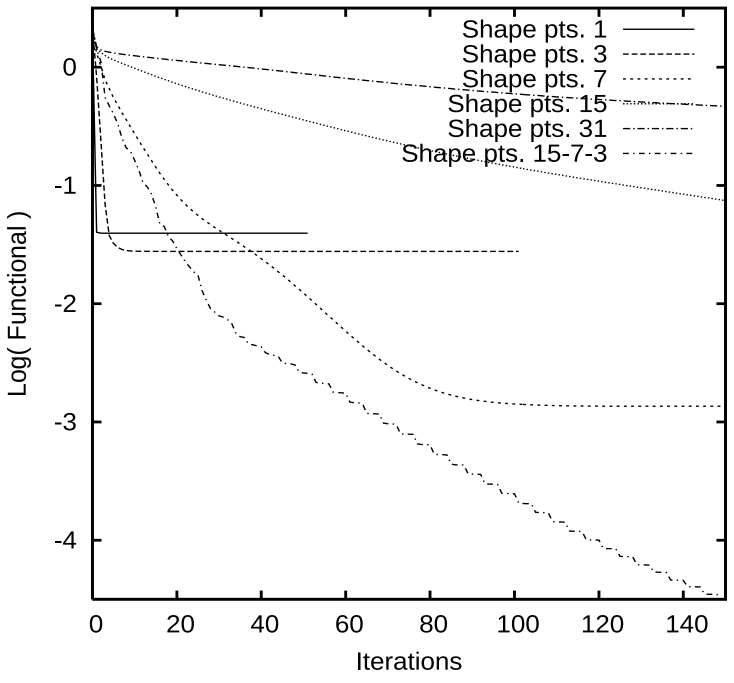


Figure 4.7: Test-case 1: computational mesh: initial (top) and target configuration (bottom)

2D convergent-divergent nozzle inverse problem for inviscid subsonic flows (the flow is modelled, here, by the Euler equations). Here, the particular inverse problem is characterised by an initial constant-section nozzle and a target sine shape. For the considered mesh of 1900 nodes (see Fig. 4.7), 63 shape grid-points are available. Since, the abscissas of the shape grid-points points are uniformly distributed, an uniform distribution is also taken for the abscissas of the Bézier control points at each level. Fig. 4.8a shows the convergence history obtained using the original shape grid-points parametrization. It clearly shows the effect of each level, indeed, the lower is the number of control parameters involved, the lower is the accuracy of the solution, but also, the faster is the convergence to the solution. Note that, the one-level method with 31 parameters already corresponds to a preconditioned gradient method since 63 degrees of freedom are available for the present mesh. Then, advantaging of the speed-up on the coarser levels, the multilevel approach largely improves the convergence rate to reach an accurate solution given by the higher levels. Concerning the new set of sub-parametrization based on Bézier control points (see Fig. 4.8b), even the one-level approach with 10 or 15 parameters gives interesting results. The corresponding multilevel strategy yields ulterior improvements in the final part of the convergence history. Nevertheless, interpreted as an optimization with respect to the Bézier control points, this multilevelling appears less impressive. Indeed, it seems that the principal gain is due to the use of a Bézier-based parametrization more than the change of control sub-spaces. This can be explained by the lack of convergence speed-up of the coarser level as shown in Fig. 4.8b.

Test-case 2: an airfoil inverse problem

In this second test-case, starting from a symmetric NACA0012 airfoil, the cambered RAE2822 should be rebuilt. The initial flow conditions are characterised by a far-field Mach number of 0.734 and an angle of incidence of 2.79^o. Furthermore, as in the previous test-case, only inviscid flows are considered. A mixed unstructured/structured mesh of 3282 nodes has been generated on a circular computational domain centred on the airfoil (see Fig. 4.9). The definition of



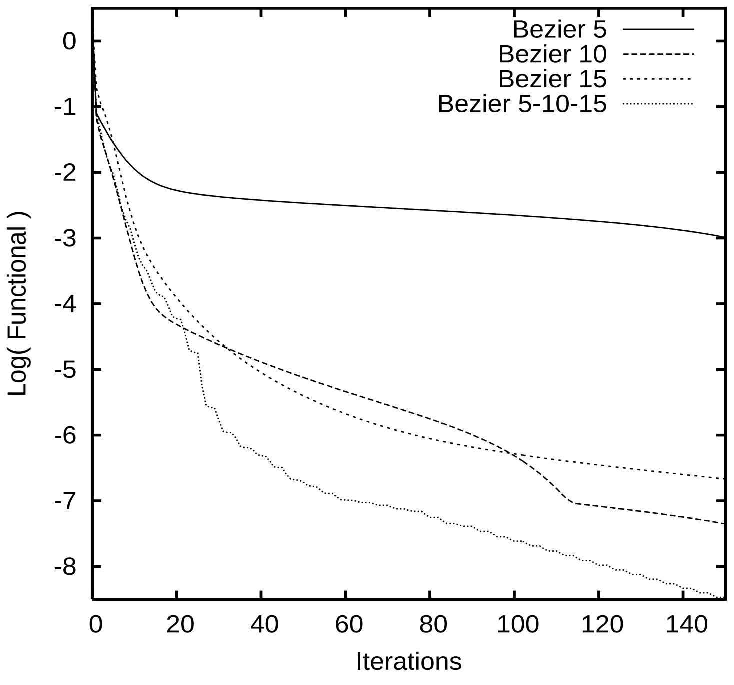


Figure 4.8: Test-case 1: Convergence history for one-level and V-cycle three-level strategies (a) shape grid-points (b) Bézier-based parametrization.

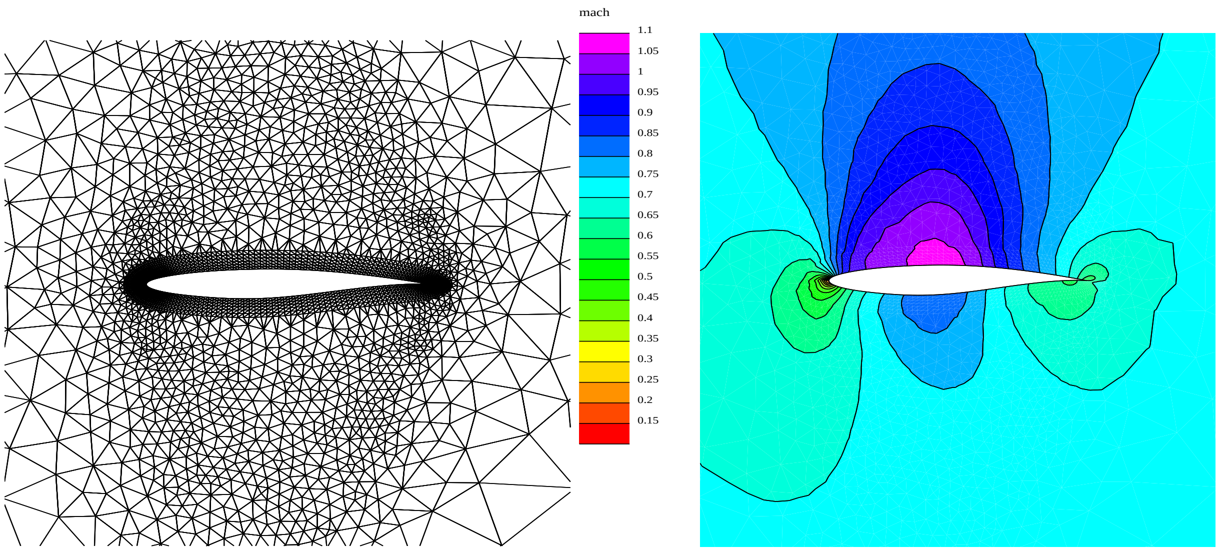
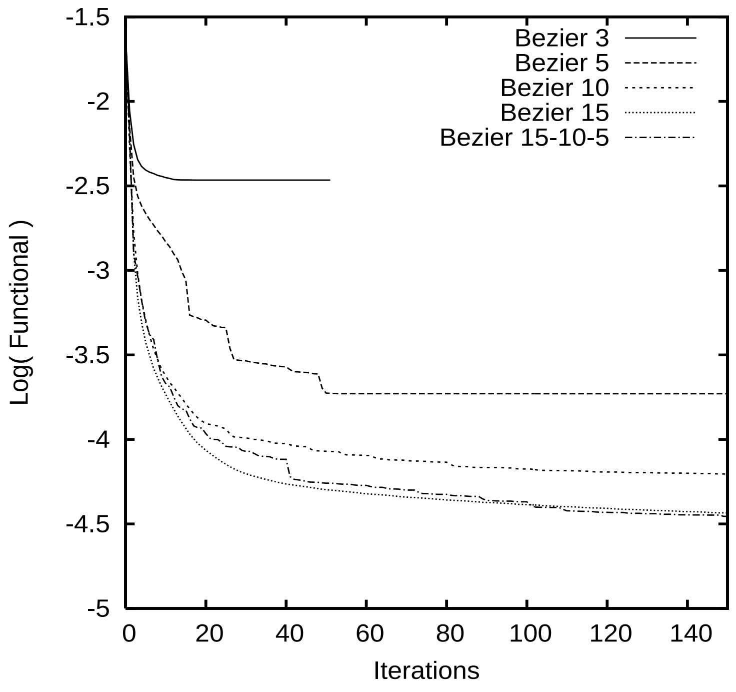


Figure 4.9: Target configuration for test-case 2: computational mesh (left) and Mach contours plot (right).

the Bézier-based sub-parametrizations have been done, here, imposing a vertical tangent at the leading edge, and thus, following the procedure proposed in Sec. 4.4.1.

The direct use of the parametrization based on shape grid-points seems more difficult for this optimization problem, in particular, near the training edge where the upper and lower profiles may be crossed over. This problem can be solved by imposing geometrical constraints, considering only coarser sub-levels in order to increase the smoothing or/and modifying the criterion for the choice of the descent step ω_r. Nevertheless, we choose, here, to consider only the other sub-parametrizations. Fig. 4.10a shows the convergence behaviour for one-level strategies as well as a V-cycle multilevel strategy on three sub-levels (5, 10 and 15 parameters) in the case of a Bézier-based parametrization. The behaviour observed with the previous test-case is magnified, here, since the finer is the level, the more accurate is the solution but without any lost in convergence speed. Consequently, a very good behaviour is obtained with the one-level approach with 15 parameters, but on another hand, the multilevel algorithm does not improve the convergence rate. Fig. 4.10b shows the convergence behaviour obtained with the parametrization based on the orthonormal shape functions. As expected from the results obtained in Sec. 4.8.1, the same kind of behaviour is obtained between the two kinds of parametrization in terms of solution accuracy, for the one-level approaches. However, since, here, the increase of number of parameters corresponds to a decrease of terms of convergence, the multilevel strategy yields an important improvement of the convergence behaviour.



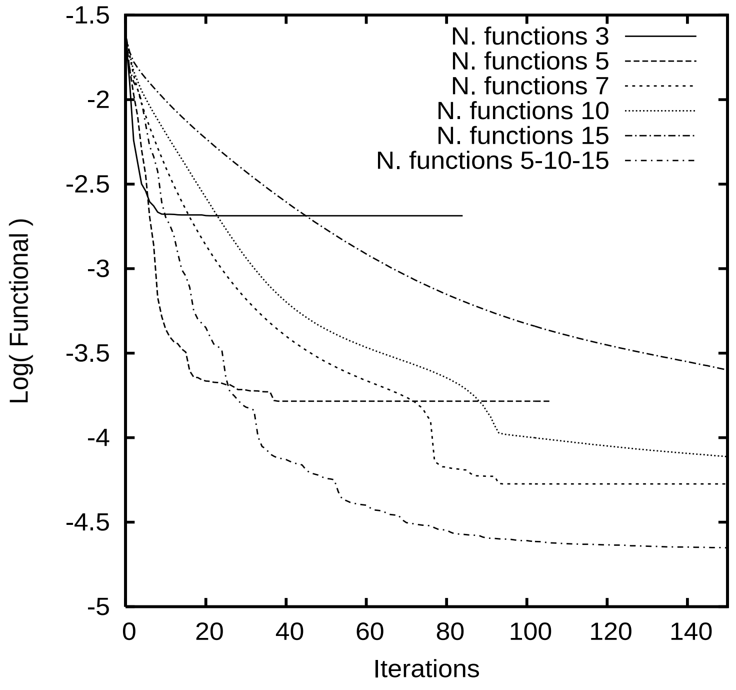


Figure 4.10: Test-case 2: Convergence history for one-level and V-cycle three-level strategies: Bézier-based parametrization (top) and parametrization based on orthonormal shape functions (bottom).

4.9 Conclusion

In the present study, the description of multilevel gradient-based methods for aerodynamic shape design is addressed. Starting from an existing formulation [Beux and Dervieux, 1994] based on an embedded parametrization of shape grid-points and on interpolation operators, a possible generalisation to other kinds of parametrizations is described. This extension requires the elaboration of an adequate family of sub-parametrizations, possibly embedded, associated to affine prolongation operators. Two particular examples are then presented, and since the shape grid-points are yet used as control variables, the resulting approaches can be interpreted as multilevel strategies as defined in [Beux and Dervieux, 1994], in which a particular prolongation operator (i.e. with a particular preconditioning) is applied. However, it can also be reinterpreted directly as multilevel approaches with respect to the new family of shape parametrization. In the first example, the sub-levels are defined through the use of Bézier control points, and starting from a consistent coarsest level, the degree-elevation property of Bézier curves is applied to successively define the different finer levels. In this context, even if, in practice, the Bézier control points can be not explicitly computed, the proposed algorithm can be also interpreted as a descent method for Bézier control points as control variables. Thus, the present descent method seems very close to the approach proposed in [Désidéri, 2003], even if, this last one has been defined independently of the specific optimization algorithm. Indeed, we prove that, under specific conditions, the two approaches can be equivalent. Thus, in this study, it has been built up an explicit link between two kinds of multilevelling: on the one hand, the preconditioned gradient-based method defined [Beux and Dervieux, 1994] and its successive generalisations, and on the other hand, the multilevel algorithm for parametric shape optimization presented in [Désidéri, 2003] and its successive extensions. In the second example, the definition of the set of sub-parametrizations is based on the use of an orthonormal basis of shape functions as shape representation. As for the case of Bézier-based parametrization, a descent direction is obtained considering as control parameters the ordinates of the shape grid-points as well as the finest sub-parametrization.

The numerical experiments shows that the new families of sub-parametrizations have suitable effects, if there are understood as an alternative gradient preconditioning for the optimization with respect to the shape grid-points. Nevertheless, to extend the range of interest of this kind of methods, it should be interpreted as a descent method in which the control variables are taken through the new set of parameters. Note that, in this case, we start with a less inefficient non preconditioned algorithm[4] since the lack of shape smoothness, typical of shape grid-points parametrization, has been avoided. Nevertheless, for the parametrization based on orthonormal shape functions, the multilevel strategy still yields an interesting speed-up of the convergence. Concerning the Bézier-based parametrization, the results are more disappointing since the multilevelling seems poorly efficient. This is due, here, to a good convergence behaviour on the finest levels while the coarsest levels do not yield any additional speed-up, and thus, the basic conditions are not present to apply effectively the multilevel/multigrid principles. Thus, such additional investigation should be performed in order to better understand the

[4]indeed, with respect to the shape grid-points, it can be viewed as a method already preconditioned but associated to a one-level strategy.

present behaviour, which is also inconsistent with the results obtained by J.-A. Désidéri and collaborators. If more attractive results can be obtained for Bézier-based parametrization, since the Bézier curves act as a basic tool for polynomial shape representation, one can also envisage to extend the formulation to more complex shape representation as B-splines (which also have properties of degree-elevation), and also, to 3D case through, for instance, tensorial Bézier parametrization. In the both examples of parametrizations presented in this study, the shape parameters are related by linear or affine application to the set of shape grid-points. Nevertheless, more general cases can be also envisaged since, even if it appears the simplest way to behave, it is not indispensable (the algorithm should not be obligatorily a descent direction with respect to the shape grid-points). Finally, note that the multilevel gradient-based method is not strictly limited to steepest descent approach. Nevertheless, an additive multilevel preconditioner, which can be defined as soon as a set of embedded sub-parametrizations is available, seems more suitable, for instance, to be use with a BFGS-type formulation (see [Courty and Dervieux, 2006]).

Chapter 5

Improvement of functional accuracy through adjoint-error correction

In several applications of CFD, the main task is to compute scalar quantities that are typically defined by integrals over the entire surface of the object being considered (wing, airplane, etc.), like lift and drag coefficient [Moran, 1984], and in the context of shape optimization we want to obtain better performances of such quantities finding an extremum of the functional (for example a minimum of the drag or a maximum of the lift) depending on quantities that define the shape of the object.

Regardless the application, when integral functionals based on approximated PDE solutions are of significant interest, it is worth to consider approaches for enhancing the accuracy of these functional approximations.

An interesting and promising approach is studied and developed in [Pierce and Giles, 2004, 2000] (see also [Venditti and Darmofal, 2002] for combination with mesh adaptation strategies). Pierce and Giles use a numerical approximation of the adjoint solution Π_h and primal residual error $\Psi_{\text{ex}}(\gamma_h, W_h)$ to correct the functional $J(\gamma_h, W_h)$ and to obtain a new estimate

$$J^{\text{c}}(\gamma_h, W_h, \Pi_h) := J(\gamma_h, W_h) - \langle \Pi_h, \Psi_{\text{ex}}(\gamma_h, W_h)\rangle \tag{5.1}$$

for the true value $J(\gamma, W_{\text{ex}})$ that is *superconvergent* in the sense that the remaining error is of smaller convergence order respect to $J(\gamma_h, W_h) - J(\gamma, W_{\text{ex}})$, being proportional to the product of the error in the primal and adjoint solutions.

Another approach to improve the accuracy of a functional is to improve the whole solution using defect-correction [Barrett et al., 1988; Koren, 1988; Skeel, 1981; Stetter, 1978], in which high-order discretization $\Psi_h^{(2)}$ is used to define a residual error that acts as a source term in calculating a higher order approximation $W_h^{(2)}$ using lower order discretization $\Psi_h^{(1)}$:

$$\Psi_h^{(1)}(\gamma_h, W_h^{(2)}) - \Psi_h^{(1)}(\gamma_h, W_h^{(1)}) = -\Psi_h^{(2)}(\gamma_h, W_h^{(1)}).$$

where $W_h^{(1)}$ is the low-order approximation.

It is interesting to note that defect-correction is a *global improvement* of the original solution, while the adjoint-error correction is an improvement of the functional only. Moreover, defect-correction needs the design and derivatives of high-order approximation residual (that is complicated and computationally expensive for unstructured grids) and this task usually must be accomplished with extra development of the flow solver (to increase the order of the discretization). To the opposite, adjoint-correction can be easily developed with AD tools (see Chapter 3) working almost exclusively with the existing solver (i.e. we don't need to modify the solver itself, but we use it as building-block to obtain its adjoint state).

However, adjoint error correction and defect correction are not mutually exclusive; the best accuracy is to be achieved through the simultaneous use of both techniques [Giles and Pierce, 2002].

Once a correction term involving an adjoint state is added to the original functional $J(\gamma, W)$, the development of its gradient for optimization may appear much more complex respect to the not-corrected case (as we have done in Chapter 3, see eq. (3.17)), therefore in this chapter we want to address the issues related to the evaluation of gradient for the corrected functional (5.1).

5.1 Adjoint error correction

Suppose that we want to evaluate a nonlinear functional $J\colon (\gamma, W) \mapsto J(\gamma, W)$ at the values $(\gamma, W_{\text{ex}}(\gamma))$, where γ is a control variable and $W_{\text{ex}}(\gamma)$ is the solution of the nonlinear equation

$$\Psi_{\text{ex}}(\gamma, W) = 0.$$

Given an *approximate* equation[1] $\Psi_h = \Psi(\gamma, W) = 0$, we define $W_h(\gamma)$ as the solution for $\Psi = 0$ and therefore an approximate solution for $\Psi_{\text{ex}} = 0$.

If we define the solution error for the problem $\Psi_{\text{ex}} = 0$ to be δW

$$\delta W := W_h - W_{\text{ex}},$$

and then linearize both the nonlinear equations and the functional around the true solution W_{ex}, we obtain

$$\Psi_{\text{ex}}(\gamma, W_h) = \Psi_{\text{ex}}(\gamma, W_{\text{ex}} + \delta W) \simeq \left.\frac{\partial \Psi_{\text{ex}}}{\partial W}\right|_{(\gamma, W_{\text{ex}})} \delta W$$

and

$$J(\gamma, W_h) = J(\gamma, W_{\text{ex}} + \delta W) = J(\gamma, W_{\text{ex}}) + \left.\frac{\partial J}{\partial W}\right|_{(\gamma, W_{\text{ex}})} \delta W + O\big(||\delta W||^2\big).$$

These can be rewritten as

$$\begin{cases} \left.\dfrac{\partial \Psi_{\text{ex}}}{\partial W}\right|_{(\gamma, W_{\text{ex}})} \delta W \simeq \Psi_{\text{ex}}(\gamma, W_h) \\ J(\gamma, W_{\text{ex}}) = J(\gamma, W_h) - \left.\dfrac{\partial J}{\partial W}\right|_{(\gamma, W_{\text{ex}})} \delta W + O\big(||\delta W||^2\big) \end{cases}$$

[1]In the sequel, to maintain the formalism as readable as possible, we omit the subscript h for the approximate equation $\Psi_h(\gamma, W) = 0$. In this way, the notation will be consistent with the one used in Chapter 3.

If Π_{ex} is defined as solving the adjoint equation

$$\left(\frac{\partial \Psi_{\text{ex}}}{\partial W}\bigg|_{(\gamma, W_{\text{ex}})}\right)^* \Pi_{\text{ex}} = \left(\frac{\partial J}{\partial W}\bigg|_{(\gamma, W_{\text{ex}})}\right)^* \tag{5.2}$$

then we obtain

$$J(\gamma, W_{\text{ex}}) = J(\gamma, W_h) - \big\langle \Pi_{\text{ex}}, \Psi_{\text{ex}}(\gamma, W_h)\big\rangle + O\big(||\delta W||^2\big).$$

Now we note that the correction term $\big\langle \Pi_{\text{ex}}, \Psi_{\text{ex}}(\gamma, W_h)\big\rangle$ correspond to the $\frac{\partial J}{\partial W}\big|_{(\gamma, W_{\text{ex}})}\delta W$ term in the Taylor expansion of the exact value $J(\gamma, W_{\text{ex}})$, so we can say that the corrected functional

$$J(\gamma, W_h) - \big\langle \Pi_{\text{ex}}, \Psi_{\text{ex}}(\gamma, W_h)\big\rangle \tag{5.3}$$

is a more accurate estimate for the true (and unknown) $J(\gamma, W_{\text{ex}})$ respect to $J(\gamma, W_h)$ alone.

It is important to note that the adjoint state Π_{ex} in (5.3) is referred to the true solution W_{ex} and not to the approximate solution W_h, in fact the linear adjoint operators $\big(\frac{\partial \Psi}{\partial W}\big)^*$ and $\big(\frac{\partial J}{\partial W}\big)^*$ in (5.2) are both evaluated at the state W_{ex}. For this reason, the true solution W_{ex} being unknown, the true adjoint state Π_{ex} is unknown as well.

In [Pierce and Giles, 2004], it is shown that in the corrected functional (5.3) we can use the approximated adjoint state Π_h referred to the approximate equation $\Psi = 0$ and its solution W_h, namely

$$\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h, W_h)}\right)^* \Pi_h = \left(\frac{\partial J}{\partial W}\bigg|_{(\gamma_h, W_h)}\right)^*, \tag{5.4}$$

due to the fact that the error on the adjoint $\langle \Pi_h - \Pi_{\text{ex}}, \Psi_{\text{ex}}(\gamma_h, W_h)\rangle$ is of higher order with respect to the correction $\big\langle \Pi_h, \Psi_{\text{ex}}(\gamma_h, W_h)\big\rangle$. To be more precise, if the solution errors for the nonlinear primal problem (i.e. $W_h - W_{\text{ex}}$) and the solution errors for the linear adjoint problem (i.e. $\Pi_h - \Pi_{\text{ex}}$) are of the same order, and they are both sufficiently smooth that the corresponding residual errors are also of the same order, then the order of accuracy of the functional approximations after making the adjoint correction is twice the order of the primal and adjoint solution [Pierce and Giles, 2004].

Summarizing, we can obtain a better estimate for the true value of the functional $J(\gamma, W_{\text{ex}})$ using, instead of the estimate $J(\gamma_h, W_h)$ for the original functional

$$J\colon (\gamma, W) \mapsto J(\gamma, W) \quad \text{with } W\colon \gamma \mapsto W(\gamma) \text{ such that } \Psi(\gamma, W) = 0,$$

the estimate $J^{\text{c}}(\gamma_h, W_h, \Pi_h)$ for the corrected functional

$$J^{\text{c}}\colon (\gamma, W, \Pi) \mapsto J^{\text{c}}(\gamma, W, \Pi) := J(\gamma, W) - \big\langle \Pi, \Psi_{\text{ex}}(\gamma, W)\big\rangle \tag{5.5}$$

subject to

$$\begin{cases} W\colon \gamma \mapsto W(\gamma) \text{ such that } \Psi(\gamma, W) = 0 \\ \Pi\colon (\gamma, W) \mapsto \Pi(\gamma, W) \quad \text{such that } \left(\dfrac{\partial \Psi}{\partial W}\bigg|_{(\gamma, W)}\right)^* \Pi = \left(\dfrac{\partial J}{\partial W}\bigg|_{(\gamma, W)}\right)^* \end{cases} . \tag{5.6}$$

Remark 5.1. It is important to note that the quantities needed for the estimate $J^{\mathrm{c}}(\gamma_h, W_h, \Pi_h)$ are *computables*, in fact the approximate (and continuous) primal and dual solutions W_h and Π_h might be created as P1 interpolation through computed (and discrete) values at grid nodes obtained, for example, solving the approximate problem $\Psi(\gamma, W) = 0$ with a finite volumes method and the corrispondent adjoint system (5.4).

Remark 5.2. Due to the fact that the variable $W(\gamma)$ depends on the control γ through the state equation $\Psi(\gamma, W(\gamma)) = 0$ (and the same thing holds for the adjoint state Π), we can consider the functionals $J(\gamma, W)$ and $J^{\mathrm{c}}(\gamma, W, \Pi)$ as functions of γ only, namely

$$j \colon \gamma \mapsto j(\gamma) := J(\gamma, W(\gamma)) \quad \text{with } W(\gamma) \text{ such that } \psi(\gamma) := \Psi(\gamma, W(\gamma)) = 0 \tag{5.7}$$

and

$$j^{\mathrm{c}} \colon \gamma \mapsto j^{\mathrm{c}}(\gamma) := J(\gamma, W(\gamma)) - \big\langle \pi(\gamma), \Psi_{\mathrm{ex}}(\gamma, W(\gamma)) \big\rangle \tag{5.8}$$

subject to

$$\begin{cases} W \colon \gamma \mapsto W(\gamma) \text{ such that } \psi(\gamma) := \Psi(\gamma, W(\gamma)) = 0 \\ \pi \colon \gamma \mapsto \pi(\gamma) := \Pi(\gamma, W(\gamma)) \quad \text{such that } \left(\left. \dfrac{\partial \Psi}{\partial W} \right|_{(\gamma, W(\gamma))} \right)^{*} \pi = \left(\left. \dfrac{\partial J}{\partial W} \right|_{(\gamma, W(\gamma))} \right)^{*} \end{cases} . \tag{5.9}$$

Remark 5.3. The difference between the W, Π and the corresponding quantities with a subscript W_h, Π_h (or W_{ex}, Π_{ex}) is that W, Π are *formal arguments* of the various functions and express a dependancy, while W_h, Π_h (or W_{ex}, Π_{ex}) are the *solutions* of the nonlinear constraint $\Psi = 0$ ($\Psi_{\mathrm{ex}} = 0$) and its associated adjoint equation. In this sense $j^{\mathrm{c}}(\gamma) = J^{\mathrm{c}}(\gamma, W, \Pi)$ is a *functional* (therefore the question about how compute its derivatives is well-posed), while $J^{\mathrm{c}}(\gamma_h, W_h, \Pi_h)$ is a *real value.*

The rest of the chapter it is devoted to the computation of the gradient $\left(\frac{dj^{\mathrm{c}}}{d\gamma}\big|_{\gamma} \right)^{*}$ in order to build a gradient-based algorithms for the optimization problem

$$\text{Find } \gamma_{\mathrm{opt}} \in \mathcal{G} \text{ such that } j^{\mathrm{c}}(\gamma_{\mathrm{opt}}) = \min_{\gamma \in \mathcal{G}} j^{\mathrm{c}}(\gamma)$$

where the functional $j^{\mathrm{c}}(\gamma)$ is defined as in (5.8) and subject to the constraints (5.9).

5.2 Gradient of the corrected functional

In Chapter 3 we have seen that to obtain the first-order derivatives of a functional $j(\gamma) = J(\gamma, W)$ respect to the control variable γ and such that the state variable $W(\gamma)$ satisfies the nonlinear constraint $\Psi(\gamma, W) = 0$, we can use an approach based on the adjoint formulation that gives

$$\left(\left. \frac{dj}{d\gamma} \right|_{\gamma} \right)^{*} = \left(\left. \frac{\partial J}{\partial \gamma} \right|_{(\gamma, W)} \right)^{*} - \left(\left. \frac{\partial \Psi}{\partial \gamma} \right|_{(\gamma, W)} \right)^{*} \Pi \tag{5.10}$$

where $\Pi\colon (\gamma, W) \mapsto \Pi(\gamma, W)$ is the function of γ and W satisfying the adjoint equation

$$\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma,W)}\right)^{*}\Pi = \left(\frac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)^{*}. \tag{5.11}$$

The adjoint-corrected functional (5.5) can be rewritten as

$$j^{\mathrm{c}}(\gamma) = J^{\mathrm{c}}(\gamma, W, \Pi) = J(\gamma, W) + F(\gamma, W, \Pi) \tag{5.12}$$

where we have introduced the functional $F\colon (\gamma, W, \Pi) \mapsto F(\gamma, W, \Pi)$ that corresponds to the correction term $\langle \Pi, \Psi_{\mathrm{ex}}(W)\rangle$, namely $F(\gamma_h, W_h, \Pi_h) = -\langle \Pi_h, \Psi_{\mathrm{ex}}(W_h)\rangle$. Then, due to the linearity of the differential operator $\frac{d}{d\gamma}$ and using the chain rule and the properties of the adjoints operators $(AB)^* = B^*A^*$ on the function f such that $f(\gamma) = F(\gamma, W(\gamma), \Pi(\gamma, W(\gamma)))$, we have

$$\boxed{\begin{aligned}\left(\frac{dj^{\mathrm{c}}}{d\gamma}\bigg|_{\gamma}\right)^{*} &= \left(\frac{dj}{d\gamma}\bigg|_{\gamma}\right)^{*} + \left(\frac{df}{d\gamma}\bigg|_{\gamma}\right)^{*}\\ &= \left(\frac{dj}{d\gamma}\bigg|_{\gamma}\right)^{*} + \left(\frac{\partial F}{\partial \gamma}\bigg|_{(\gamma,W,\Pi)}\right)^{*} + \left(\frac{dW}{d\gamma}\bigg|_{\gamma}\right)^{*}\left(\frac{\partial F}{\partial W}\bigg|_{(\gamma,W,\Pi)}\right)^{*} +\\ &\quad + \left(\frac{d\Pi}{d\gamma}\bigg|_{(\gamma,W,\Pi)}\right)^{*}\left(\frac{\partial F}{\partial \Pi}\bigg|_{(\gamma,W,\Pi)}\right)^{*}\end{aligned}} \tag{5.13}$$

where the first term $\left(\frac{dj}{d\gamma}\big|_{\gamma}\right)^{*}$ is the gradient of the original functional and could be computed firstly solving the adjoint system (5.11) and then using the (5.10), in the same way we have done in Section 3.5.2.

An important remark is that the derivative of the adjoint state $\left(\frac{d\Pi}{d\gamma}\big|_{(\gamma,W,\Pi)}\right)^{*}$ needs second-order derivatives in order to be evaluated (see Section 5.4).

It is interesting to note that the result (5.13) could be obtained from another point of view. In the case of the original (uncorrected) functional, the dependencies was on the design variable γ and on the state variable W and this last variable (that depends on γ) is defined implicitly through the state equation $\Psi(\gamma, W) = 0$. In the case of the corrected functional, we have the additional dependancy on the adjoint state Π (that, to be more precise, should be written like $\Pi(\gamma, W)$, i.e. a function of γ and W) obtained solving the adjoint equation (5.11). In this way, we can consider the variable Π as another state variable for the corrected functional, and we can say that its dependencies are, other than the control γ, on the *extended state variable* $\hat{W} = (W, \Pi)$ where this last variable is defined implicitly through the *approximate extended state equation*

$$\hat{\Psi}(\gamma, \hat{W}) := \begin{pmatrix}\Psi_s(\gamma, W)\\ \Psi_a(\gamma, W, \Pi)\end{pmatrix} = \begin{pmatrix}\Psi(\gamma, W)\\ \left(\frac{\partial \Psi}{\partial W}\big|_{(\gamma,W)}\right)^{*}\Pi - \left(\frac{\partial J}{\partial W}\big|_{(\gamma,W)}\right)^{*}\end{pmatrix} = \begin{pmatrix}0\\0\end{pmatrix}. \tag{5.14}$$

In other words, for the corrected functional, Π becomes a state variable together with the original state variable W, and therefore the adjoint system becomes an additional equality constraint to be satisfied.

From the definition above, $\Psi_s(\gamma, W) = 0$ correspond tho the original state equation $\Psi(\gamma, W) = 0$ and $\Psi_a(\gamma, W, \Pi) = 0$ correspond to the original adjoint equation (5.11). With the notation introduced in the previous section, we say that $\hat{W}_h = (W_h, \Pi_h)$ is the solution of the approximate extended state equation (5.14). Since $\Psi_s(\cdot)$ does not depends on Π, the function $W(\gamma)$ can be first computed independently from Π (solving $\Psi(\gamma, W) = 0$) and then $\Pi(\gamma, W)$ (solving the adjoint equation $\Psi_a(\gamma, W, \Pi) = 0$).

With these definitions, we can apply the adjoint approach to the corrected functional J^c and to the extended state function $\hat{\Psi}$ (where for the original case we used J and Ψ) and obtain

$$\left(\left.\frac{dj^{\mathrm{c}}}{d\gamma}\right|_{\gamma}\right)^* = \left(\left.\frac{\partial J^{\mathrm{c}}}{\partial \gamma}\right|_{(\gamma,\hat{W})}\right)^* - \left(\left.\frac{\partial \hat{\Psi}}{\partial \gamma}\right|_{(\gamma,\hat{W})}\right)^* \hat{\Pi} \tag{5.15}$$

where the *extended adjoint state* $\hat{\Pi}\colon (\gamma, \hat{W}) \mapsto \hat{\Pi}(\gamma, \hat{W})$ satisfies the new adjoint equation

$$\left(\left.\frac{\partial \hat{\Psi}}{\partial \hat{W}}\right|_{(\gamma,\hat{W})}\right)^* \hat{\Pi} = \left(\left.\frac{\partial J^{\mathrm{c}}}{\partial \hat{W}}\right|_{(\gamma,\hat{W})}\right)^* . \tag{5.16}$$

To show the equivalence of the two different approaches introduced above, we need the following lemma

Lemma 5.1. *Let* $W : \gamma \mapsto W(\gamma)$ *and* $\Pi\colon (\gamma, W) \mapsto \Pi(\gamma, W)$ *functions such that verify the equation* $\Psi_a(\gamma, W, \Pi) = 0$ *where*

$$\Psi_a\colon (\gamma, W, \Pi) \mapsto \left(\left.\frac{\partial \Psi}{\partial W}\right|_{(\gamma,W)}\right)^* \Pi - \left(\left.\frac{\partial J}{\partial W}\right|_{(\gamma,W)}\right)^*$$

and $\Psi\colon (\gamma, W) \mapsto \Psi(\gamma, W)$*. Then holds*

$$\left(\left.\frac{d\Pi}{d\gamma}\right|_{(\gamma,W)}\right)^* = -\left[\left(\left.\frac{dW}{d\gamma}\right|_{\gamma}\right)^* \left(\left.\frac{\partial \Psi_a}{\partial W}\right|_{(\gamma,W,\Pi)}\right)^* + \left(\left.\frac{\partial \Psi_a}{\partial \gamma}\right|_{(\gamma,W,\Pi)}\right)^*\right] \left(\left.\frac{\partial \Psi}{\partial W}\right|_{(\gamma,W)}\right)^{-1} . \tag{5.17}$$

Proof. Due to the dependency of W and Π on γ ($\Pi(\gamma, W) = \Pi(\gamma, W(\gamma)) = \pi(\gamma)$) through the implicit equation $\Psi_a(\gamma, W, \Pi) = 0$, we can consider Ψ_a depending solely on γ, namely $\Psi_a(\gamma, W, \Pi) = \psi_a(\gamma) = 0$ and therefore holds $\left.\frac{d\psi_a}{d\gamma}\right|_{\gamma} = 0$. Using the chain rule we have

$$\begin{aligned} 0 &= \left(\left.\frac{\partial \Psi_a}{\partial \gamma}\right|_{(\gamma,W,\Pi)}\right) + \left(\left.\frac{\partial \Psi_a}{\partial W}\right|_{(\gamma,W,\Pi)}\right)\left(\left.\frac{dW}{d\gamma}\right|_{\gamma}\right) + \left(\left.\frac{\partial \Psi_a}{\partial \Pi}\right|_{(\gamma,W,\Pi)}\right)\left(\left.\frac{d\Pi}{d\gamma}\right|_{(\gamma,W)}\right) \\ &= \left(\left.\frac{\partial \Psi_a}{\partial \gamma}\right|_{(\gamma,W,\Pi)}\right) + \left(\left.\frac{\partial \Psi_a}{\partial W}\right|_{(\gamma,W,\Pi)}\right)\left(\left.\frac{dW}{d\gamma}\right|_{\gamma}\right) + \left(\left.\frac{\partial \Psi}{\partial W}\right|_{(\gamma,W)}\right)^*\left(\left.\frac{d\Pi}{d\gamma}\right|_{(\gamma,W)}\right) \end{aligned}$$

where we used the fact that the function Ψ_a is linear in Π.

Rearranging the terms in the last equation and using the property of the adjoint operators $(AB)^* = B^*A^*$ and $(A^*)^* = A$, we obtain

$$\left(\left.\frac{d\Pi}{d\gamma}\right|_{(\gamma,W)}\right)^* \left(\left.\frac{\partial \Psi}{\partial W}\right|_{(\gamma,W)}\right) = -\left(\left.\frac{dW}{d\gamma}\right|_{\gamma}\right)^* \left(\left.\frac{\partial \Psi_a}{\partial W}\right|_{(\gamma,W,\Pi)}\right)^* - \left(\left.\frac{\partial \Psi_a}{\partial \gamma}\right|_{(\gamma,W,\Pi)}\right)^*$$

and right-multiplying both sides of the previous equation by $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma,W)}\right)^{-1}$, the (5.17) is verified. $\square$

To show the equivalence of the (5.11)–(5.13) and the (5.14)–(5.16) we write the extended adjoint state $\hat{\Pi} = \begin{pmatrix} \Pi_s \\ \Pi_a \end{pmatrix}$ where the subscript are referred accordingly with the definition (5.14). Differentiating the (5.14) and the (5.12) we obtain

$$\begin{cases} \left(\dfrac{\partial \hat{\Psi}}{\partial \hat{W}}\bigg|_{(\gamma,\hat{W})}\right)^* = \begin{pmatrix} \left(\dfrac{\partial \Psi_s}{\partial W}\bigg|_{(\gamma,W)}\right)^* & \left(\dfrac{\partial \Psi_a}{\partial W}\bigg|_{(\gamma,\hat{W})}\right)^* \\ 0 & \left(\dfrac{\partial \Psi_a}{\partial \Pi}\bigg|_{(\gamma,\hat{W})}\right)^* \end{pmatrix} \\ \left(\dfrac{\partial \hat{\Psi}}{\partial \gamma}\bigg|_{(\gamma,\hat{W})}\right)^* = \left(\left(\dfrac{\partial \Psi_s}{\partial \gamma}\bigg|_{(\gamma,W)}\right)^*, \left(\dfrac{\partial \Psi_a}{\partial \gamma}\bigg|_{(\gamma,\hat{W})}\right)^*\right) \end{cases}$$

and

$$\begin{cases} \left(\dfrac{\partial J^{\mathrm{c}}}{\partial \hat{W}}\bigg|_{(\gamma,\hat{W})}\right)^* = \begin{pmatrix} \left(\dfrac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)^* + \left(\dfrac{\partial F}{\partial W}\bigg|_{(\gamma,\hat{W})}\right)^* \\ \left(\dfrac{\partial F}{\partial \Pi}\bigg|_{(\gamma,\hat{W})}\right)^* \end{pmatrix} \\ \left(\dfrac{\partial J^{\mathrm{c}}}{\partial \gamma}\bigg|_{(\gamma,\hat{W})}\right)^* = \left(\dfrac{\partial J}{\partial \gamma}\bigg|_{(\gamma,W)}\right)^* + \left(\dfrac{\partial F}{\partial \gamma}\bigg|_{(\gamma,\hat{W})}\right)^* \end{cases}$$

Due to the adjoint equation (5.16) and the block-triangular nature of the Jacobian matrix $\left(\frac{\partial \hat{\Psi}}{\partial \hat{W}}\right)^*$, we observe that Π_a can be computed first, then Π_s. In fact the system (5.16) writes

$$\begin{cases} \left(\dfrac{\partial \Psi_s}{\partial W}\bigg|_{(\gamma,W)}\right)^* \Pi_s + \left(\dfrac{\partial \Psi_a}{\partial W}\bigg|_{(\gamma,W,\Pi)}\right)^* \Pi_a = \left(\dfrac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)^* + \left(\dfrac{\partial F}{\partial W}\bigg|_{(\gamma,W,\Pi)}\right)^* \\ \left(\dfrac{\partial \Psi_a}{\partial \Pi}\bigg|_{(\gamma,W,\Pi)}\right)^* \Pi_a = \left(\dfrac{\partial F}{\partial \Pi}\bigg|_{(\gamma,W,\Pi)}\right)^* \end{cases}$$

and we can compute the extended adjoint state as

$$\hat{\Pi} = \begin{pmatrix} \Pi_s \\ \Pi_a \end{pmatrix} = \begin{pmatrix} \left(\dfrac{\partial \Psi_s}{\partial W}\bigg|_{(\gamma,W)}\right)^{-*} \Bigg[\left(\dfrac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)^* + \left(\dfrac{\partial F}{\partial W}\bigg|_{(\gamma,W,\Pi)}\right)^* + \\ \qquad - \left(\dfrac{\partial \Psi_a}{\partial W}\bigg|_{(\gamma,W,\Pi)}\right)^* \left(\dfrac{\partial \Psi_a}{\partial \Pi}\bigg|_{(\gamma,W,\Pi)}\right)^{-*} \left(\dfrac{\partial F}{\partial \Pi}\bigg|_{(\gamma,W,\Pi)}\right)^* \Bigg] \\ \left(\dfrac{\partial \Psi_a}{\partial \Pi}\bigg|_{(\gamma,W,\Pi)}\right)^{-*} \left(\dfrac{\partial F}{\partial \Pi}\bigg|_{(\gamma,W,\Pi)}\right)^* \end{pmatrix}.$$

Therefore, remembering that $\Psi_s := \Psi$, the equation (5.15) becomes

$$\begin{aligned}
\left(\frac{dj^c}{d\gamma}\bigg|_\gamma\right)^* &= \left(\frac{\partial J^c}{\partial\gamma}\bigg|_{(\gamma,W,\Pi)}\right)^* - \left(\frac{\partial\hat{\Psi}}{\partial\gamma}\bigg|_{(\gamma,W,\Pi)}\right)^*\hat{\Pi} \\
&= \left(\frac{\partial J}{\partial\gamma}\bigg|_{(\gamma,W)}\right)^* + \left(\frac{\partial F}{\partial\gamma}\bigg|_{(\gamma,W,\Pi)}\right)^* + \\
&\quad - \left\{\left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma,W)}\right)^*\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma,W)}\right)^{-*}\left[\left(\frac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)^* + \left(\frac{\partial F}{\partial W}\bigg|_{(\gamma,W,\Pi)}\right)^* + \right.\right. \\
&\quad\quad \left. - \left(\frac{\partial\Psi_a}{\partial W}\bigg|_{(\gamma,W,\Pi)}\right)^*\left(\frac{\partial\Psi_a}{\partial\Pi}\bigg|_{(\gamma,W,\Pi)}\right)^{-*}\left(\frac{\partial F}{\partial\Pi}\bigg|_{(\gamma,W,\Pi)}\right)^*\right] + \\
&\quad\quad \left. + \left(\frac{\partial\Psi_a}{\partial\gamma}\bigg|_{(\gamma,W,\Pi)}\right)^*\left(\frac{\partial\Psi_a}{\partial\Pi}\bigg|_{(\gamma,W,\Pi)}\right)^{-*}\left(\frac{\partial F}{\partial\Pi}\bigg|_{(\gamma,W,\Pi)}\right)^*\right\}
\end{aligned}$$

The adjoint equation (5.11) is linear respect to Π so the equivalence $\left(\frac{\partial\Psi_a}{\partial\Pi}\right)^* = \frac{\partial\Psi}{\partial W}$ holds. Moreover, from $\frac{d\Psi}{d\gamma} = 0$ we can express the derivative of the state variable W respect to the control γ as

$$\frac{dW}{d\gamma}\bigg|_\gamma = -\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma,W)}\right)^{-1}\left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma,W)}\right)$$

and finally we can write

$$\begin{aligned}
\left(\frac{dj^c}{d\gamma}\bigg|_\gamma\right)^* &= \left(\frac{\partial J}{\partial\gamma}\bigg|_{(\gamma,W)}\right)^* + \left(\frac{\partial F}{\partial\gamma}\bigg|_{(\gamma,W,\Pi)}\right)^* + \left\{\left(\frac{dW}{d\gamma}\bigg|_\gamma\right)^*\left[\left(\frac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)^* + \right.\right. \\
&\quad \left. + \left(\frac{\partial F}{\partial W}\bigg|_{(\gamma,W,\Pi)}\right)^* - \left(\frac{\partial\Psi_a}{\partial W}\bigg|^*_{(\gamma,W,\Pi)}\right)\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma,W)}\right)^{-1}\left(\frac{\partial F}{\partial\Pi}\bigg|_{(\gamma,W,\Pi)}\right)^*\right] + \\
&\quad \left. - \left(\frac{\partial\Psi_a}{\partial\gamma}\bigg|_{(\gamma,W,\Pi)}\right)^*\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma,W)}\right)^{-1}\left(\frac{\partial F}{\partial\Pi}\bigg|_{(\gamma,W,\Pi)}\right)^*\right\}
\end{aligned}$$

and after some rearrangements we obtain

$$\begin{aligned}
\left(\frac{dj^c}{d\gamma}\bigg|_\gamma\right)^* &= \left(\frac{dj}{d\gamma}\bigg|_\gamma\right)^* + \left(\frac{\partial F}{\partial\gamma}\bigg|_{(\gamma,W,\Pi)}\right)^* + \left(\frac{dW}{d\gamma}\bigg|_\gamma\right)^*\left(\frac{\partial F}{\partial W}\bigg|_{(\gamma,W,\Pi)}\right)^* + \\
&\quad - \left[\left(\frac{dW}{d\gamma}\bigg|_\gamma\right)^*\left(\frac{\partial\Psi_a^*}{\partial W}\bigg|_{(\gamma,W,\Pi)}\right) + \left(\frac{\partial\Psi_a}{\partial\gamma}\bigg|_{(\gamma,W,\Pi)}\right)^*\right]\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma,W)}\right)^{-1}\left(\frac{\partial F}{\partial\Pi}\bigg|_{(\gamma,W,\Pi)}\right)^*
\end{aligned} \tag{5.18}$$

that is equal to the (5.13) by the Lemma 5.1.

5.3 Algorithm for computing the gradient of the adjoint-corrected functional

Despite the fact that the two approaches presented in the Section 5.2 are theoretically equivalents, their implementations in the discrete case with the AD tools are significantly differents.

The first approach is based on the definition a new scalar function $f(\gamma) = F(\gamma, W, \Pi)$ that acts as an additive correction for the original functional $j(\gamma)$; in this sense (by the linearity of the differential operator $\frac{d}{d\gamma}$) the resulting gradient $\left(\frac{dj^{\mathrm{c}}}{d\gamma}\big|_{\gamma}\right)^*$ is considered to be the sum of two different contributions (eq. 5.13):

- the gradient of the original (not-corrected) functional, namely $\left(\frac{dj}{d\gamma}\bigg|_{\gamma}\right)^*$ and
- the term $\left(\frac{df}{d\gamma}\bigg|_{\gamma}\right)^*$ that is the gradient of the adjoint correction $-\big\langle \Pi, \Psi_{\mathrm{ex}}(\gamma, W)\big\rangle$.

Thus, the algorithm for the first approach could be splitted in two part: the first one is devoted to the computation of $\left(\frac{dj}{d\gamma}\big|_{\gamma}\right)^*$ (and this task requires only first-order derivatives, see Section 3.5.2 for details) and the second one is the computation of $\left(\frac{df}{d\gamma}\big|_{\gamma}\right)^*$ (requiring, as we will see in the Section 5.4, second-order derivatives).

The second approach is based on the definition of a new (extended) state function $\hat{\Psi}(\gamma, \hat{W})$ and considering the original adjoint state Π as part of the new state variable $\hat{W}$, and then computing the gradient $\left(\frac{dj^{\mathrm{c}}}{d\gamma}\big|_{\gamma}\right)^*$ considering the corrected functional $j^{\mathrm{c}}(\gamma)$ as a whole. The difficulties here rely on the fact that the extended state functions contain a routine that is differentiated in Reverse-mode, namely the Reverse-mode differentiation of the $\hat{\Psi}(\gamma, \hat{W})$, that is required for the computation of the extended adjoint state $\hat{\Pi}$ (solution of the equation (5.16)). The Reverse differentiation of the routine implementing such extended state function, results in a differentiation mode (Reverse-on-Reverse), which is not studied yet.

For the above reasons we implemented the first approach, which a possible algorithm is summarized as follows (see also Fig. 5.1):

1. compute the state W_h such that $\Psi(\gamma_h, W_h) = 0$;
2. compute $\bar{W}_J = \left(\frac{\partial J}{\partial W}\bigg|_{(\gamma_h, W_h)}\right)^*$
3. compute the adjoint Π_h solving the linear system $\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h, W_h)}\right)^* \Pi_h = \bar{W}_J$;
4. compute the gradient $\left(\frac{dj}{d\gamma}\bigg|_{\gamma}\right)^*$ with the following steps:
   (a) compute $\bar{\gamma}_J = \left(\frac{\partial J}{\partial \gamma}\bigg|_{(\gamma_h, W_h)}\right)^*$;
   (b) compute $\bar{\gamma}_\Psi = \left(\frac{\partial \Psi}{\partial \gamma}\bigg|_{(\gamma_h, W_h)}\right)^* \Pi_h$;

   (c) evaluate $\left(\frac{dj}{d\gamma}\Big|_{\gamma}\right)^* = \bar{\gamma}_J - \bar{\gamma}_\Psi$;

5. compute $\bar{\gamma}_F = \left(\frac{\partial F}{\partial \gamma}\Big|_{(\gamma_h, W_h, \Pi_h)}\right)^*$, $\bar{W}_F = \left(\frac{\partial F}{\partial W}\Big|_{(\gamma_h, W_h, \Pi_h)}\right)^*$ and $\bar{\Pi}_F = \left(\frac{\partial F}{\partial \Pi}\Big|_{(\gamma_h, W_h, \Pi_h)}\right)^*$;

6. compute $g = \left(\frac{dW}{d\gamma}\Big|_{\gamma}\right)^* \bar{W}_F$ with the following steps:

   (a) compute the vector l solving the linear system $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h, W_h)}\right)^* l = \bar{W}_F$;

   (b) compute $g = -\left(\frac{\partial \Psi}{\partial \gamma}\Big|_{(\gamma_h, W_h)}\right)^* l$

7. compute $m = \left(\frac{d\Pi}{d\gamma}\Big|_{(\gamma_h, W_h, \Pi_h)}\right)^* \bar{\Pi}_F$ (see the Section 5.4);

8. evaluate the gradient of the corrected functional as $\left(\frac{dj^{\mathrm{c}}}{d\gamma}\Big|_{\gamma}\right)^* = \left(\frac{dj}{d\gamma}\Big|_{\gamma}\right)^* + \bar{\gamma}_F + g + m$

Regarding the computational cost to obtain the gradient of the adjoint-corrected functional, we can see from the previous algorithm that the main contribution is due to the solution of two adjoint linear systems (step 3 to compute the adjoint state Π_h and step 6a) in addition to the evaluation of the term $\left(\frac{d\Pi}{d\gamma}\right)^* \bar{\Pi}_F$, which cost will be estimated in the next section.

If we apply the matrix-free strategy defined in Chapter 3 and we use the same estimates for the overhead associated with the differentiated code (i.e. α_T for the Tangent mode differentiation and α_R for the Reverse mode) and for the number of iterations to the convergence ($n_{\mathrm{iter},R}$ iteration so solve and adjoint linear system), we have for step 3 and step 6a a runtime cost of $2\alpha_R n_{\mathrm{iter},R}$.

5.4 Gradient of the adjoint-correction term

In the previous section, we have seen that the algorithm to compute the gradient (respect to the control variable γ) of the corrected functional (5.12), involves the evaluation of the quantity $m = \left(\frac{d\Pi}{d\gamma}\Big|_{(\gamma_h, W_h)}\right)^* \bar{\Pi}_F$ (step 7), where the adjoint state Π is the function (of the control $\gamma \in \mathbb{R}^n$ and the state variables $W \in \mathbb{R}^N$) that solves the linear system $\left(\frac{\partial \Psi}{\partial W}\right)^* \Pi = \left(\frac{\partial J}{\partial W}\right)^*$. How can we perform this task using Automatic Differentiation?

In the demonstration of Lemma 3.1 on page 61, we have seen that $\lambda_h = \left(\frac{d\Pi}{d\gamma}\Big|_{(\gamma_h, W_h)}\right)\delta$ is the solution of the linear system

$$\begin{aligned}\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h, W_h)}\right)^* \lambda_h = \frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^*\Bigg|_{(\gamma_h, W_h)} \delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^*\Bigg|_{(\gamma_h, W_h)} \theta_h + \\ - \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial \Psi}{\partial W}\right)^* \Pi_h\right]\Bigg|_{(\gamma_h, W_h)} \delta - \frac{\partial}{\partial W}\left[\left(\frac{\partial \Psi}{\partial W}\right)^* \Pi_h\right]\Bigg|_{(\gamma_h, W_h)} \theta_h\end{aligned} \quad (5.19)$$

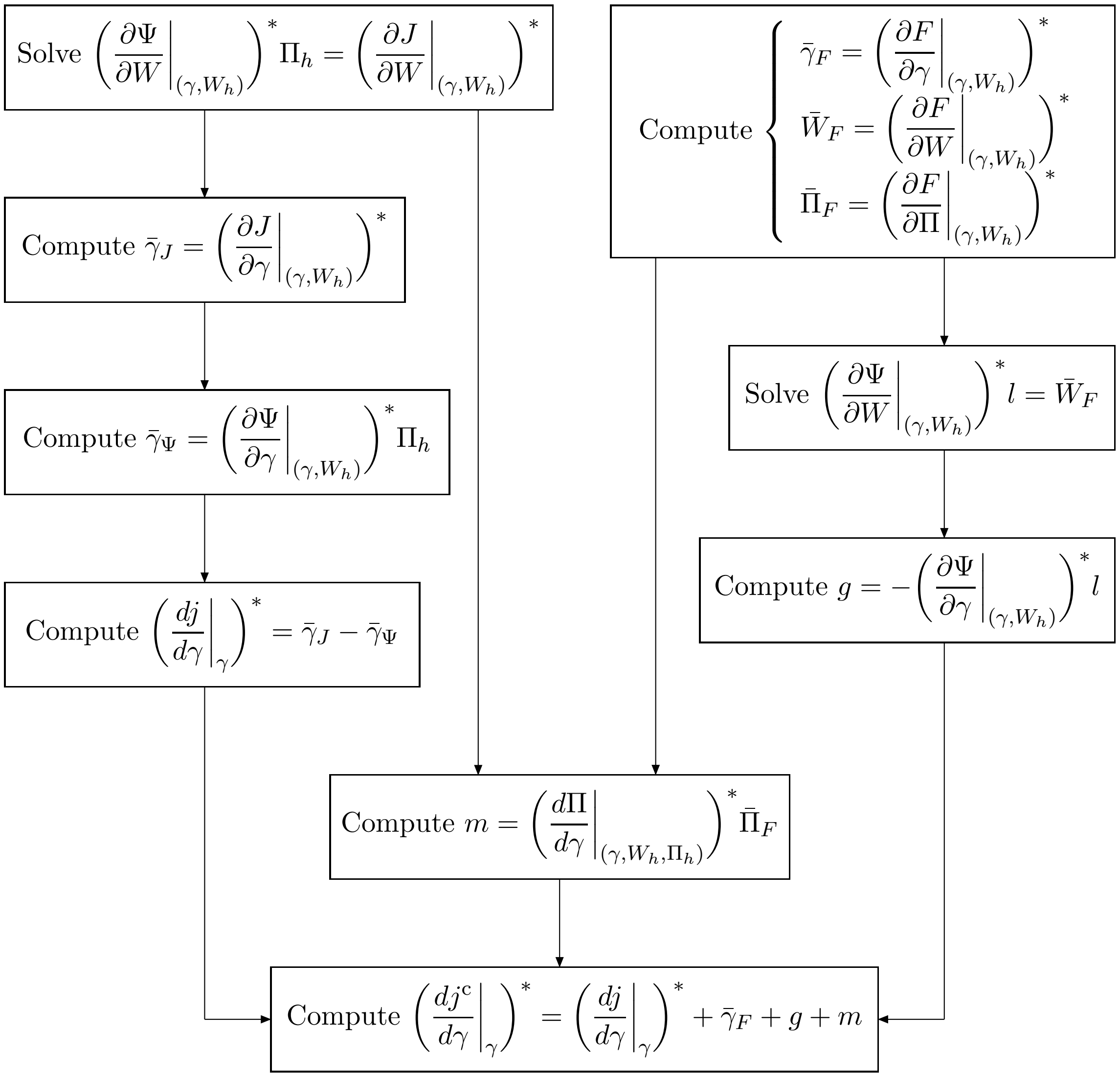

Solve $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma,W_h)}\right)^* \Pi_h = \left(\frac{\partial J}{\partial W}\Big|_{(\gamma,W_h)}\right)^*$
Compute $\bar{\gamma}_J = \left(\frac{\partial J}{\partial \gamma}\Big|_{(\gamma,W_h)}\right)^*$
Compute $\bar{\gamma}_\Psi = \left(\frac{\partial \Psi}{\partial \gamma}\Big|_{(\gamma,W_h)}\right)^* \Pi_h$
Compute $\left(\frac{dj}{d\gamma}\Big|_{\gamma}\right)^* = \bar{\gamma}_J - \bar{\gamma}_\Psi$
Compute $\begin{cases} \bar{\gamma}_F = \left(\frac{\partial F}{\partial \gamma}\Big|_{(\gamma,W_h)}\right)^* \\ \bar{W}_F = \left(\frac{\partial F}{\partial W}\Big|_{(\gamma,W_h)}\right)^* \\ \bar{\Pi}_F = \left(\frac{\partial F}{\partial \Pi}\Big|_{(\gamma,W_h)}\right)^* \end{cases}$
Solve $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma,W_h)}\right)^* l = \bar{W}_F$
Compute $g = -\left(\frac{\partial \Psi}{\partial \gamma}\Big|_{(\gamma,W_h)}\right)^* l$
Compute $m = \left(\frac{d\Pi}{d\gamma}\Big|_{(\gamma,W_h,\Pi_h)}\right)^* \bar{\Pi}_F$
Compute $\left(\frac{dj^{\mathrm{c}}}{d\gamma}\Big|_{\gamma}\right)^* = \left(\frac{dj}{d\gamma}\Big|_{\gamma}\right)^* + \bar{\gamma}_F + g + m$

Figure 5.1: The algorithm to compute the $\left(\frac{dJ^{\mathrm{c}}}{d\gamma}\big|_{(\gamma,W_h)}\right)^*$, the gradient of the corrected functional $j^{\mathrm{c}}(\gamma)$ (see Section 5.4).

and with $\theta_h \in \mathbb{R}^N$ solution of the linear system

$$\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)\theta_h = -\left(\frac{\partial \Psi}{\partial \gamma}\bigg|_{(\gamma_h,W_h)}\right)\delta \ . \tag{5.20}$$

Then, one possibility is computing the $N \times n$ matrix $\frac{d\Pi}{d\gamma}\big|_{(\gamma,W_h)}$ column-by-column using the relation $\lambda_i = \frac{d\Pi}{d\gamma}\big|_{(\gamma_h,W_h)} e_i$ (where $e_i \in \mathbb{R}^n$ is the column-vector with the value 1 at the i-th position and 0 otherwise). The drawback of this approach is that it requires the solution of $2n$ linear systems (5.19)–(5.20), so when $n \gg 1$ it will be prohibitively expensive.

The key idea here is to perform a transposition of the equation (5.19) in order to obtain an expression for λ_h^* (that is $\delta^*\big(\frac{d\Pi}{d\gamma}\big|_{(\gamma_h,W_h)}\big)^*$), and then reassembling the various terms in a way that permits us to compute the quantity $\big(\frac{d\Pi}{d\gamma}\big|_{(\gamma_h,W_h)}\big)^*\chi$ (where χ is a generic vector of $\mathbb{R}^N$) using the differentiation modes availables with the AD tools. It can be done considering the following lemma

Lemma 5.2. *If the following quantities are defined*

- $\eta_h = \left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{-1}\chi$
- $\Pi_h = \left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{-*}\left(\frac{\partial J}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{*}$
- $\bar{\gamma}_\xi = \left[\frac{\partial}{\partial \gamma}\big(\dot{J}_W - \langle \Pi_h, \dot{\Psi}_W\rangle\big)\right]\bigg|^*_{(\gamma_h,W_h)}$ *and* $\bar{W}_\xi = \left[\frac{\partial}{\partial W}\big(\dot{J}_W - \langle \Pi_h, \dot{\Psi}_W\rangle\big)\right]\bigg|^*_{(\gamma_h,W_h)}$ *where are defined the functions* $\dot{J}_W(\gamma,W) = \left(\frac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)\eta_h$ *and* $\dot{\Psi}_W(\gamma,W) = \left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma,W)}\right)\eta_h$
- $\varphi_h = \left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{-*}\bar{W}_\xi$

then holds

$$\left(\frac{d\Pi}{d\gamma}\bigg|_{(\gamma_h,W_h)}\right)^*\chi = \bar{\gamma}_\xi - \left(\frac{\partial \Psi}{\partial \gamma}\bigg|_{(\gamma_h,W_h)}\right)^*\varphi_h \ . \tag{5.21}$$

Proof. Let $a(\gamma,W) = \big(\frac{\partial J}{\partial W}\big|_{(\gamma,W)}\big)^*$ and $b(\gamma,W) = \big(\frac{\partial \Psi}{\partial W}\big|_{(\gamma,W)}\big)^*\Pi_h$, then the equation (5.19) becomes

$$\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^*\lambda = \frac{\partial a}{\partial \gamma}\bigg|_{(\gamma_h,W_h)}\delta + \frac{\partial a}{\partial W}\bigg|_{(\gamma_h,W_h)}\theta_h - \frac{\partial b}{\partial \gamma}\bigg|_{(\gamma_h,W_h)}\delta - \frac{\partial b}{\partial W}\bigg|_{(\gamma_h,W_h)}\theta_h$$

and remembering the properties of the trasposition $(AB)^* = B^*A^*$, we have

$$\lambda^*\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right) = \delta^*\left(\frac{\partial a^*}{\partial \gamma} - \frac{\partial b^*}{\partial \gamma}\right)\bigg|_{(\gamma_h,W_h)} + \theta_h^*\left(\frac{\partial a^*}{\partial W} - \frac{\partial b^*}{\partial W}\right)\bigg|_{(\gamma_h,W_h)}$$

Being θ_h the solution of the linear system $\big(\frac{\partial\Psi}{\partial W}\big|_{(\gamma_h,W_h)}\big)\theta = -\big(\frac{\partial\Psi}{\partial\gamma}\big|_{(\gamma_h,W_h)}\big)\delta$, we can express its adjoint as

$$\theta_h^* = -\delta^*\left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma_h,W_h)}\right)^*\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{-*}$$

then

$$\begin{aligned}\lambda_h^* = \delta^*\Bigg[\left(\frac{\partial a^*}{\partial\gamma}-\frac{\partial b^*}{\partial\gamma}\right)\bigg|_{(\gamma_h,W_h)} &+ \\ -\left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma_h,W_h)}\right)^*\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{-*}&\left(\frac{\partial a^*}{\partial W}-\frac{\partial b^*}{\partial W}\right)\bigg|_{(\gamma_h,W_h)}\Bigg]\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{-1}\end{aligned}$$

remembering that $\lambda_h^* = \delta^*\big(\frac{d\Pi}{d\gamma}\big|_{(\gamma_h,W_h)}\big)^*$, we can express the quantity $\big(\frac{d\Pi}{d\gamma}\big|_{(\gamma_h,W_h)}\big)^*\chi$ as

$$\begin{aligned}\left(\frac{d\Pi}{d\gamma}\bigg|_{(\gamma_h,W_h)}\right)^*\chi = \Bigg[\left(\frac{\partial a^*}{\partial\gamma}-\frac{\partial b^*}{\partial\gamma}\right)\bigg|_{(\gamma_h,W_h)} &+ \\ -\left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma_h,W_h)}\right)^*\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{-*}&\left(\frac{\partial a^*}{\partial W}-\frac{\partial b^*}{\partial W}\right)\bigg|_{(\gamma_h,W_h)}\Bigg]\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{-1}\chi\end{aligned} \tag{5.22}$$

From a computationally point of view, the last equation should be evaluated right-to-left (in order to use matrix-by-vector operation only) and the explicit computation of the inverse operators $\big(\frac{\partial\Psi}{\partial W}\big|_{(\gamma_h,W_h)}\big)^{-1}$ and $\big(\frac{\partial\Psi}{\partial W}\big|_{(\gamma_h,W_h)}\big)^{-*}$ can be avoided, considering the application of these inverse operator on a vector as the solution of a certain linear system. For the specific case, if we put $\eta_h = \big(\frac{\partial\Psi}{\partial W}\big|_{(\gamma_h,W_h)}\big)^{-1}\chi$ and $\varphi_h = \big(\frac{\partial\Psi}{\partial W}\big|_{(\gamma_h,W_h)}\big)^{-*}\big(\frac{\partial a^*}{\partial W}-\frac{\partial b^*}{\partial W}\big)\big|_{(\gamma_h,W_h)}\eta_h$, we can first compute η_h as the solution of the linear system $\big(\frac{\partial\Psi}{\partial W}\big|_{(\gamma_h,W_h)}\big)\eta_h = \chi$ and then φ_h as the solution of the adjoint linear system $\big(\frac{\partial\Psi}{\partial W}\big|_{(\gamma_h,W_h)}\big)^*\varphi_h = \big(\frac{\partial a^*}{\partial W}-\frac{\partial b^*}{\partial W}\big)\big|_{(\gamma_h,W_h)}\eta_h$. Therefore the equation (5.22) becomes

$$\left\{\begin{aligned}\eta_h &= \left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{-1}\chi \\ \varphi_h &= \left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^{-*}\left(\frac{\partial a^*}{\partial W}-\frac{\partial b^*}{\partial W}\right)\bigg|_{(\gamma_h,W_h)}\eta_h \\ \left(\frac{d\Pi}{d\gamma}\bigg|_{(\gamma_h,W_h)}\right)^*\chi &= \left(\frac{\partial a^*}{\partial\gamma}-\frac{\partial b^*}{\partial\gamma}\right)\bigg|_{(\gamma_h,W_h)}\eta_h - \left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma_h,W_h)}\right)^*\varphi_h\end{aligned}\right. \tag{5.23}$$

The next step is to replace $a(\gamma,W) = \big(\frac{\partial J}{\partial W}\big|_{(\gamma,W)}\big)^*$ and $b(\gamma,W) = \big(\frac{\partial\Psi}{\partial W}\big|_{(\gamma,W)}\big)^*\Pi_h$ into the equation (5.23) and evaluate them at the solution (γ_h,W_h). Due to the fact that η_h is a specific value (and not a function on γ or W) we can write

$$\left(\frac{\partial a}{\partial\gamma}\bigg|_{(\gamma,W)}\right)^*\eta_h = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)^*\right]^*\eta_h = \left[\frac{\partial}{\partial\gamma}\left(\left\langle\eta_h,\left(\frac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)^*\right\rangle\right)\right]^*$$

but $\langle \eta_h, \big(\frac{\partial J}{\partial W}\big|_{(\gamma,W)}\big)^*\rangle = \langle \big(\frac{\partial J}{\partial W}\big|_{(\gamma,W)}\big)\eta_h, 1\rangle = \big(\frac{\partial J}{\partial W}\big|_{(\gamma,W)}\big)\eta_h$ and this last term can be considered to be a real-valued functional of the variables γ and W, then we obtain

$$\left(\frac{\partial a}{\partial \gamma}\bigg|_{(\gamma_h,W_h)}\right)^* \eta_h = \left[\frac{\partial}{\partial \gamma}\left(\left(\frac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)\eta_h\right)\right]\Bigg|^*_{(\gamma_h,W_h)} = \left(\frac{\partial \dot{J}_W}{\partial \gamma}\bigg|_{(\gamma_h,W_h)}\right)^* \tag{5.24}$$

where $\dot{J}_W(\gamma, W) = \big(\frac{\partial J}{\partial W}\big|_{(\gamma,W)}\big)\eta_h$. The derivative involving the b function is analogous to the previous one, namely

$$\begin{aligned}\left(\frac{\partial b}{\partial \gamma}\bigg|_{(\gamma_h,W_h)}\right)^* \eta_0 &= \left[\frac{\partial}{\partial \gamma}\left(\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma,W)}\right)^* \Pi_h\right)\right]\Bigg|^*_{(\gamma_h,W_h)} \eta_h \\ &= \left[\frac{\partial}{\partial \gamma}\left(\langle \eta_h, \left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma,W)}\right)^* \Pi_h\rangle\right)\right]\Bigg|^*_{(\gamma_h,W_h)} \\ &= \left[\frac{\partial}{\partial \gamma}\left(\langle \Pi_h, \left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma,W)}\right) \eta_h\rangle\right)\right]\Bigg|^*_{(\gamma_h,W_h)} \\ &= \left[\frac{\partial}{\partial \gamma}\left(\langle \Pi_h, \dot{\Psi}_W\rangle\right)\right]\Bigg|^*_{(\gamma_h,W_h)}\end{aligned} \tag{5.25}$$

where $\dot{\Psi}_W(\gamma, W) = \big(\frac{\partial \Psi}{\partial W}\big|_{(\gamma,W)}\big)\eta_h$. With the same arguments we have

$$\left(\frac{\partial a}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^* \eta_h = \left(\frac{\partial \dot{J}_W}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^* \tag{5.26}$$

$$\left(\frac{\partial b}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^* \eta_h = \left[\frac{\partial}{\partial W}\left(\langle \Pi_h, \dot{\Psi}_W\rangle\right)\right]\Bigg|^*_{(\gamma_h,W_h)} \tag{5.27}$$

Putting the equations (5.24)–(5.27) into the (5.23) we obtain the (5.21) ending, in this way, the proof of Lemma 5.2. □

Summarizing, the algorithm to compute $\big(\frac{d\Pi}{d\gamma}\big|_{(\gamma_h,W_h)}\big)^*\chi$ is then the following (see also Fig. 5.2):

1. compute the state W_h such that $\Psi(\gamma_h, W_h) = 0$;
2. compute $\bar{W}_J = \left(\frac{\partial J}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^*$
3. compute the adjoint state Π_h solving the linear system $\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^* \Pi_h = \bar{W}_J$;
4. compute η_h as solution of the linear system $\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)\eta_h = \chi$
5. let $\dot{J}_W(\gamma, W) = \left(\frac{\partial J}{\partial W}\bigg|_{(\gamma,W)}\right)\eta_h$ and $\dot{\Psi}_W(\gamma, W) = \left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma,W)}\right)\eta_h$, then

   (a) compute the quantity $\bar{\gamma}_\xi = \left[\dfrac{\partial}{\partial \gamma}\big(\dot{J}_W - \langle \Pi_h, \dot{\Psi}_W\rangle\big)\right]\bigg|^{*}_{(\gamma_h, W_h)}$

   (b) compute the quantity $\bar{W}_\xi = \left[\dfrac{\partial}{\partial W}\big(\dot{J}_W - \langle \Pi_h, \dot{\Psi}_W\rangle\big)\right]\bigg|^{*}_{(\gamma_h, W_h)}$

6. compute φ_h as solution of the adjoint linear system $\left(\dfrac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h, W_h)}\right)^{*}\varphi_h = \bar{W}_\xi$

7. compute $\left(\dfrac{d\Pi}{d\gamma}\bigg|_{(\gamma_h, W_h)}\right)^{*}\chi = \bar{\gamma}_\xi - \left(\dfrac{\partial \Psi}{\partial \gamma}\bigg|_{(\gamma_h, W_h)}\right)^{*}\varphi_h$

Remark 5.4. It it important to mention the fact that the terms $\dot{J}_W(\gamma, W) = \big(\frac{\partial J}{\partial W}\big|_{(\gamma,W)}\big)\eta_h$ and $\dot{\Psi}_W(\gamma, W) = \big(\frac{\partial \Psi}{\partial W}\big|_{(\gamma,W)}\big)\eta_h$ are the directional derivatives of J and Ψ respect to W (along the direction η_h) and therefore can be obtained using AD with Tangent-mode differentiation. In this way the terms

$$\left(\frac{\partial a}{\partial \gamma}\bigg|_{(\gamma_h, W_h)}\right)^{*}\eta_h = \left(\frac{\partial \dot{J}_W}{\partial \gamma}\bigg|_{(\gamma_h, W_h)}\right)^{*} \tag{5.28}$$

$$\left(\frac{\partial a}{\partial W}\bigg|_{(\gamma_h, W_h)}\right)^{*}\eta_h = \left(\frac{\partial \dot{J}_W}{\partial W}\bigg|_{(\gamma_h, W_h)}\right)^{*} \tag{5.29}$$

can be easily computed using the Reverse mode differentiation of a routine differentiated in Tangent mode: for this reason we call this double-differentiation mode Reverse-on-Tangent (RoT). Regarding the terms involving the scalar product of the adjoint state Π_h with $\dot{\Psi}_W$, namely

$$\left(\frac{\partial b}{\partial \gamma}\bigg|_{(\gamma_h, W_h)}\right)^{*}\eta_h = \left[\frac{\partial}{\partial \gamma}\big(\langle \Pi_h, \dot{\Psi}_W\rangle\big)\right]\bigg|^{*}_{(\gamma_h, W_h)} \tag{5.30}$$

and

$$\left(\frac{\partial b}{\partial W}\bigg|_{(\gamma_h, W_h)}\right)^{*}\eta_h = \left[\frac{\partial}{\partial W}\big(\langle \Pi_h, \dot{\Psi}_W\rangle\big)\right]\bigg|^{*}_{(\gamma_h, W_h)}, \tag{5.31}$$

we note that the Reverse mode differentiation of a vectorial function $\Phi\colon x \mapsto \Phi(x)$, with $x,\Phi(x) \in \mathbb{R}^N$ writes as

$$\bar{x}_\Phi = \left(\frac{d\Phi}{dx}\right)^{*}\bar{\Phi} = \begin{pmatrix} \cdots \\ \displaystyle\sum_{k=1}^{N} \frac{d\Phi_k}{dx_i}\bar{\Phi}_k \\ \cdots \end{pmatrix} = \begin{pmatrix} \cdots \\ \dfrac{d\langle \bar{\Phi}, \Phi\rangle}{dx_i} \\ \cdots \end{pmatrix} = \left(\frac{d\langle \bar{\Phi}, \Phi\rangle}{dx}\right)^{*}$$

with $\bar{x}_\Phi$, $\bar{\Phi} \in \mathbb{R}^n$. The last relation means that the Reverse mode differentiation can be viewed as the differentiation of a scalar product and this is exactly what we need to compute in the right hand side of (5.30)-(5.31), where the role of the function Φ is assumed by $\dot{\Psi}_W$ (i.e. the Tangent mode differentiation of Ψ respect to W and along the direction η_h) and the vector $\bar{\Phi}$ is substituted by the adjoint state Π_h. Again, as we have done above, this approach could be viewed like a Reverse-on-Tangent mode differentiation.

Remark 5.5. In Section 3.7 we have seen that Tangent-on-Reverse differentiation raises some Automatic Differentiation issues for TAPENADE. The problem relies on the implementation of the differentiated `PUSH/POP` routines (i.e. `PUSH_D/POP_D`) that use the same stack for the original and differentiated variables (at present, to overcome this error the user must fix this problem by hand on the ToR code, activating the stack from the beginning of the program).

Then, due to the fact that now we are dealing with Reverse-on-Tangent, i.e. a double differentiation in which one mode will add `PUSH/POP` pairs, the natural question could be: "Will I have also problems to perform RoT with TAPENADE ?". The answer is: "No". The reason is simple: we perform Reverse differentiation *after* the Tangent one, so the `PUSH/POP` pairs will be eventually added at the second stage of differentiation (it is worth to remember that Tangent mode adds the differentiated code mantaining the same program structure, without the necessity of a stack).

Remark 5.6. The cost of RoT differentiation (steps 5a-5b), that could be estimated as $\alpha_R\alpha_T$, is negligible respect to the cost for computing with an iterative matrix-free method the solution of the involved linear systems ($\alpha_T < \alpha_R \ll n_{\text{iter}}$). Thus, provided the adjoint state Π_h, the main contribution in terms of run-time cost for the evaluation of $\left(\frac{d\Pi}{d\gamma}\right)^* \bar{\Pi}_F$ is due to the solution of two linear systems: one is in Tangent form (step 4) which cost is $\alpha_T n_{\text{iter},T}$, and the other one (step 6) is in adjoint form and its cost can be estimated as $\alpha_R n_{\text{iter},R}$.

5.4.1 Implementation

In the same manner we have done for ToT and ToR (Sections 3.6.1 and 3.6.2) we would go into the practical details of the implementation for RoT, and how the differentiated routine should be invoked to have te correct results. As usual, let us suppose that the routine computing the state residual $\Psi(\gamma, W)$ is `state_residuals(psi,gamma,w)`, where the input variables are `gamma` and `w`, and the output variable is `psi`

$$\texttt{state\_residuals}(\underset{\downarrow}{\texttt{psi}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{w}})\ .$$

Automatic Differentiation in Tangent mode with respect to the input variables `w` builds the subroutine

$$\texttt{state\_residuals\_dw\_d}(\underset{\downarrow}{\texttt{psi}}, \underset{\downarrow}{\texttt{psid}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{w}}, \overset{\downarrow}{\texttt{wd}})$$

that has the additional output `psid` $= \dot{\Psi}_W = \left(\frac{\partial \Psi}{\partial W}\right)\dot{W}$ and where `wd` $= \dot{W}$ is the additional input variable. Now we differentiate the last routine in Reverse mode considering `psid` as the output variable and with respect to `gamma` and `w`, obtaining

$$\texttt{state\_residuals\_dw\_d\_b}(\underset{\downarrow}{\texttt{psi}}, \underset{\downarrow}{\texttt{psid}}, \overset{\downarrow}{\texttt{psidb}}, \overset{\downarrow}{\texttt{gamma}}, \underset{\downarrow}{\texttt{gammab}}, \overset{\downarrow}{\texttt{w}}, \underset{\downarrow}{\texttt{wb}}, \overset{\downarrow}{\texttt{wd}}) \tag{5.32}$$

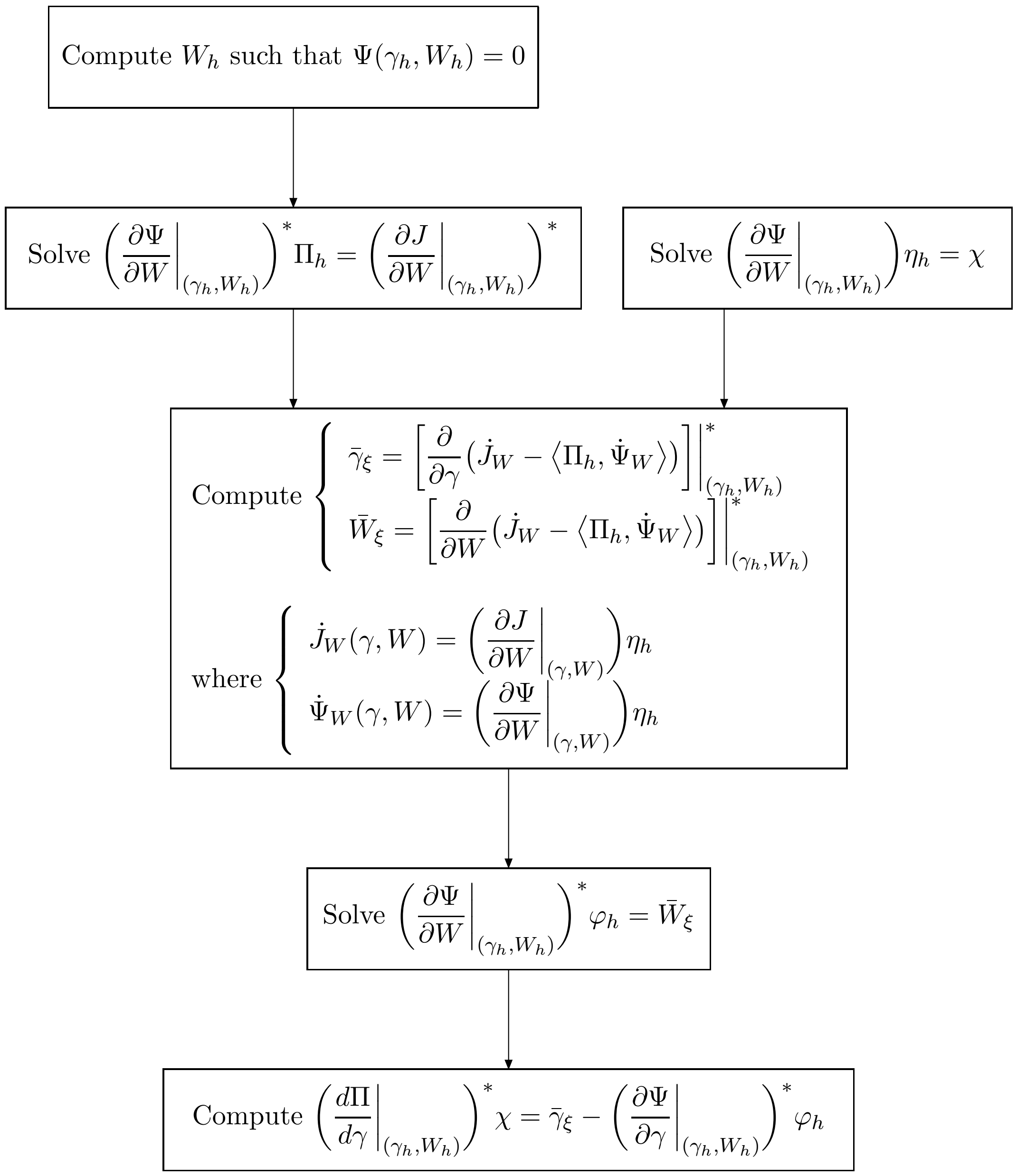

Compute W_h such that $\Psi(\gamma_h, W_h) = 0$
Solve $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h, W_h)}\right)^* \Pi_h = \left(\frac{\partial J}{\partial W}\Big|_{(\gamma_h, W_h)}\right)^*$
Solve $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h, W_h)}\right) \eta_h = \chi$
Compute $\begin{cases} \bar{\gamma}_\xi = \left[\frac{\partial}{\partial \gamma}\left(\dot{J}_W - \langle \Pi_h, \dot{\Psi}_W \rangle\right)\right]\Big|^*_{(\gamma_h, W_h)} \\ \bar{W}_\xi = \left[\frac{\partial}{\partial W}\left(\dot{J}_W - \langle \Pi_h, \dot{\Psi}_W \rangle\right)\right]\Big|^*_{(\gamma_h, W_h)} \end{cases}$
where $\begin{cases} \dot{J}_W(\gamma, W) = \left(\frac{\partial J}{\partial W}\Big|_{(\gamma, W)}\right) \eta_h \\ \dot{\Psi}_W(\gamma, W) = \left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma, W)}\right) \eta_h \end{cases}$
Solve $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h, W_h)}\right)^* \varphi_h = \bar{W}_\xi$
Compute $\left(\frac{d\Pi}{d\gamma}\Big|_{(\gamma_h, W_h)}\right)^* \chi = \bar{\gamma}_\xi - \left(\frac{\partial \Psi}{\partial \gamma}\Big|_{(\gamma_h, W_h)}\right)^* \varphi_h$

Figure 5.2: The algorithm to compute $\left(\frac{d\Pi}{d\gamma}\big|_{(\gamma_h, W_h)}\right)^* \chi$: it needs the solutions of 2 linear systems + the adjoint state Π_h. Note that this algorithm use Reverse-on-Tangent differentiation in order to compute the quantities $\bar{\gamma}_\xi$, $\bar{W}_\xi$.

where $\texttt{psidb} = \bar{\dot{\Psi}}$ is the additional input variable and

$$\texttt{gammab} = \bar{\gamma}_{\dot{\Psi}_W} = \left[\frac{\partial}{\partial \gamma}\Big(\big\langle \bar{\dot{\Psi}}, \Big(\frac{\partial \Psi}{\partial W}\Big)\dot{W}\big\rangle\Big)\right]^*$$
$$\texttt{wb} = \bar{W}_{\dot{\Psi}_W} = \left[\frac{\partial}{\partial W}\Big(\big\langle \bar{\dot{\Psi}}, \Big(\frac{\partial \Psi}{\partial W}\Big)\dot{W}\big\rangle\Big)\right]^*$$

are the additional outputs. In order to compute the quantities (5.30)-(5.31) needed by the algorithm for computing $\big(\frac{d\Pi}{d\gamma}\big|_{(\gamma_h,W_h)}\big)^*\chi$, we must call the routine (5.32) with the right arguments, namely

$$\texttt{state\_residuals\_dw\_d\_b}(\underset{\Psi}{\texttt{psi}}, \underset{\dot{\Psi}_W}{\texttt{psid}}, \overset{\Pi_h}{\texttt{psidb}}, \overset{\gamma_h}{\texttt{gamma}}, \underset{\bar{\gamma}_{\dot{\Psi}_W}}{\texttt{gammab}}, \overset{W_h}{\texttt{w}}, \underset{\bar{W}_{\dot{\Psi}_W}}{\texttt{wb}}, \overset{\eta_h}{\texttt{wd}})$$

in this way the output variables $\bar{\gamma}_{\dot{\Psi}_W}$, $\bar{W}_{\dot{\Psi}_W}$ write as

$$\bar{\gamma}_{\dot{\Psi}_W} = \left[\frac{\partial}{\partial \gamma}\Big(\big\langle \Pi_h, \Big(\frac{\partial \Psi}{\partial W}\Big)\eta_h\big\rangle\Big)\right]\Bigg|^*_{(\gamma_h,W_h)}$$
$$\bar{W}_{\dot{\Psi}_W} = \left[\frac{\partial}{\partial W}\Big(\big\langle \Pi_h, \Big(\frac{\partial \Psi}{\partial W}\Big)\eta_h\big\rangle\Big)\right]\Bigg|^*_{(\gamma_h,W_h)},$$

that are exactly the quantities defined in (5.25)-(5.27).

The same previous approach can be used for the terms $\big(\frac{\partial \dot{J}_W}{\partial \gamma}\big)^*$ and $\big(\frac{\partial \dot{J}_W}{\partial W}\big)^*$. In this case we start from the function that computes the functional $J(\gamma, W)$

$$\texttt{functional}(\underset{\downarrow}{\texttt{j}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{w}})$$

which Automatic Differentiation in Tangent mode with respect to the variable `w` builds the routine

$$\texttt{functional\_dw\_d}(\underset{\downarrow}{\texttt{j}}, \underset{\downarrow}{\texttt{jd}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{w}}, \overset{\downarrow}{\texttt{wd}})$$

where the additional input is $\texttt{wd} = \dot{W}$ and the additional output is $\texttt{jd} = \dot{J}_W = \big(\frac{\partial J}{\partial W}\big)\dot{W}$. If we finally perform a Reverse mode differentiation of the routine above, considering the output variable to be `jd` and with respect to `gamma` and `w`, we obtain

$$\texttt{functional\_dw\_d\_b}(\underset{\downarrow}{\texttt{j}}, \underset{\downarrow}{\texttt{jd}}, \overset{\downarrow}{\texttt{jdb}}, \overset{\downarrow}{\texttt{gamma}}, \underset{\downarrow}{\texttt{gammab}}, \overset{\downarrow}{\texttt{w}}, \underset{\downarrow}{\texttt{wb}}, \overset{\downarrow}{\texttt{wd}}) \tag{5.33}$$

where $\texttt{jdb} = \bar{\dot{J}}$ is an input variable and

$$\texttt{gammab} = \bar{\gamma}_{\dot{J}_W} = \left[\frac{\partial}{\partial \gamma}\Big(\big\langle \bar{\dot{J}}, (\frac{\partial J}{\partial W})\dot{W}\big\rangle\Big)\right]^*$$
$$\texttt{wb} = \bar{W}_{\dot{J}_W} = \left[\frac{\partial}{\partial W}\Big(\big\langle \bar{\dot{J}}, (\frac{\partial J}{\partial W})\dot{W}\big\rangle\Big)\right]^*$$

are output variables. In order to compute the quantities (5.28)-(5.29) needed by equation (5.23), we must call the routine (5.32) using the right arguments, namely

$$\texttt{functional\_dw\_d\_b}(\underset{J}{\texttt{j}}, \underset{j_W}{\texttt{jd}}, \overset{1.0}{\texttt{jdb}}, \overset{\gamma_h}{\texttt{gamma}}, \underset{\bar{\gamma}_{j_W}}{\texttt{gammab}}, \overset{W_h}{\texttt{w}}, \underset{\bar{W}_{j_W}}{\texttt{wb}}, \overset{\eta_h}{\texttt{wd}}) \ .$$

Using the scheme above, we obtain the output variables

$$\bar{\gamma}_{j_W} = \left[\frac{\partial}{\partial \gamma} \left(\frac{\partial J}{\partial W} \eta_h \right) \right] \Bigg|^*_{(\gamma_h, W_h)}$$
$$\bar{W}_{j_W} = \left[\frac{\partial}{\partial W} \left(\frac{\partial J}{\partial W} \eta_h \right) \right] \Bigg|^*_{(\gamma_h, W_h)}$$

that are exactly the quantities defined by (5.28)-(5.29).

5.5 Conclusion

For the evaluation of the gradient $\frac{dj}{d\gamma}$ the main contribution to the cost is due to the computation of the adjoint state Π_h and this cost is estimated to be $\alpha_R n_{\text{iter},R}$, where $n_{\text{iter},R}$ is the number of iterations needed to solve the adjoint linear system and α_R is the overhead associated with the Reverse mode differentiation of the state residual $\Psi(\gamma, W)$ (for which we assume an unitary cost). For the gradient of the correction term, $\frac{df}{d\gamma}$, the main contribution to the runtime cost is due to the solution of three linear systems (2 adjoint + 1 tangent) for an estimated cost of $2\alpha_R n_{\text{iter},R} + \alpha_T n_{\text{iter},T}$, where we assumed $n_{\text{iter},T}$ to be the number of iterations needed to solve the tangent linear system and α_R is the overhead associated with the Tangent mode differentiation of the state residual $\Psi(\gamma, W)$. Therefore the total cost for the computation of the gradient for the adjoint-corrected functional can be estimated $3\alpha_R n_{\text{iter},R} + \alpha_T n_{\text{iter},T}$ (see also Table 5.1).

At the present time we have no numerical experiments, but we have planned to apply this strategy in order to build a gradient-based optimization algorithm for the adjoint-corrected functional, in which we'll use the 3D Euler solver presented in Chapter 1.

	Main contribution to the cost	Approx. runtime cost
$\dfrac{dj}{d\gamma}$	$\Pi_h = \left(\dfrac{\partial\Psi}{\partial W}\right)^{-*}\bar{W}_J$	$\alpha_R n_{\text{iter},R}$
$\dfrac{df}{d\gamma}$	$l = \left(\dfrac{\partial\Psi}{\partial W}\right)^{-*}\bar{W}_F$ $\eta_h = \left(\dfrac{\partial\Psi}{\partial W}\right)^{-1}\bar{\Pi}_F$ $\varphi_h = \left(\dfrac{\partial\Psi}{\partial W}\right)^{-*}\bar{W}_\xi$	$\alpha_R n_{\text{iter},R}$ $\alpha_T n_{\text{iter},T}$ $\alpha_R n_{\text{iter},R}$
$\dfrac{dj^c}{d\gamma} = \dfrac{dj}{d\gamma} + \dfrac{df}{d\gamma}$	3 adjoint linear systems + 1 tangent linear system	$3\alpha_R n_{\text{iter},R} + \alpha_T n_{\text{iter},T}$

Table 5.1: Runtime cost for the adjoint-corrected functional. $n_{\text{iter},T}$ ($n_{\text{iter},R}$) is the number of iterations needed to solve the tangent (adjoint) linear system and α_T (α_R) is the overhead associated with the Tangent-mode (Reverse-mode) differentiation of the state residual $\Psi(\gamma, W)$ (for which we assume an unitary cost).

Chapter 6

Numerical experiments using Automatic Differentiation

In Chapter 3 we have presented two algorithms based on Automatic Differentiation (ToT and ToR) for computing the Hessian matrix of a functional subject to satisfy an equality constraint. In this chapter, we want to present some numerical experiments we have done in order to validate the two approaches and to give an idea about the cost for the evaluation of the Hessian matrix with Automatic Differentiation on a three-dimensional CFD code. Moreover, we want to analyze the performances of linear solvers built following the matrix-free approach as described in Section 3.3.

The numerical code we used, is a FORTRAN77 implementation of the finite-volume method for 3D unstructured meshes presented in Chapter 1, in which a steady solution for the Euler equations is searched. The numerical algorithm uses the Roe's scheme for the numerical fluxes in the inner domain and a Steger-Warming flux-splitting for the farfield boundary, resulting in a method with first-order spatial accuracy. Moreover, it permits to have second-order spatial accuracy using a MUSCL scheme associated with the Van Albada-Van Leer limiter.

Remark 6.1. From a mathematical point of view Roe's scheme is not differentiable, but only piecewise differentiable.

In order to follow the approach presented in Section 3.8, the required work has been divided in three phases:

- first of all we wrote the two scripts (see Appendix C) that drives TAPENADE to perform the first- and second-order differentiations required by the various algorithms;
- then we have implemented (and validated) the libraries containing the algorithms in which the differentiated functions are used;
- finally we have implemented the specific functional of interest (in our case the drag coefficient) and the state residual in two routines which interfaces are `functional(j,gamma,w)` and `state_residual(psi,gamma,w)`. The implementation is made recognizing and extracting the required quantities and procedures from the CFD code.

From the scheme above, we note that the CFD solver is not modified: our algorithms need only to know the state solution W_h and some procedures *from* the solver. However, we have made some modification into the code in order to permit our algorithms to work efficiently. The original code uses Jacobi (or Gauss-Seidel) iterations for the relaxation procedure and, in order to converge faster, it uses as preconditioner the inverse-diagonal of the Jacobian matrix $\frac{\partial \Psi^{(1)}}{\partial W}$. This approach, although is easy and cheap to build, is not well-suited to be used in the algorithms for computing the gradient and the Hessian matrix. The problem rely on the fact that the computational cost of such algorithms depends strongly on the cost to solve the linear system inside them, and therefore we need to use more sophisticaded approaches to build a good preconditioner and a fast (and robust) linear solver. For this reason we have modified the code introducing the routines to compute ILU(p) factorization (see [Saad, 1996]) of the first-order Jacobian (and the ILU(p) factorization of the transpose $\left(\frac{\partial \Psi^{(1)}}{\partial W}\right)^T$) that we use as preconditioners for the GMRES-RCI linear solvers in our algorithms. The ILU(p) implementation we used is from the Y. Saad's SPARSKIT[1] library [Saad, 1994]. In Section 6.1 is presented a study of the efficiency of the linear solver using various levels of fill for the ILU(p) preconditioner.

Summarizing, we have implemented in the library for the gradient and Hessian evaluation the following methods:

- the GMRES-RCI solvers for the tangent linear system $\left(\frac{\partial \Psi}{\partial W}\right)\xi = b$ and the adjoint linear system $\left(\frac{\partial \Psi}{\partial W}\right)^T \xi = c$, using the "matrix-free" approach as described in Section 3.3;
- the routine for the computation of the adjoint state Π_h and the gradient of the constrained functional $\frac{dj}{d\gamma}$, accordingly with equations (3.18) and (3.17);
- the full Hessian evaluation using ToT approach, as described in Section 3.6.1 (see also Fig. 3.6);
- the diagonal part of the Hessian using ToT approach;
- the Hessian-by-vector multiplication using ToR differentiated functions (Lemma 3.1);
- the full Hessian evaluation using ToR approach, as described in Section 3.6.2 (see also Fig. 3.8).

Moreover, we have implemented the algorithms to validate of first- and secon-order differentiated routines. The validation strategies use finite differences for the tangent mode differentiation and a *"dot-product"* test for Reverse differentiation, as explained in [Hascoët and Pascual, 2004].

The numerical experiments involve the study of efficency of linear solvers using a matrix-free approach in which AD is used to perform matrix-by-vector multiplications, and the computation of the gradient and the Hessian matrix of the drag coefficient, respect to the angle of attack and the Mach number, for two different geometries and different flow regimes.

[1]`http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`

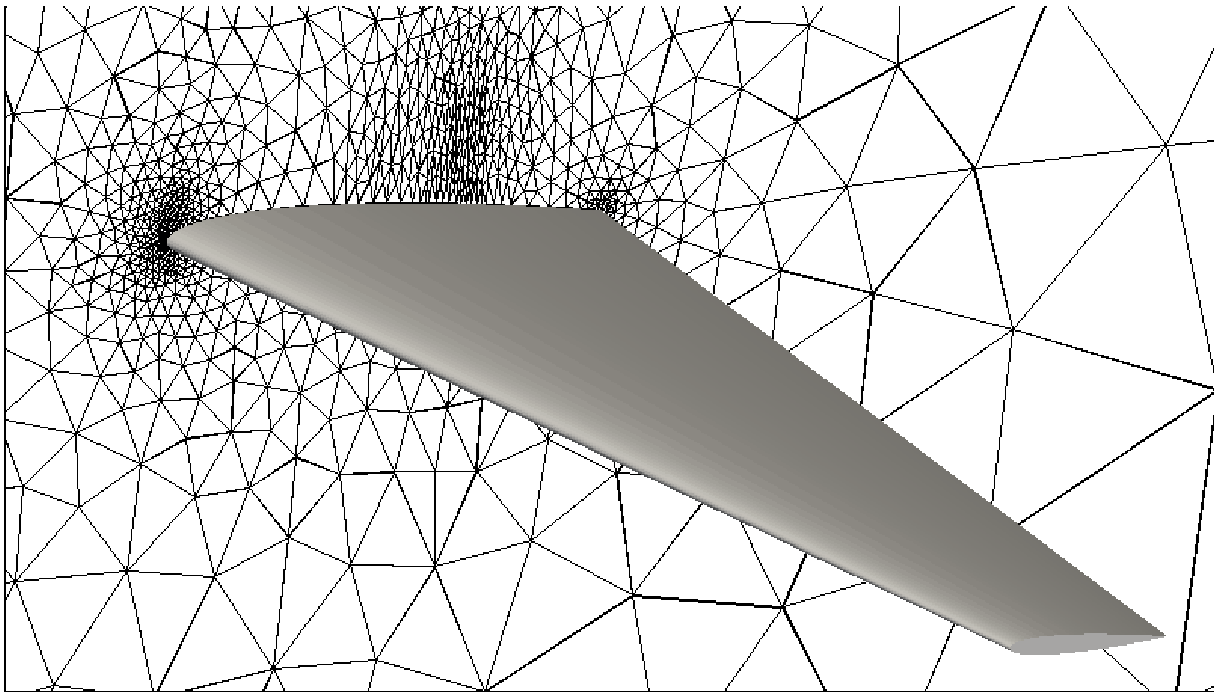


Figure 6.1: Wing shape and mesh in the symmetry plane.

6.1 Study of efficiency of matrix-free methods to solve linear systems in the AD context

The testcase considered here corresponds to the wing shape of a business aircraft (courtesy of Piaggio Aero Industries), for a transonic regime. The nominal operating conditions are defined by the free-stream Mach number $M_\infty = 0.83$ and the incidence $\alpha = 2°$. The wing section correspond to the NACA0012 airfoil and a picture is given in Figure 6.1. The 3D unstructured mesh has 31124 nodes and 173445 elements (see [Andreoli et al., 2003] for the details about the wing geometry and the mesh generation).

For the experiments we have used a GMRES algorithm with restart [Frayssé et al., 2003; Saad, 1996], and we studied the impact of different preconditioners and different dimensions of the Krylov space for solving the first- and second-order accurate linear system $\left(\frac{\partial \Psi^{(1,2)}}{\partial W}\right)\xi = b$ and the corresponding adjoint $\left(\frac{\partial \Psi^{(1,2)}}{\partial W}\right)^* \xi = c$.

For the preconditioning we have used an Incomplete LU factorization with different levels of fill (ILU(p), [Saad, 1996]) applied on the first-order Jacobian $A^{(1)} = \left(\frac{\partial \Psi^{(1)}}{\partial W}\right)$ (for the adjoint linear system we have factorized its transposte ${A^{(1)}}^T$). The first-order Jacobian $A^{(1)}$ is a block-sparse structurally-symmetric matrix and for our mesh it has 449028 5×5-blocks (31124 blocks on the diagonal and 417904 off-diagonal): its sparsity pattern is shown in the Figure 6.2.

For each configuration of the parameters (order of spatial accuracy, type of preconditioner, dimension of the Krylov space) we have solved two linear systems $\left(\frac{\partial \Psi}{\partial W}\right)\xi = b$ (i.e. we have

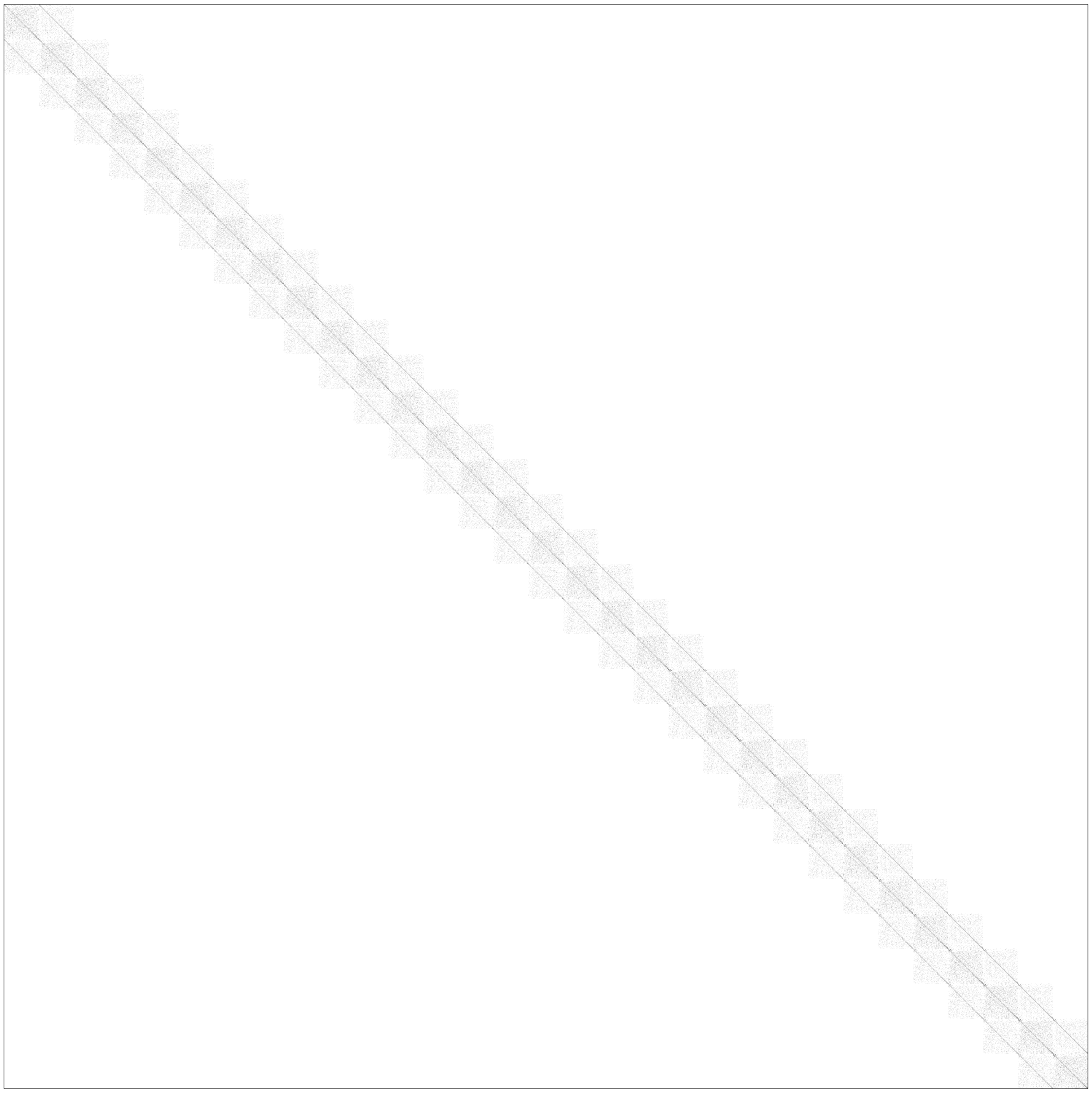


Figure 6.2: Sparsity pattern of the first-order Jacobian matrix $\frac{\partial \Psi^{(1)}}{\partial W}$ for the 3D unstructured mesh corresponding to the wing geometry in Fig. 6.1. The matrix has a block-sparse structurally-symmetric pattern, with 449028 5×5-blocks (31124 blocks on the diagonal and 417904 off-diagonal).

found the solution ξ for two different values of the right hand side b) and three adjoint linear systems $\left(\frac{\partial \Psi}{\partial W}\right)^* \xi = c$. To be more precise, for the first case we have used the quantities $b = -\frac{\partial \Psi}{\partial \alpha}$ and $b = -\frac{\partial \Psi}{\partial M}$ (where α is the angle of attack and M the free-stream Mach number), while for the adjoint linear system we have used $c = \left(\frac{\partial c_D}{\partial W}\right)^T$ (where c_D is the drag coefficient) and other two quantities that are needed to obtain the Hessian matrix of the drag coefficient (respect to the angle of attack and the Mach number) using the Tangent-on-Reverse algorithm (see the Section 3.6.2).

Moreover, due to the availability of the block-diagonal inverse D^{-1} of the matrix $A^{(1)}$, we have applied the above strategy to te equivalent systems $D^{-1}A^{(1,2)}\xi = D^{-1}b$ and $D^{-T}{A^{(1,2)}}^T\xi = D^{-T}c$. The stopping criterion for the GMRES algorithm is based on the backward error for the preconditioned system and was set to $\varepsilon = 10^{-12}$.

For the different runs, we have used a 2.66 GHz-64bit Intel Xeon processor (5150 series) with 4 MB of L2-cache memory and 8 GB of RAM. All the computation are performed with double precision (8 Bytes) floating point numbers.

ILU(p) preconditioning. The preconditioners are built using 3 different levels of fill: ILU(0) (i.e. no fill respect the original structure), ILU(1) and ILU(2). The computational cost in terms of CPU time and memory to build each preconditioner is shown in Table 6.1.

	CPU time	Memory
ILU(0)	3 sec.	$\sim$ 86 MB
ILU(1)	21 sec.	$\sim$ 170 MB
ILU(2)	69 sec.	$\sim$ 350 MB

Table 6.1: Computational cost (time and memory requirements) for different levels of fill for the Incomplete LU factorization. The original matrix is relative on a mesh with 31124 nodes and it has 449028 5×5 blocks resulting in a memory requirement of $\sim$ 86 MB.

Krylov space dimension. We have used 3 different dimensions for the Krylov space: 15 (Tables 6.3, 6.6), 30 (Tables 6.4, 6.7) and 200 (Tables 6.5, 6.8). The results show that, as expected, the number of iterations needed for convergence decrease as the dimension of the Krylov space increase, but at a cost of a greater memory requirement. In the implementation of GMRES that we have used, the dimension d_{work} of the workspace depends on the dimension of the solution n and the dimension of Krylov space d_K and, for the implementation we used, it should be not less than $d_K(d_K+n+5)+6n+2$ [Frayssé et al., 2003]. In our case, the dimension of the solution was $n = 5 \times 31124$ and the corresponding dimension for the workspace respect to the Krylov space is shown in the Table 6.2.

Order 1

For the case in which the matrix A is the first-order accurate Jacobian $\frac{\partial \Psi^{(1)}}{\partial W}$, we note that the number of iterations needed to converge to the solution is nearly independent from the right

d_K	d_{work}	Memory
15	3268322	26146576 Byte ($\sim$ 25 MB)
30	5603372	44826976 Byte ($\sim$ 43 MB)
200	32877847	263022776 Byte ($\sim$ 251 MB)

Table 6.2: Minimum dimension for the workspace respect to the dimension of the Krylov space d_K in the GMRES algorithm. The dimension for the workspace depends on the particular implementation and in our case must be $d_{\text{work}} \geq d_K(d_K + n + 5) + 6n + 2$ [Frayssé et al., 2003], where n is the dimension of the right hand side of the linear system.

hand side and generally (if we use the same preconditioner) the system $A\xi = b$ requires more iteration for the convergence respect to the transposed one $A^T\xi = c$ (Tables 6.3–6.5).

Due to the strategy that we have adopted for the matrix-by-vector multiplication inside the GMRES solver (see Section 3.3) and by the overhead associated with the differentiated code (in this case for the tangent mode differentiation we had $\alpha_T \simeq 2.2$, while for the reverse mode $\alpha_R \simeq 4.6$), we have for the adjoint linear system a cost-for-iteration about twice larger respect to the tangent linear system.

Regarding the preconditioner, we note that when increasing the level of filling we need a smaller number of iterations to converge, but this fact does not give a proportional saving for the computational cost, due to the greater number of nonzero elements of the preconditioner itself (in other words the application of the ILU(p+1) preconditioner on a vector has a higher cost respect to the application of ILU(p)). This fact results in a comparable performance between the ILU(1) and ILU(2), but as we have seen above, the ILU(2) requires a lot more memory and a greater computational cost to be built.

For the dimension of the Krylov space, we have obtained similar cost between $d_K = 15$ and $d_K = 30$ using ILU(1) and ILU(2). For ILU(0) we note an improvement for the performance using $d_K = 30$ only for the solution of the direct system $A\xi = b$. As expected, the case $d_K = 200$ result in the lowest computational time, but with a greater cost in terms of memory.

Using the equivalent linear system built with the left-multiplication of the matrix A (A^T) and the right-hand-side b (c) by the block-inverse matrix D^{-1} (D^{-T}) to compute the solution ξ, does not affect in a significative way the performance of the solver.

Order 2

As for case of the first-order accurate Jacobian, when we solve the second-order accurate linear system $A\xi = b$ (or the adjoint linear system $A^T\xi = c$) with $A = \frac{\partial\Psi^{(2)}}{\partial W}$ and using a preconditioner built for the first-order accurate problem, we have a number of iterations nearly independent from the right hand side and generally a higher number of iterations for the tangent linear system respect to the adjoint one (Tables 6.6–6.8).

In this case the overhead associated with the differentiated code is $\alpha_T \simeq 2.0$ for the tangent mode and $\alpha_R \simeq 3.3$ for the reverse mode, resulting in a cost-for-iteration for the solution of the adjoint linear system that is about 1.65 times greater respect to the analogous operation for the tangent linear system.

The choice of the preconditioner is more crucial here respect to the first-order accurate problem: in fact if the Krylov space is too small the ILU(0) preconditioner results in a lack of convergence (at least until our limit of 600 iterations) for the tangent linear system. Using ILU(1) and ILU(2) we note the same behaviour in terms of number of iterations (i.e. for a given system ILU(1) requires more iterations than ILU(2)) and in terms of computational time in some cases we have obtained better performances for ILU(1) respect to ILU(2). For these reasons, together with the considerations about the computational time to build the preconditioner and the memory requirements, the better strategy for the preconditioner seems to be the ILU(1) respect to ILU(2).

Regarding the Krylov space, we note a stronger influence on the number of iteration respect to the first-order case (where $d_K = 15$ and $d_K = 30$ result in a similar cost and the case $d_K = 200$ result in a saving in terms of computational time of $\sim 15\%$). In this case (and considering the ILU(1) strategy for the preconditioner) moving from $d_K = 15$ to $d_K = 30$ results in a save of $\sim 15\%$, and from $d_K = 30$ to $d_K = 200$ results in the saving of the computational time of a factor > 2. The major drawback of a such great dimension for the Krylov space is, as we have seen, the memory requirements for the workspace of the GMRES solver, therefore the best strategy seems to be in choosing the higher possible dimension such that the workspace for the GMRES routine is contained in the RAM of the system.

Using the alternative formulation for the linear system with the multiplication of both side by the block inverse matrix (D^{-1} for the tangent linear system an D^{-T} for the adjoint) results in a lower number of iterations to the convergence, especially for the tangent case.

	A	$D^{-1}A$	A^T	$D^{-T}A^T$
ILU(0)	$295 + 302$ (71.7)	$295 + 302$ (71.6)	$173 + 172 + 174$ (66.7)	$173 + 172 + 174$ (63.9)
ILU(1)	$113 + 106$ (35.9)	$113 + 104$ (35.4)	$91 + 92 + 95$ (44.2)	$91 + 92 + 92$ (44.8)
ILU(2)	$91 + 82$ (39.0)	$78 + 80$ (**33.0**)	$64 + 67 + 68$ (**36.7**)	$64 + 67 + 68$ (37.7)

Table 6.3: Order 1, Krylov= 15. Iteration needed to GMRES for the convergence of the various linear systems and different preconditioners: the columns 2 and 3 regard the solution of $A\xi = b$, while the columns 3 and 4 regard the solution of $A^T\xi = c$. In the parenthesis is shown the average time (in seconds) to solve a linear system. The bold value is the minimum time (without considering the ILU(p) factorization) to solve the linear system.

	A	$D^{-1}A$	A^T	$D^{-T}A^T$
ILU(0)	$205 + 199$ (49.2)	$205 + 198$ (48.9)	$167 + 169 + 172$ (66.7)	$167 + 169 + 172$ (64.7)
ILU(1)	$103 + 101$ (33.4)	$103 + 100$ (33.7)	$87 + 87 + 87$ (42.6)	$87 + 87 + 87$ (41.1)
ILU(2)	$91 + 70$ (37.3)	$74 + 69$ (**30.1**)	$62 + 64 + 66$ (37.10)	$62 + 64 + 66$ (**36.7**)

Table 6.4: Order 1, Krylov= 30. Iteration needed to GMRES for the convergence of the various linear systems and different preconditioners: the columns 2 and 3 regard the solution of $A\xi = b$, while the columns 3 and 4 regard the solution of $A^T\xi = c$. In the parenthesis is shown the average time (in seconds) to solve a linear system. The bold value is the minimum time (without considering the ILU(p) factorization) to solve the linear system.

	A	$D^{-1}A$	A^T	$D^{-T}A^T$
ILU(0)	$148 + 148$ (38.1)	$148 + 148$ (38.6)	$134 + 137 + 138$ (55.1)	$134 + 137 + 138$ (54.2)
ILU(1)	$81 + 80$ (27.7)	$81 + 79$ (28.2)	$73 + 74 + 75$ (36.8)	$73 + 74 + 75$ (37.1)
ILU(2)	$108 + 60$ (45.0)	$61 + 59$ (**25.6**)	$54 + 57 + 57$ (**31.8**)	$54 + 57 + 57$ (31.9)

Table 6.5: Order 1, Krylov= 200. Iteration needed to GMRES for the convergence of the various linear systems and different preconditioners: the columns 2 and 3 regard the solution of $A\xi = b$, while the columns 3 and 4 regard the solution of $A^T\xi = c$. In the parenthesis is shown the average time (in seconds) to solve a linear system. The bold value is the minimum time (without considering the ILU(p) factorization) to solve the linear system.

	A	$D^{-1}A$	A^T	$D^{-T}A^T$
ILU(0)	$--$	$--$	454	453
ILU(1)	$458 + 421$ (168.7)	$447 + 408$ (**167.3**)	$354 + 365 + 392$ (**239.5**)	$353 + 364 + 388$ (250.1)
ILU(2)	$379 + 357$ (171.2)	$374 + 356$ (170.0)	$329 + 352 + 349$ (250.1)	$329 + 349 + 348$ (250.4)

Table 6.6: Order 2, Krylov= 15. Iteration needed to GMRES for the convergence of the various linear systems and different preconditioners: the columns 2 and 3 regard the solution of $A\xi = b$, while the columns 3 and 4 regard the solution of $A^T\xi = c$. In the parenthesis is shown the average time (in seconds) to solve a linear system. The bold value is the minimum time (without considering the ILU(p) factorization) to solve the linear system. For ILU(0) the GMRES algorithm to solve $Ax = b$ does not converge.

	A	$D^{-1}A$	A^T	$D^{-T}A^T$
ILU(0)	$562 + 511$ (163.0)	$560 + 501$ (158.0)	$438 + 504 + 469$ (262.0)	$438 + 500 + 466$ (258.0)
ILU(1)	$431 + 361$ (154.27)	$412 + 345$ (**143.9**)	$307 + 317 + 314$ (210.7)	$307 + 315 + 313$ (203.8)
ILU(2)	$361 + 301$ (157.0)	$350 + 300$ (151.9)	$244 + 271 + 268$ (199.5)	$244 + 266 + 267$ (**186.25**)

Table 6.7: Order 2, Krylov= 30. Iteration needed to GMRES for the convergence of the various linear systems and different preconditioners: the columns 2 and 3 regard the solution of $A\xi = b$, while the columns 3 and 4 regard the solution of $A^T\xi = c$. In the parenthesis is shown the average time (in seconds) to solve a linear system. The bold value is the minimum time (without considering the ILU(p) factorization) to solve the linear system.

	A	$D^{-1}A$	A^T	$D^{-T}A^T$
ILU(0)	$240 + 270$ (82.6)	$232 + 262$ (78.8)	$195 + 212 + 217$ (121.5)	$195 + 211 + 211$ (122.9)
ILU(1)	$210 + 201$ (103.7)	$152 + 148$ (**60.7**)	$133 + 141 + 141$ (92.5)	$133 + 140 + 140$ (95.0)
ILU(2)	$201 + 127$ (96.1)	$129 + 126$ (61.7)	$113 + 121 + 120$ (89.5)	$113 + 120 + 120$ (**89.4**)

Table 6.8: Order 2, Krylov= 200. Iteration needed to GMRES for the convergence of the various linear systems and different preconditioners: the columns 2 and 3 regard the solution of $A\xi = b$, while the columns 3 and 4 regard the solution of $A^T\xi = c$. In the parenthesis is shown the average time (in seconds) to solve a linear system. The bold value is the minimum time (without considering the ILU(p) factorization) to solve the linear system.

6.2 Gradient and Hessian evaluation using AD

For the following tests, we have computed the gradient and the Hessian matrix of the drag coefficient (our functional) with respect to the free-stream Mach number and incidence angle. For the gradient we used the adjoint approach as explained in Section 3.5.2, while for the Hessian we used ToT and ToR approach, as explained in Sections 3.6.1 and 3.6.2. For the solution of the linear system involved in the various algorithms, we used the matrix-free approach (Sections 3.3 and 6.1) and for the preconditioners an ILU(1) strategy applied to the first-order accurate Jacobian $\left(\frac{\partial \Psi^{(1)}}{\partial W}\right)$ (and to the corresponding transpose $\left(\frac{\partial \Psi^{(1)}}{\partial W}\right)^T$ for the adjoint case).

6.2.1 Testcase 1: wing shape geometry

The Testcase 1 corresponds to the same wing shape and the same nominal operating conditions we used in the previous section (i.e. free-stream Mach number $M_\infty = 0.83$ and incidence angle $\alpha = 2°$). For this testcase we have computed the gradient and the Hessian using the CFD solver with first- and second-order spatial accuracy. Moreover, to validate and compare our results, we have built a database of drag coefficients corresponding to a 21×21 grid in the α-M_∞ plane around the nominal value ($\alpha = 2°, M_\infty = 0.83$): in other words we have solved 441 steady Euler equations $\Psi(\gamma, W) = 0$.

As shown in Fig 6.3, using first-order spatial accuracy (i.e. without MUSCL scheme and limiters) we have obtained the correct gradient and Hessian (both validated using divided differences). In this figure, we note a very good agreement between the results obtained with a second-order Taylor expansion (Fig 6.3 on bottom-left) and the values obtained with nonlinear simulations (Fig 6.3 on top-left). The relative difference between the two approaches is shown in Fig 6.3 on the bottom-right position.

To give a rough idea of the computational cost, we need about 60 seconds on a 2.66 GHz-64bit Intel Xeon workstation to solve the nonlinear equation $\Psi(\gamma, W) = 0$ and evaluate the functional at the nominal values $M_\infty = 0.83$, $\alpha = 2°$ (this cost depends on the flow regime: with higher Mach numbers the solver needs more iterations to achieve, if possible, the same error level on the residual), while to compute the adjoint state and the gradient we spent about 45 seconds. Regarding the Hessian, ToT and ToR gave us the same numerical results but, as expected their cost differs: ToT required about 75 seconds while ToR required about 162 seconds. Therefore, to obtain the second-order Taylor expansion we used ToT and the runtime cost was about 3 minutes, while to built the fully nonlinear database the runtime cost was more than 7.5 hours.

Using the second-order spatial accuracy (with MUSCL approach and Van Albada-Van Leer limiter) and the same operational conditions, we have obtained the correct gradient (Fig 6.4 in the top-right position) but *wrong* Hessian (Fig 6.4 in the bottom position). This is not a problem of the implementation of the algorithms: ToT and ToR gave us the same numerical results and all linear systems involved for the Hessian evaluation converged to the solution with an error level on the residual of 10^{-12}.

We think that the reason of this strange behaviour could rely on numerical truncation errors on the double-differentiated code. An example of this issue is given considering the function

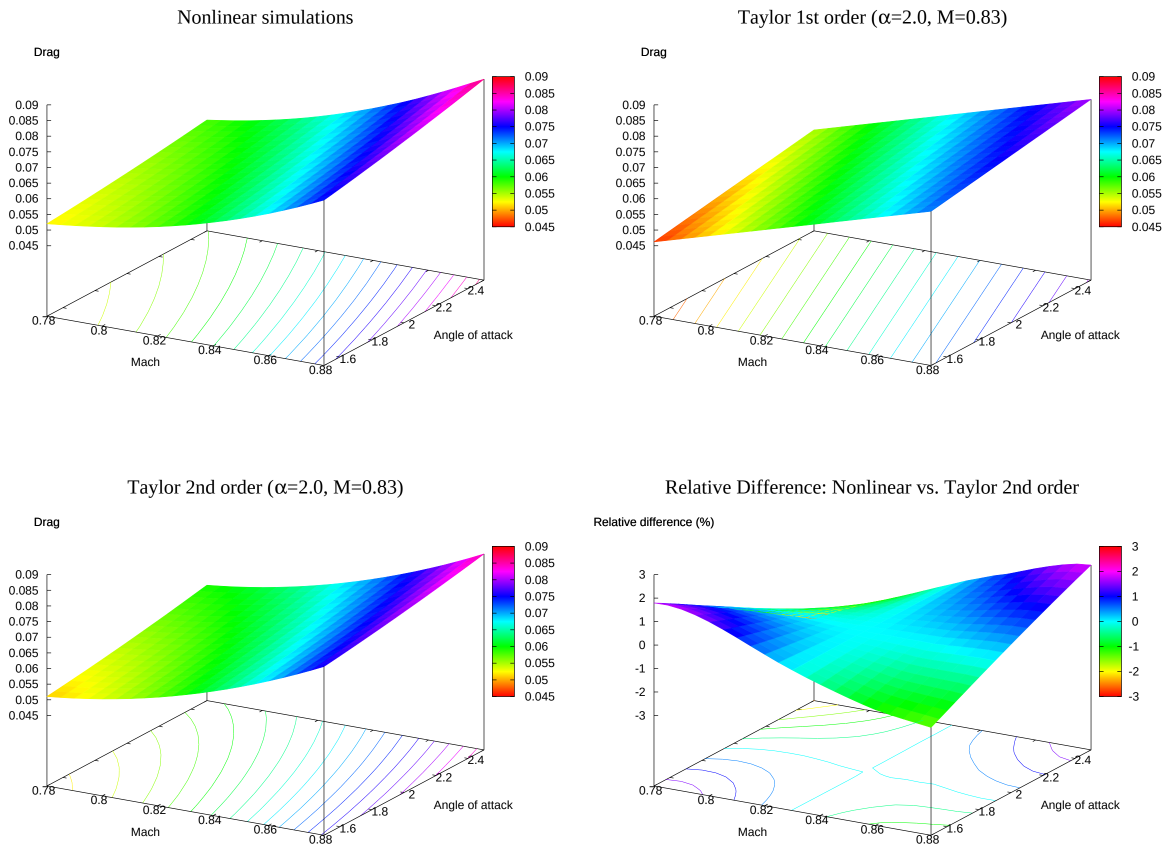


Figure 6.3: Drag coefficient vs. Mach number and angle of attack (first-order spatial accuracy) for the Piaggio wing: nonlinear simulations (top-left); first-order (top-right) and second-order (bottom-left) Taylor approximation around $\alpha = 2°$ and $M = 0.83$; relative difference between the nonlinear simulations and the second-order Taylor approximation (bottom-right).

$f(x) = x\sqrt{x}$ that is continuous and differentiable in the domain $\big[0, +\infty\big)$ while its derivative $f'(x) = \frac{3}{2}\sqrt{x}$ is continuous but differentiable only for $x \in \big(0, +\infty\big)$. For the second derivative $f''(x) = \frac{3}{4}\frac{1}{\sqrt{x}}$ holds in fact $\lim_{x\to 0^+} f''(x) = +\infty$. Thus, if we run the code implementing $f''(x) = \frac{3}{4}\frac{1}{\sqrt{x}}$ with $x > 0$ but very small, due to the finite representation of numbers in computers, we could have some numerical problems. We emphasize the fact that the problem is not necessarily due to the non-differentiability of the Roe's scheme used by the solver: we have the same feature for the first-order case and the results are correct (moreover, in the second-order case the gradient is correct). Another test made is the Hessian evaluation using a lower and an higher Mach number: in the lower case (with $M = 0.5$) the Hessian is correct, while with the higher values of $M = 0.87$ the Hessian computed is wrong (with exactly the same executable program and the same mesh). This problem merits further research.

Regarding the computational cost, we need about 3 minutes to solve the nonlinear equation $\Psi(\gamma, W) = 0$ and evaluate the functional at the nominal values $M_\infty = 0.83$, $\alpha = 2°$ (this cost depends on the flow regime: with higher Mach number the solver needs more iterations to achieve, if possible, the same error level on the residual), while to compute the adjoint state and the gradient we spent about 4 minutes. Regarding the Hessian, ToT and ToR gave us the same (wrong) numerical results but, as expected their cost differs: ToT required about 5.5 minutes while ToR required about 13 minutes. Therefore, to obtain the second-order Taylor expansion we used ToT and the runtime cost was about 13 minutes, while to built the fully nonlinear database the runtime cost was more than 23 hours.

It is interesting to note in this case that the ratio between the runtime cost for the gradient (or the cost for the Hessian) and the runtime cost for the nonlinear solution is higher with respect the first-order accuracy: this is due to the fact that the linear system in the algorithms for the gradient and the Hessian are solved using the preconditioner built with the first-order Jacobian, thus such second-order linear systems require more iterations to achieve the convergence (see Section 6.1 and Tables 6.3-6.8)

6.2.2 Testcase 2: SSBJ geometry

The Testcase 2 corresponds to the Supersonic Business Jet geometry (courtesy of Dassault Aviation), and a picture is given in Figure 6.5. The corresponding 3D unstructured mesh has 31643 nodes and 169161 elements, where its connectivity can be obtained by the the sparsity pattern of the first-order accurate Jacobian and is shown in Fig. 6.6.

Motivated by the problem encountered in the Testcase 1 when we used the second-order accurate scheme for the flow solver, we have evaluated the gradient and the Hessian of the drag coefficient respect to the angle of attack and the free-stream Mach number in different flow regimes. The angle of attack was $\alpha = 3°$ and the various regimes are given by different values for free-stream Mach numbers: from a transonic flow with $M = 0.8$ to a supersonic flow with $M = 1.6$. Due to the high computational cost, we didn't compute the databases of values obtained solving the Euler equations, but we validated the gradient and the Hessian using divided differences.

Quite surprisingly, this time we obtained the correct Hessian matrix for the whole set of

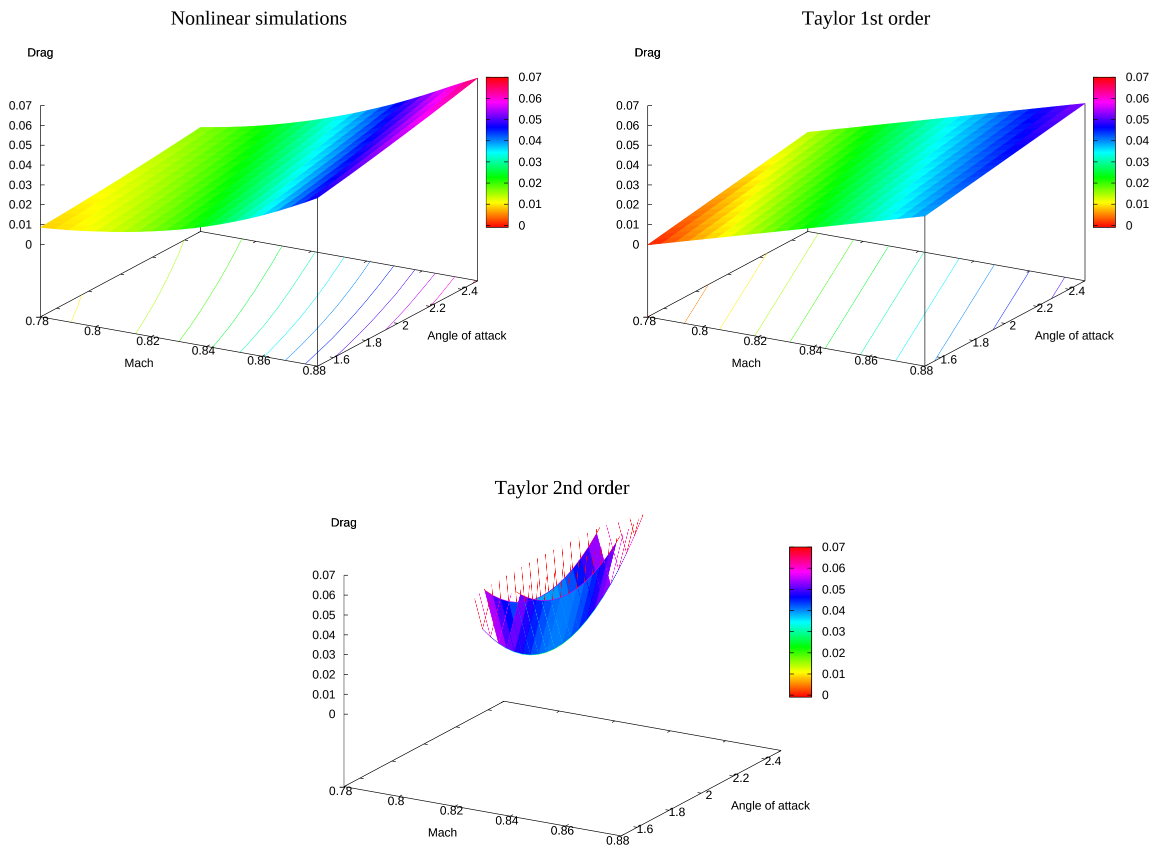


Figure 6.4: Drag coefficient vs. Mach number and angle of attack (second-order spatial accuracy) for the Piaggio wing: nonlinear simulations (top-left); first-order (top-right) and second-order (bottom-left) Taylor approximation around $\alpha = 2^\circ$ and $M = 0.83$. The second-order Taylor approximation is not accurate!.

free-stream Mach numbers, even for the critical value $M = 1.0$. Some plots of the first- and second-order Taylor expansion of the drag coefficient in function of the free-stream Mach number and angle of attack are given in Figs. 6.7-6.9.

This result enforces our suspect about the possibility of numerical truncation errors for the Hessian evaluation in the Testcase 1. However, we plan to repeat Hessian evaluation for the same geometry and flow regime of the Testcase 1 but with a different (finer) mesh.

6.3 Conclusions

In this Chapter we have presented some numerical experiments regarding the application of AD technique to solve linear systems and to compute first- and second-order derivatives of a functional subject to satisfy a set of nonlinear PDEs (the steady Euler equations).

For the solution of the linear systems (in which the matrix is a Jacobian) we have tested the matrix-free approach described in Section 3.3 in which a GMRES-RCI approach is used. The tests made involve the study of the ILU(p) preconditioner and of the dimension of the Krylov space. Regarding the level of filling in the ILU(p) strategies, we found that a too high value results in an algorithm that requires less iterations to converge but, due to the increased number of non-zero elements in the factorization, each iteration has an higher run-time cost; while a too low value results in a less robust algorithm. Regarding the dimension of the Krylov space d_K, we found that higher values give better performances but the algorithm require much memory. Due to the fact that we know in advance the quantity of memory needed by the GMRES algorithm, a good strategy for the choice of d_K could be the use of the highest possible value that is compatible with the available computational resources.

For the first- and second-order derivatives, we have tested and verified the algorithms in Chapter 3 for two grids and several flow regimes, but in one case the Hessian was not correct (ToT and ToR gave the same result). We think that this problem relies on numerical truncation errors. As we have seen, the run-time cost for the "gradient + complete Hessian" evaluation depends on the number of variables, and in our case (using ToT) we have obtained a cost that is $2 \div 3$ times the cost required for the solution of the steady Euler equations.

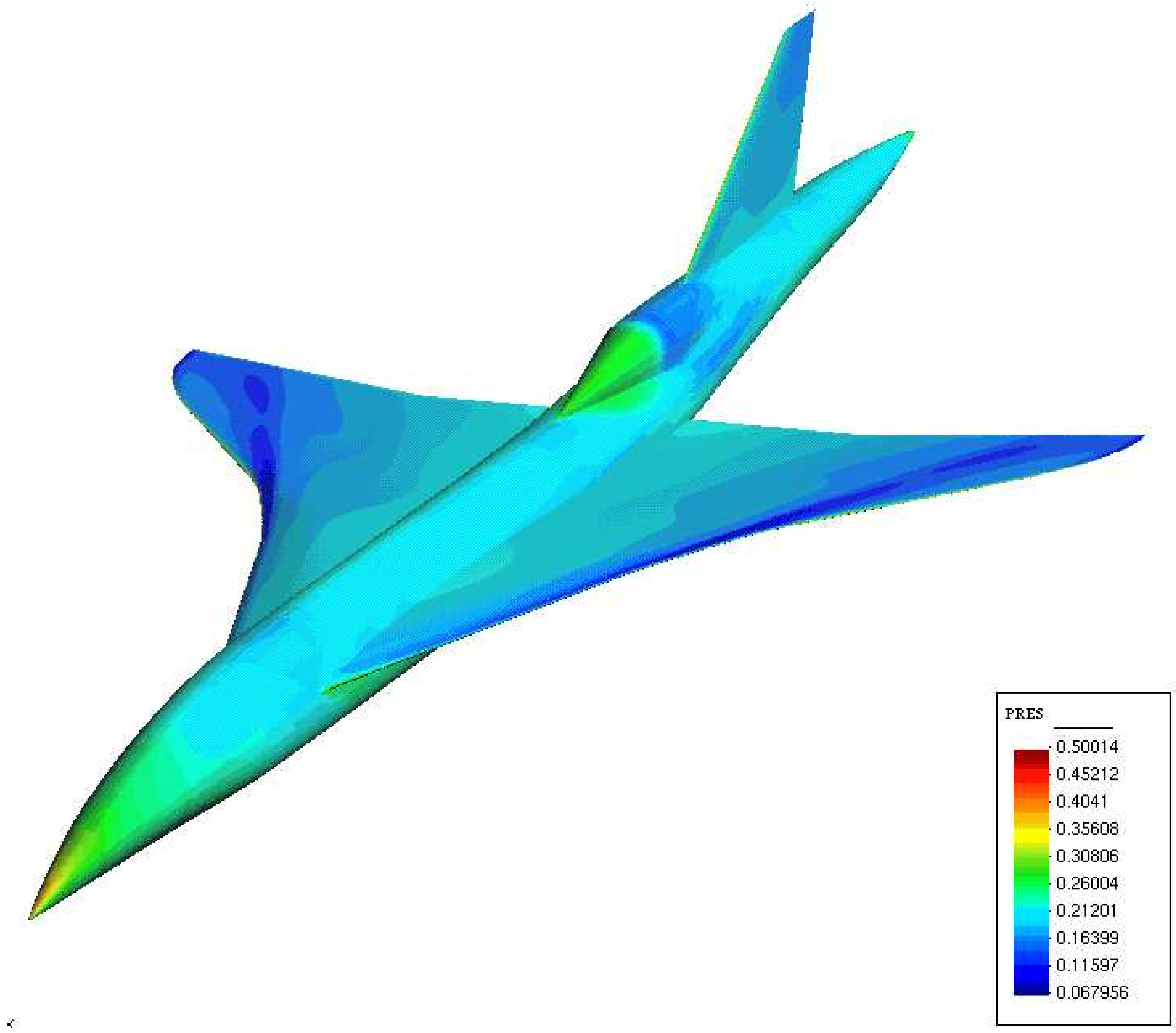


Figure 6.5: SSBJ geometry: pressure field on body+wings

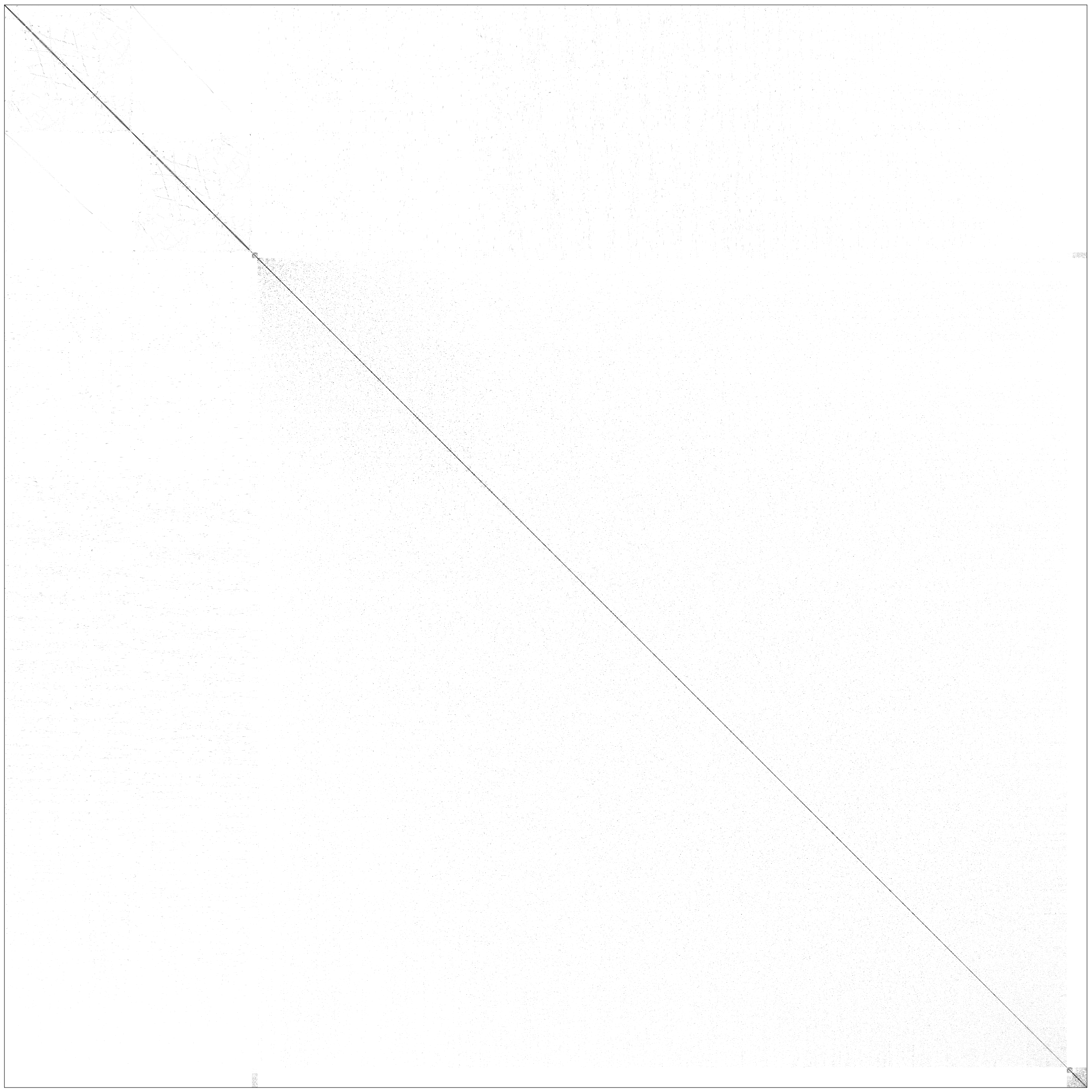


Figure 6.6: Sparsity pattern of the first-order Jacobian matrix $\frac{\partial \Psi^{(1)}}{\partial W}$ for the 3D unstructured mesh corresponding to the SSBJ geometry in Fig. 6.5. The matrix has a block-sparse structurally-symmetric pattern, with 448037 5×5-blocks (31643 blocks on the diagonal and 416394 off-diagonal).

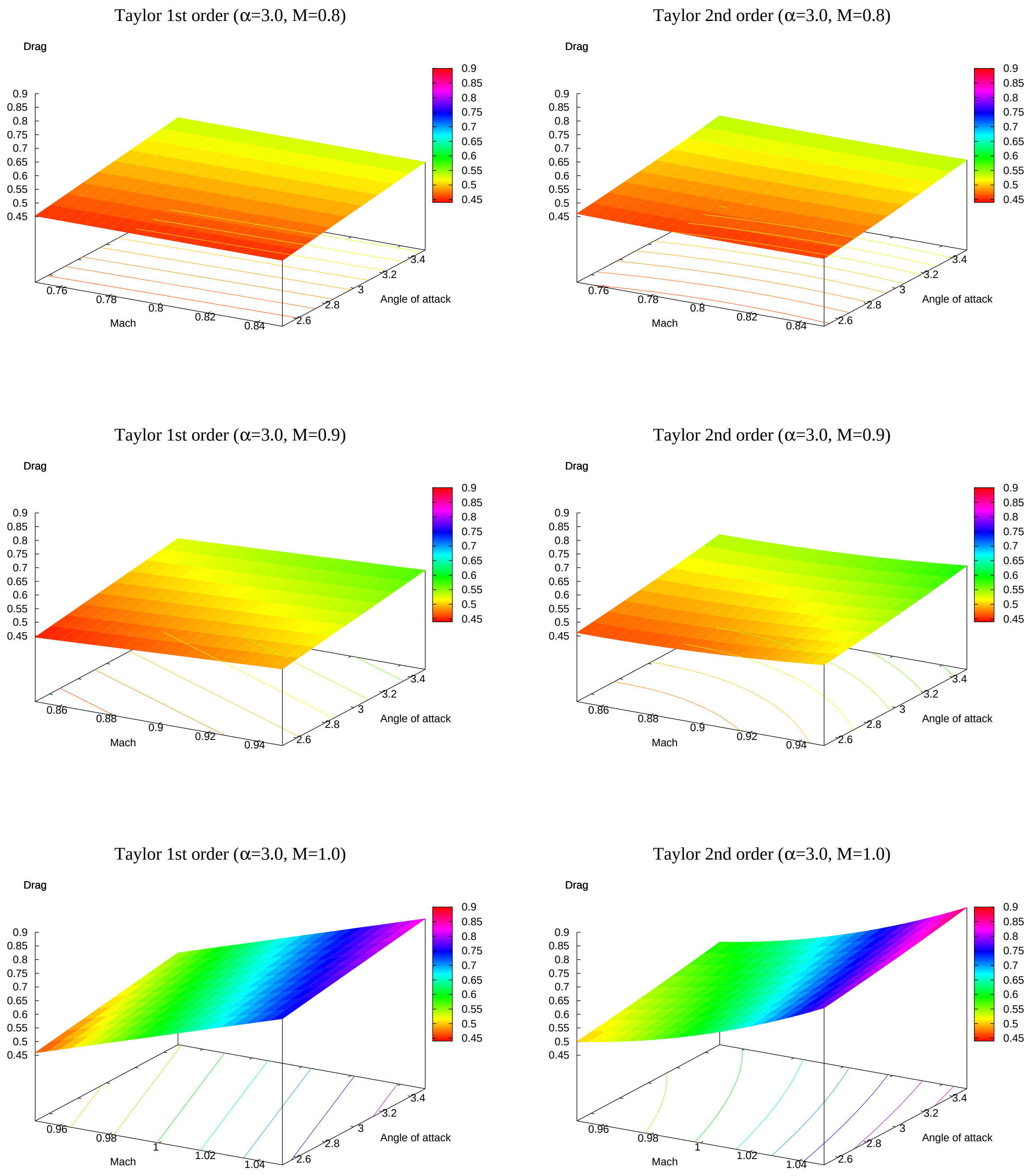


Figure 6.7: Drag coefficient vs. Mach number and angle of attack (second-order spatial accuracy) for the SSBJ mesh: first-order (left column) and second-order (right column) Taylor approximation around $\alpha = 3^\circ$ and $M = 0.8$ (top), $M = 0.9$ (center), $M = 1.0$ (bottom).

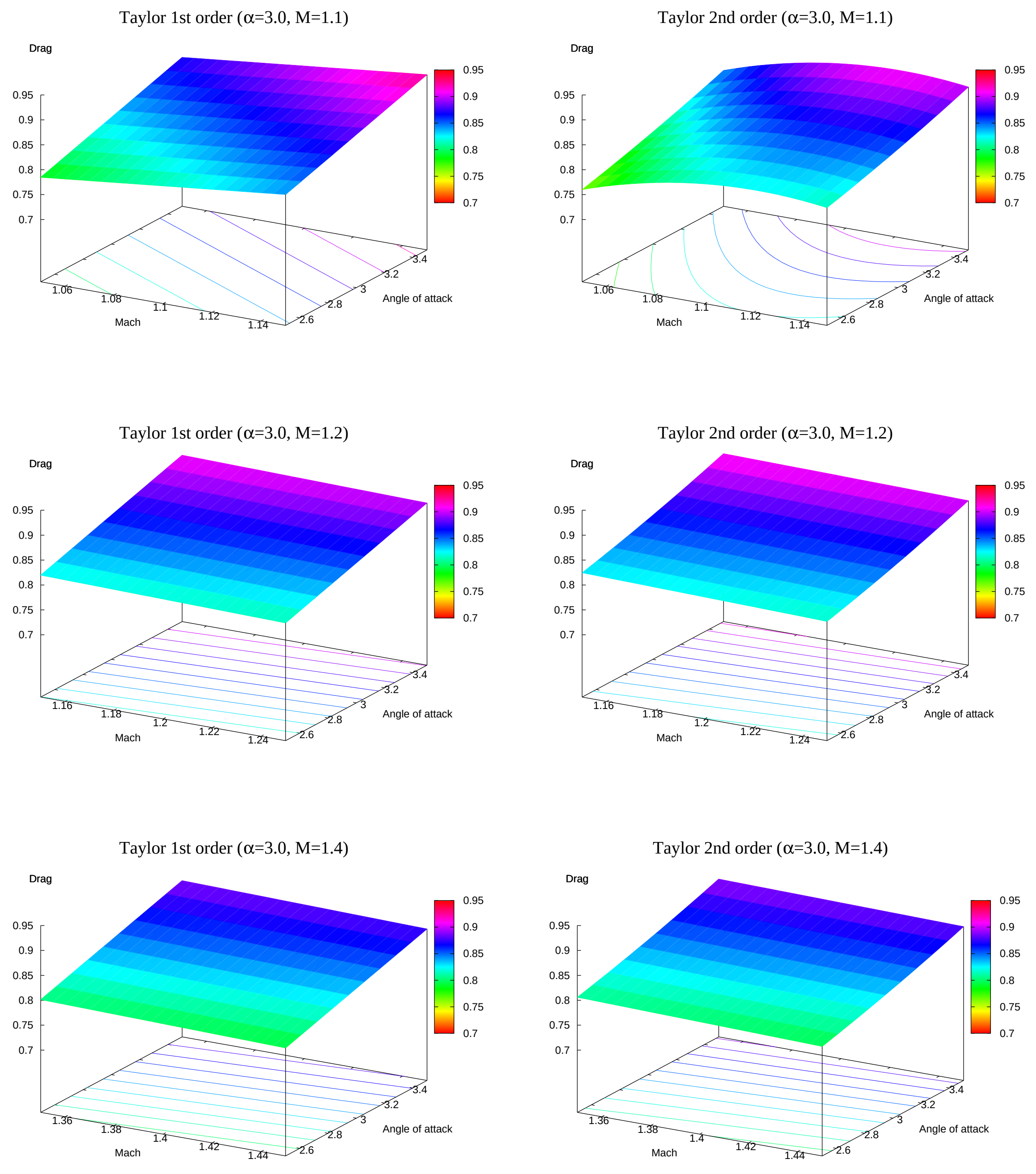


Figure 6.8: Drag coefficient vs. Mach number and angle of attack (second-order spatial accuracy) for the SSBJ mesh: first-order (left column) and second-order (right column) Taylor approximation around $\alpha = 3^\circ$ and $M = 1.1$ (top), $M = 1.2$ (center), $M = 1.4$ (bottom).

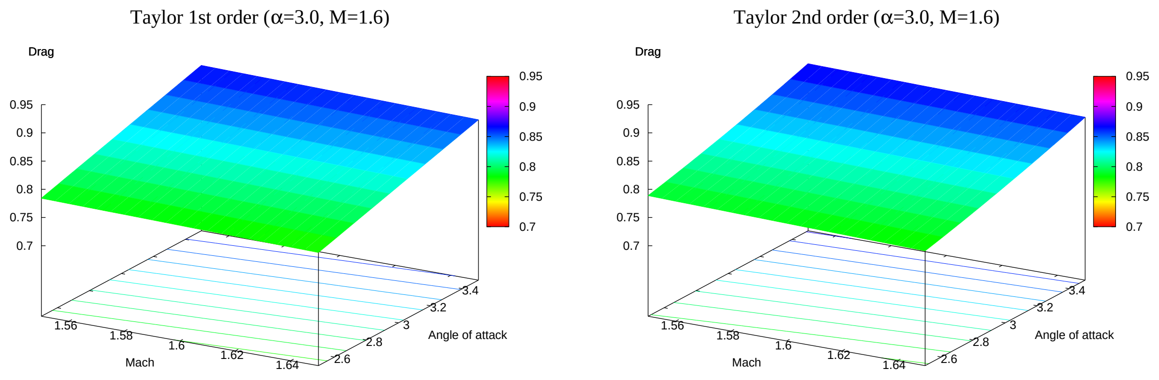


Figure 6.9: Drag coefficient vs. Mach number and angle of attack (second-order spatial accuracy) for the SSBJ mesh: first-order (left) and second-order (right) Taylor approximation around $\alpha = 3^\circ$ and $M = 1.6$.

Conclusions and future work

In this work we studied some algorithms and applications of first- and second-order derivatives in aerodynamic optimal design.

- In Chapter 1 we have described the mathematical flow model and the numerical finite-volume schemes implemented by our CFD codes (2D and 3D) for solving the steady Euler equations: this solution is implicit and relies on a preconditioned iteration.

- In Chapter 2 we have described some techniques developed for uncertainty analysis and propagation, and we have given some examples of techniques used for robust design, in which first- and second-order derivatives are required.

- In Chapter 3 we have started giving a brief description of the two differentiation mode of AD (Tangent and Reverse), then we have analyzed two different approaches for the differentiation of a functional subject to satisfy an equality constraint given by equations solved with fixed-point methods (like the steady Euler equations). In the first approach the entire process, involving the solution algorithm for the state equation and the evaluation of the functional, is considered to be implemented by a single program and it is differentiated as a whole. Conversely, the second approach (*differentiation of explicit parts*) considers the solution algorithm for the state equation and the functional evaluation as separate processes, and applies differentiation only to the routines which compute the state residual and the functional (in which the computational graph is assumed to be fixed). The first approach is easy to implement but has some reliability and performance problems: due to the presence of iterative processes to solve the state equations, we have not a fixed computational graph and therefore the AD theory is not well-founded. Moreover, if the differentiation would be correct, this approach requires the differentiation of an iterative algorithm that is very costly (in terms of CPU time and memory). To the opposite, the second approach results in a robust and cheaper way to perform differentiation (we need to evaluate the differentiated routines only at the final state) and it is designed to produce a matrix-by-vector output for the assembly of adjoint system.

  In Section 3.3 we have motivated and presented an approach (*matrix-free methods*) to solve the linear system $A\xi = b$ (or $A^T\xi = b$) when the matrix A is the Jacobian of some function. The key idea is to use iterative solvers (like GMRES) that do not require to know the matrix A but only the result of the matrix-by-vector multiplication. The availability of a preconditioner (built from the first-order Jacobian used for the pseudo-time iteration in the flow solver) is used to accelerate the convergence velocity. Some numerical experiments are presented in Section 6.1 in which we tested some ILU(p) factorizations for the preconditioner and different sizes for the Krylov space.

In Section 3.5 we recalled two strategies for computing the gradient of a constrained functional with AD: the first approach uses Tangent mode differentiation and its run-time cost is proportional to the number of independent variables; while the second approach is based on the adjoint formulation and uses Reverse mode differentiation resulting in a run-time cost that is independent from the number of independent variables.

Section 3.6.1 is devoted to describe an existing algorithm [Sherman et al., 1996], called Tangent-on-Tangent (forward-on-forward in [Ghate and Giles, 2007]) for computing the Hessian matrix. This approach is based on an adjoint formulation and requires a double Tangent-differentiation of the routines implementing the functional evaluation and the state residual. This approach computes the Hessian matrix element-by-element and for the full Hessian its run-time cost is due mainly to the sum of two different contributions: the solution of n linear systems (where n is the number of independent variables) and the evaluation of $\frac{n}{2}(n+1)$ double-differentiated routines. Due to the presence of a quadratic term in the expression for the cost, this approach is suited for the Hessian evaluation when the number of independent variables is not too high ($\lesssim 1000$).

In Section 3.5 we have developed a new approach based on Tangent-on-Reverse differentiation of the routines implementing the functional evaluation and the state residual. This approach builds the full Hessian matrix column-by-column, using Hessian-by-vector multiplication, in which the run-time cost of this multiplication is independent from the number of variables but it requires the solution of two linear systems. Therefore the run-time cost of this algorithm to evaluate the full Hessian is proportional to the number of independent variables.

From the previous arguments regarding the run-time cost, we have defined a mixed strategy for the full Hessian evaluation that combines the advantages of the two approaches. The proposed formulation coincides with pure ToT when the number of independent variables are small while the ToT/ToR strategy results more efficient than both ToT and ToR for a larger number of variables.

Using Tangent-on-Reverse differentiation we had some problems regarding the stack management performed by TAPENADE (the software used to perform Automatic Differentiation). In Section 3.7 we have described this issue and how to fix it by hand. Finally, in Section 3.8 we have described how to build a general framework that frees the user to implement the various algorithms for a specific problem, resulting in a smaller time requirement for the development and the implementation on existing codes.

Some numerical experiments regarding the first- and second-order differentiation using the previous algorithms on a 3D CFD code are presented in Sections 6.2.1 and 6.2.2.

- In Chapter 4 we have presented a new multilevel gradient-based method for aerodynamic shape design. Starting from an existing formulation [Beux and Dervieux, 1994] based on an embedded parametrization of shape grid-points and on interpolation operators, a possible generalisation to other kinds of parametrizations is described. Two particular examples of parametrizations are then presented, and since the shape grid-points are used as control variables, the resulting approaches can be interpreted as multilevel strategies as defined in [Beux and Dervieux, 1994], in which a particular prolongation operator (i.e. with a particular preconditioning) is applied. However, it can also be reinterpreted directly as multilevel approaches with respect to the new family of shape parametrization. In the first example, the sub-levels are defined through the use of Bézier control points, and starting from a consistent coarsest level, the degree-elevation property of Bézier curves is applied to successively define the different finer levels. In this context, even if, in practice, the Bézier control points can be not explicitly computed, the proposed algorithm can be also interpreted as a descent method for Bézier control points as control variables. In the second example, the definition of the set of sub-parametrizations is based on the use of an orthonormal basis of shape functions as shape representation. As for the case of Bézier-based parametrization, a descent direction is obtained considering as control parameters the ordinates of the shape grid-points as well as the finest sub-parametrization.

The numerical experiments shows that the new families of sub-parametrizations have suitable effects, if there are understood as an alternative gradient preconditioning for the optimization with respect to the shape grid-points. Nevertheless, to extend the range of interest of this kind of methods, they should be interpreted as descent methods in which the control variables are taken through the new set of parameters. Concerning the Bézier-based parametrization, the results are more disappointing with respect to the parametrization based on orthonormal shape functions, since the multilevelling seems poorly efficient. This is due, here, to a good convergence behaviour on the finest levels while the coarsest levels do not yield any additional speed-up, and thus, the basic conditions are not present to apply effectively the multilevel/multigrid principles. Thus, such additional investigation should be performed in order to better understand the present behaviour which is also inconsistent with the results obtained by J.-A. Désidéri and collaborators. If more attractive results can be obtained for Bézier-based parametrization, since the Bézier curves act as a basic tool for polynomial shape representation, one can also envisage to extend the formulation to more complex shape representation as B-splines (which also have properties of degree-elevation), and also, to 3D case through, for instance, tensorial Bézier parametrization. In the both examples of parametrizations presented in this study, the shape parameters are related by linear or affine application to the set of shape grid-points.

- In Chapter 5 we have developed a new algorithm to compute the gradient of an improved functional built as the sum of the original functional and an adjoint-correction term as defined in [Pierce and Giles, 2004, 2000]. The resulting algorithm uses Automatic Differentiation techniques and needs to evaluate the first-order derivative of the adjoint state, and therefore involves the evaluation of second-order derivatives of the original functional and state residual but in a new differentiation mode: Reverse-on-Tangent. The run-time cost for this algorithm is independent from the number of variables and is due mainly to the solution of four linear systems: one of them is for the gradient of the original functional while the others three are required for the gradient evaluation of the adjoint-correction term.

Due to the fact that the Reverse differentiation is performed after the Tangent one, and due to the fact that the Tangent differentiation does not change the structure of the original program, the Reverse-on-Tangent mode does not suffer from the stack management issue we have encountered for Tangent-on-Reverse.

We plan to use this algorithm to build a gradient-based descent method for optimal shape design. At this time we do not have numerical results.

- in Chapter 6 we have presented some numerical experiments regarding the application of Tangent-on-Tangent and Tangent-on-Reverse algorithm in order to evaluate the gradient and Hessian evaluation of the drag coefficient with respect to Mach number and angle of attack in which we used the 3D Euler solver described in Chapter 1. For our tests we have considered two different 3D geometries and different orders of spatial accuracy of the solution.

  As expected, the two algorithms for the Hessian evaluation gave the same numerical results, but with different runtime-costs: in our case with only two independent variables, the ToT approach was $2 \div 2.5$ times faster with respect to ToR. We have obtained correct results apart one case in which we had inaccurate values for the Hessian (but correct values for the gradient). We think that this problem could be due to some numerical truncation errors in the double differentiated routine, and we planned to perform further computations, using the same flow regime and geometry but with a different (and more refined) mesh.

  Moreover, we have presented some numerical experiments regarding the behaviour of the iterative linear solvers built using Automatic Differentiation and some ILU(p) strategies for preconditioning. We have solved tangent and adjoint linear systems using GMRES on first- and second-order accurate Jacobian (obtained by AD) in which the preconditioner is built from first-order accurate Jacobian (defect-correction approach). We have observed that the in our case the better strategy for preconditioning in terms of run-time cost is given by ILU(1) (while ILU(2) needs less iterations to converge but a greater run-time cost for each iteration). Regarding the performances difference between the solution of the tangent linear system $A\xi = b$ and the corresponding adjoint linear system $A^T\xi = c$, we have observed that the first case generally requires a greater number of iterations with respect the second one but a lower run-time cost, due to the fact that the matrix-by-vector multiplication is performed with differentiated routines and the cost of Tangent mode differentiation is usually lower with respect to Reverse mode.

The work done, demonstrates that Automatic Differentiation is mature enough to be used in a variety of applications and moreover, this Thesis describe in details the steps to apply. We recommend the proposed techniques for the Hessian evaluation for the following purposes:

- *robust optimization*: with the availability of the Hessian matrix we can build a gradient-based algorithm for shape optimization using the robust functional (2.21) and we can afford the Maximum Expected Value approach for robustness (Section 2.3);

- *adjoint-corrected functionals:* we want to build and use a gradient-based shape optimization algorithm for the adjoint-corrected functional (5.8)

Much more ambitious is the development of Automatic Differentiation techniques for the computation of second-order derivatives in the case of *unsteady flows* (e.g. Naviers-Stokes with turbulence models). Until now, only first-order differentiation has been performed for this kind of problems (see [Tber et al., 2007] for an Oceanographic application), and thus, the Hessian computation still represents a big challenge.

Appendices

Appendix A

Basic definitions in probability

The *cumulative distribution function* (CDF) of random variable X is defined as $F(x) = P(X \leq x)$. The *mathematical expectation* of a function of X, say $h(X)$ is defined as

$$E[h(X)] = \int h(x) dF(x)$$

where the notation $\int$ should be understood as a Stieltjes integral. In most applications, two cases are important:

- F is differentiable, in which case X is a *continuous* random variable and $f(x) = dF(x)/dx$ is called *probability density function* (PDF) of X. In such cases

$$E[h(X)] = \int h(x) f(h) dx$$

- F is a step function with countably many jumps of size $p_1, p_2, \ldots$ at $x_1, x_2, \ldots$ respectively, in which case X is a *discrete* random variable and

$$E[h(X)] = \sum_{i=1}^{\infty} p_i h(x_i)$$

The *mean* of a random variable X is defined as its expectation value $\mu = E[X]$. The *centered moment of order* r is defined as

$$E[(X-\mu)^r] \tag{A.1}$$

From the definition of expectation and because of the properties of the integrals, the centered moment of order 1 is always equal to zero. The centered moment of order 2 is called *variance* and holds $\sigma^2 = E[(X-\mu)^2] = E[X^2] - \mu^2$.

Theorem 2 (Chebyshev's inequality). *Let X be a random variable with expected value μ and finite variance σ^2. Then for any real number $k > 0$,*

$$\Pr(|X-\mu| \geq k\sigma) \leq \frac{1}{k^2} \tag{A.2}$$

Proof. For any event X, let $\chi(X)$ be the indicator random variable of X, i.e. $\chi(X)$ equals 1 if X occurs and 0 otherwise. Then

$$\begin{aligned}\Pr(|X-\mu| \geq k\sigma) &= E\big[\chi\big(|X-\mu| \geq k\sigma\big)\big] = E\big[\chi\big((X-\mu)^2/k^2\sigma^2 \geq 1\big)\big] \\ &\leq E\left[\frac{(X-\mu)^2}{k^2\sigma^2}\right] = \frac{1}{k^2\sigma^2}E\big[(X-\mu)^2\big] = \frac{1}{k^2}\end{aligned} \tag{A.3}$$

□

The theorem can be useful despite loose bounds because it applies to random variables of any distribution, and because these bounds can be calculated knowing no more about the distribution than the mean and variance.

Definition A.1. The sequence ξ_i $(i = 1, 2, \dots)$ of random variables converges P-*a.s.* (*almost surely* or *with probability one*) to the random variable ξ if $P\{\xi_i \to \xi\} = 1$, i.e. if the set for which ξ_i does not converge to ξ has probability zero.

Theorem 3 (**Strong Law of Large Numbers**). *Let* ξ_i $(i = 1, 2, \dots)$ *a sequence of independent identically distribuited random variables with* $E\big[|\xi_i|\big] < \infty$. *Then*

$$\lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N} \xi_i = \mu \quad (P\text{-}a.s.)$$

where $\mu = E\big[\xi_i\big]$.

Proof. See [Shirayev, 1996]. □

Appendix B

High-order approximations

Let $j \in C^{\infty}(\Omega, \mathbb{R})$ and $\Omega \subseteq \mathbb{R}^n$. The Taylor series expansion of j at the deterministic point $\mu_\gamma \in \Omega$ could be written as

$$j(\gamma) = A_0 + A_1 + A_2 + A_3 + A_4 + O(||\delta\gamma||^5) \tag{B.1}$$

where $\gamma = \mu_\gamma + \delta\gamma$ is a random vector such that $E\big[\gamma\big] = \mu_\gamma$ and

$$\begin{cases} A_0 = j(\mu_\gamma) \\ A_1 = \displaystyle\sum_i \frac{dj}{d\gamma^{(i)}}\Big|_{\mu_\gamma} \delta\gamma^{(i)} \\ A_2 = \displaystyle\frac{1}{2!}\sum_{i,k} \frac{d^2 j}{d\gamma^{(i)} d\gamma^{(k)}}\Big|_{\mu_\gamma} \delta\gamma^{(i)}\delta\gamma^{(k)} \\ A_3 = \displaystyle\frac{1}{3!}\sum_{i,k,l} \frac{d^3 j}{d\gamma^{(i)} d\gamma^{(k)} d\gamma^{(l)}}\Big|_{\mu_\gamma} \delta\gamma^{(i)}\delta\gamma^{(k)}\delta\gamma^{(l)} \\ A_4 = \displaystyle\frac{1}{4!}\sum_{i,k,l,m} \frac{d^4 j}{d\gamma^{(i)} d\gamma^{(k)} d\gamma^{(l)} d\gamma^{(m)}}\Big|_{\mu_\gamma} \delta\gamma^{(i)}\delta\gamma^{(k)}\delta\gamma^{(l)}\delta\gamma^{(m)} \end{cases} \tag{B.2}$$

where all derivatives are ealuated at μ_γ and the upperscript means $\delta\gamma^{(i)}$ for the i-th element of the vector $\delta\gamma \in \mathbb{R}^n$. From the above definitions, and remembering that the expectation acts only on the non-deterministic variables (i.e. on the $\delta\gamma$), we have the following properties

$$\begin{cases} E\big[A_0\big] = A_0 & \\ E\big[A_1\big] = 0 & \\ E\big[A_i\big] = O\big(E\big[\delta\gamma^i\big]\big) & \text{for } i \geq 2 \\ E\big[A_i A_k\big] = O\big(E\big[\delta\gamma^{i+k}\big]\big) & \text{for } i, k \geq 1 \end{cases} \tag{B.3}$$

The mean value of the functional is then

$$\mu_j = E\big[j\big] = A_0 + E\big[A_2\big] + E\big[A_3\big] + E\big[A_4\big] + O\big(E\big[\delta\gamma^5\big]\big) \tag{B.4}$$

or in a more explicit way

$$
\begin{aligned}
\mu_j = j(\mu_\gamma) &+ \frac{1}{2!}\sum_{i,k}\frac{d^2 j}{d\gamma^{(i)}d\gamma^{(k)}}\Big|_{\mu_\gamma} E\big[\delta\gamma^{(i)}\delta\gamma^{(k)}\big] + \\
&+ \frac{1}{3!}\sum_{i,k,l}\frac{d^3 j}{d\gamma^{(i)}d\gamma^{(k)}d\gamma^{(l)}}\Big|_{\mu_\gamma} E\big[\delta\gamma^{(i)}\delta\gamma^{(k)}\delta\gamma^{(l)}\big] + \\
&+ \frac{1}{4!}\sum_{i,k,l,m}\frac{d^4 j}{d\gamma^{(i)}d\gamma^{(k)}d\gamma^{(l)}d\gamma^{(m)}}\Big|_{\mu_\gamma} \big[\delta\gamma^{(i)}\delta\gamma^{(k)}\delta\gamma^{(l)}\delta\gamma^{(m)}\big] + O\big(E\big[\delta\gamma^5\big]\big)
\end{aligned}
$$

In order to compute the variance of j we need to multiply the equation (B.1) by itself, and keeping the lower order terms we obtain

$$
j^2(\gamma) = A_0^2 + A_1^2 + A_2^2 + 2A_0(A_1 + A_2 + A_3 + A_4) + 2A_1(A_2 + A_3) + O(||\delta\gamma||^5)
$$

Using some algebra and the properties (B.3)

$$
\begin{aligned}
E\big[j^2\big] &= A_0^2 + E\big[A_1^2\big] + 2A_0E\big[A_2\big] + 2E\big[A_1A_2\big] + 2A_0E\big[A_3\big] + \\
&\quad + E\big[A_2^2\big] + 2E\big[A_1A_3\big] + 2A_0E\big[A_4\big] + O\big(E\big[\delta\gamma^5\big]\big) \\
E\big[j\big]^2 &= A_0^2 + E\big[A_2\big]\big(E\big[A_2\big] + 2A_0\big) + E\big[A_3\big]\big(E\big[A_3\big] + 2E\big[A_2\big] + 2A_0\big) \\
&\quad + E\big[A_4\big]\big(E\big[A_4\big] + 2E\big[A_3\big] + 2E\big[A_2\big] + 2A_0\big) + O\big(E\big[\delta\gamma^5\big]\big)
\end{aligned}
$$

and finally the variance

$$
\begin{aligned}
\sigma_j^2 &= E\big[j^2\big] - E\big[j\big]^2 = \\
&= E\big[A_1^2\big] - E\big[A_2\big]^2 + 2E\big[A_1A_2\big] - E\big[A_3\big]\big(E\big[A_3\big] + 2E\big[A_2\big]\big) + \\
&\quad + E\big[A_2^2\big] + 2E\big[A_1A_3\big] - E\big[A_4\big]\big(E\big[A_4\big] + 2E\big[A_3\big] + 2E\big[A_2\big]\big) + O\big(E\big[\delta\gamma^5\big]\big)
\end{aligned}
\tag{B.5}
$$

It is interesting note here that if $\delta\gamma$ are normally distribuited then the moments of odd-order are zero [Shirayev, 1996]

$$
E\big[\delta\gamma^{2k-1}\big] = 0 \quad \Rightarrow \quad \begin{cases} E\big[A_{2k-1}\big] = 0 & \text{for } k \geq 1 \\ E\big[A_iA_k\big] = 0 & \text{for } i+k \text{ odd} \end{cases}
$$

and the mean and the variance approximations become

$$
\boxed{
\begin{aligned}
\mu_j &= E\big[j\big] = A_0 + E\big[A_2\big] + E\big[A_4\big] + O\big(E\big[\delta\gamma^6\big]\big) \\
\sigma_j^2 &= E\big[A_1^2\big] - E\big[A_2\big]^2 + E\big[A_2^2\big] + 2E\big[A_1A_3\big] + \\
&\quad - E\big[A_4\big]\big(E\big[A_4\big] + 2E\big[A_2\big]\big) + O\big(E\big[\delta\gamma^6\big]\big)
\end{aligned}
}
\tag{B.6}
$$

If $\delta\gamma$ is unidimensional then $E[\delta\gamma^{2k}] = \frac{(2k)!}{2^k k!}\sigma_\gamma^{2k}$ [Shirayev, 1996] and then we can study the needed derivatives for a given error order. Namely,

	μ_j	σ_j^2
$O(\sigma_\gamma^2)$	0	$--$
$O(\sigma_\gamma^4)$	$0, 2$	1
$O(\sigma_\gamma^6)$	$0, 2, 4$	$1, 2, 3$

where the numbers are the order of the derivatives needed for the required error order (the 0-th derivative is the functional itself).

To be more explicit, for the case where the uncertainties are random and normally distribuited, we have

$$\begin{aligned} E\big[A_1^2\big] &= \sum_{i,k} G_i G_k C_{ik} \\ E\big[A_2^2\big] - A\big[A_2\big]^2 &= \frac{1}{4} \sum_{i,k,l,m} H_{ik} H_{lm} (C_{il} C_{km} + C_{im} C_{kl}) \end{aligned} \tag{B.7}$$

where $G_i = \dfrac{\partial j}{\partial \gamma^{(i)}}\Big|_{\mu_\gamma}$ are the elements of the gradient, $H_{ik} = \dfrac{d^2 j}{d\gamma^{(i)} d\gamma^{(k)}}\Big|_{\mu_\gamma}$ are the elements of the Hessian matrix and $C_{ik} = E\big[\delta\gamma_u^{(i)} \delta\gamma_u^{(k)}\big] = \mathrm{cov}(\gamma_u^{(i)}, \gamma_u^{(k)})$ are the elements of the *covariance matrix*. Furthermore, if the (normal) uncertainties are independents, then the relation $C_{ik} = \sigma_i^2 \delta_{ij}$ holds, where $\sigma_i^2 = E\big[\delta\gamma^{(i)} \delta\gamma^{(i)}\big]$ and Equations (B.7) become

$$\begin{aligned} E\big[A_1^2\big] &= \sum_i G_i^2 \sigma_i^2 \\ E\big[A_2^2\big] - A\big[A_2\big]^2 &= \frac{1}{2} \sum_{i,k} H_{ik}^2 \sigma_i^2 \sigma_k^2 \end{aligned} \tag{B.8}$$

The last two equations are usually used in literature to define the First-Order and Second-Order Moments Method, where the term containing the third-order derivative $E\big[A_1 A_3\big]$ is neglected, namely

First-Order Moments Method

$$\begin{cases} \mu_j = j(\mu_\gamma) \\ \sigma_j^2 = \sum\limits_i G_i^2 \sigma_i^2 \end{cases}$$

Second-Order Moments Method

$$\begin{cases} \mu_j = j(\mu_\gamma) + \dfrac{1}{2} \sum\limits_i H_{ii} \sigma_i^2 \\ \sigma_j^2 = \sum\limits_i G_i^2 \sigma_i^2 + \dfrac{1}{2} \sum\limits_{i,k} H_{ik}^2 \sigma_i^2 \sigma_k^2 \end{cases}$$

Appendix C

Bash Scripts and Makefile to perform differentiation using TAPENADE

C.1 First-order differentiation

```
WORKDIR="/user/masmarti/home/Hessian_pgm_v2"

# TAPENADE command
TAPENADE="$1"

# name of the program
PROGRAM_NAME="$2"

# name of the directory containing the program to differentiate
PROGRAMDIR=$WORKDIR/"$3"

# name of the directory for the differentiated source code (first and second order)
WORKDIR_DIFF=$WORKDIR/"$4"

# name of the directory containing the link to the source code and headers files
SOURCE_DIR=$WORKDIR/"$5"

# name of the directory for the first order differentiated source code
FIRST_ORDER_DIR=$WORKDIR/"$6"

# name of the directory for the second order differentiated source code
# (not used in this script)
SECOND_ORDER_DIR=$WORKDIR/"$7"

# name of the file containing the implementation of
# the functional and the state residual
INPUT_FILE="$8"
```

```
# extension of the file $INPUT_FILE containing the implementation of
# the functional and the state residual
INPUT_FILE_EXTENSION="$9"

LOGFILE=$WORKDIR_DIFF"/first_order_derivatives.log"

# remove the existing directories for differentiated code
rm -r $WORKDIR_DIFF

mkdir $WORKDIR_DIFF
mkdir $SOURCE_DIR
mkdir $FIRST_ORDER_DIR


#INPUT_FILE="functional_and_state_residuals.f90"
ln -s $PROGRAMDIR"/"$INPUT_FILE $SOURCE_DIR"/"

# find the *.f files in the directoy $PROGRAMDIR and in its subdirectory
file_list_f=$(find $PROGRAMDIR"/" -name "*.f")

# find the *.h files in the directoy $PROGRAMDIR and in its subdirectory
file_list_h=$(find $PROGRAMDIR"/" -name "*.h")

# find the *.f90 files in the directoy $PROGRAMDIR and in its subdirectory
file_list_f90=$(find $PROGRAMDIR"/" -name "*.f90")

file_list=$file_list_f" "$file_list_h" "$file_list_f90

# make a soft link of the previous files (fortran sources and headers)
# in the directory $SOURCE_DIR
for file in $file_list
do
  ln -s $file $SOURCE_DIR"/"
done

# make a list of the *.f files
file_list_f=$(find $SOURCE_DIR"/" -name "*.f")

# make a list of the *.f90 files
file_list_f90=$(find $SOURCE_DIR"/" -name "*.f90")

file_list=$file_list_f" "$file_list_f90

# output directory for first_order_derivatives
OUTPUT_DIR=$FIRST_ORDER_DIR

LIST_VARIABLE_IN="gamma w"
LIST_VARIABLE_OUT="j psi"
```

```
LIST_SUBROUTINE="state_residuals func"

SUFFIX_TANGENT="_d"
SUFFIX_REVERSE="_b"

MSGLEVEL="20"

#FIXINTERFACE=""
FIXINTERFACE="-fixinterface"

# Copy the empty differentiated functions in the case of inactive input variables
# due to the fact that the ToT and/or ToR methods will refer to these functions.
# In other words, you MUST have these functions (empty or not) to properly link with
# the libraries
case $INPUT_FILE_EXTENSION in
  "f" ) EMPTYFILES_DIR=$WORKDIR"/Empty_files/fortran77" ;;
  "f90" ) EMPTYFILES_DIR=$WORKDIR"/Empty_files/fortran90" ;;
esac
cp $(find $EMPTYFILES_DIR"/first_order_derivatives/" -name \
   "*."$INPUT_FILE_EXTENSION) $FIRST_ORDER_DIR"/"

#
# Call TAPENADE to differentiate the source code
#
# The routines differentiated in TANGENT mode respect all its input
# variables will have the suffix "_d" after the name and the suffix "d"
# for the dot variables
#   Example: if we have the routine
#                func(j,gamma,w)
#   where J is an output variables and gamma and W are input variables,
#   the tangent mode differentiated version will be
#                func_d(j, jd, gamma, gammad, w, wd)
#
# The routines differentiated in REVERSE mode respect all its input
# variables will have the suffix "_b" after the name and the suffix "b"
# for the bar variables
#    Example: if we have the routine
#                func(j,gamma,w)
#    where J is an output variables and gamma and W are input variables,
#    the reverse mode differentiated version will be
#                func_b(j, jb, gamma, gammab, w, wb)
#
# In the case of differentiation (in tangent or reverse mode) of only one
# input variable, the name of the differentiated function will have the
# suffix "_d$nameofvariables" and then the suffix "_d" for the tangent mode
# and the suffix "_b" for the reverse mode
#    Example: the function func_dw_b(j, jb, gamma, w, wb, n_control, n_state) means
#             that the function func(j, gamma, w, n_control, n_state) is
#             differentiated with respect to the variable w (_dw) in reverse mode (_b)
```

```
#
#

for SUBROUTINE in $LIST_SUBROUTINE
do
# Separate differentiation respect to the variables "gamma" and "w"
  if [ "$SUBROUTINE" = "state_residuals" ]
  then
        VARIABLE_OUT="psi"
  elif [ "$SUBROUTINE" = "functional" ]
  then
      VARIABLE_OUT="j"
  fi
  for VARIABLE_IN in $LIST_VARIABLE_IN
  do
    NAME="_d"$VARIABLE_IN$SUFFIX_TANGENT
    $TAPENADE -d \
      -root $SUBROUTINE \
      -outvars $VARIABLE_OUT \
      -vars $VARIABLE_IN \
      -difffuncname $NAME \
      -O $OUTPUT_DIR \
      $FIXINTERFACE \
      -msglevel $MSGLEVEL \
      $file_list >> $LOGFILE
  done

  for VARIABLE_IN in $LIST_VARIABLE_IN
  do
    NAME="_d"$VARIABLE_IN$SUFFIX_REVERSE
    $TAPENADE -b \
      -root $SUBROUTINE \
      -outvars $VARIABLE_OUT \
      -vars $VARIABLE_IN \
      -difffuncname $NAME \
      -O $OUTPUT_DIR \
      $FIXINTERFACE \
      -numberpushpops \
      -msglevel $MSGLEVEL \
      $file_list  >> $LOGFILE
    done

  $TAPENADE -b \
    -root $SUBROUTINE \
    -outvars $VARIABLE_OUT \
    -vars """ $LIST_VARIABLE_IN """ \
    -difffuncname $SUFFIX_REVERSE \
    -O $OUTPUT_DIR \
    $FIXINTERFACE \
    -msglevel $MSGLEVEL \
    -numberpushpops \
    $file_list  >> $LOGFILE

  $TAPENADE -d \
    -root $SUBROUTINE \
    -outvars $VARIABLE_OUT \
    -vars """ $LIST_VARIABLE_IN """ \
    -difffuncname $SUFFIX_TANGENT \
    -O $OUTPUT_DIR \
    $FIXINTERFACE \
    -msglevel $MSGLEVEL \
    $file_list  >> $LOGFILE
done
```

C.2 Second-order differentiation

```
WORKDIR=‘pwd‘

# TAPENADE command
TAPENADE="$1"

# name of the program
PROGRAM_NAME="$2"

# name of the directory containing the program to differentiate
PROGRAMDIR=$WORKDIR/"$3"

# name of the directory for the differentiated source code (first and second order)
WORKDIR_DIFF=$WORKDIR/"$4"

# name of the directory containing the link to the source code and headers files
SOURCE_DIR=$WORKDIR/"$5"

# name of the directory for the first order differentiated source code
FIRST_ORDER_DIR=$WORKDIR/"$6"

# name of the directory for the second order differentiated source code
# (not used in this script)
SECOND_ORDER_DIR=$WORKDIR/"$7"

# name of the file containing the implementation of the functional
# and the state residual
INPUT_FILE="$8"

# extension of the file $INPUT_FILE containing the implementation of the functional
# and the state residual
INPUT_FILE_EXTENSION="$9"
```

```
LOGFILE=$WORKDIR_DIFF"/second_order_derivatives.log"
EMPTYFILES_DIR=$WORKDIR"/Empty_files/second_order_derivatives"

TAPENADEMISC_DIR=$WORKDIR"/TapenadeMisc"

rm -r $SECOND_ORDER_DIR
mkdir $SECOND_ORDER_DIR

# find the *.f files in the directoy $FIRST_ORDER_DIR and in its subdirectory
file_list_f=$(find $FIRST_ORDER_DIR"/" -name "*.f")

# find the *.h files in the directoy $FIRST_ORDER_DIR and in its subdirectory
file_list_h=$(find $FIRST_ORDER_DIR"/" -name "*.h")

# find the *.f90 files in the directoy $FIRST_ORDER_DIR and in its subdirectory
file_list_f90=$(find $FIRST_ORDER_DIR"/" -name "*.f90")

file_list=$file_list_f" "$file_list_h" "$file_list_f90

# make a soft link of the previous files (fortran sources and headers)
# in the directory $SOURCE_DIR
for file in $file_list
do
  ln -s $file $SOURCE_DIR"/"
done

# find the *.f files
file_list_f=$(find $SOURCE_DIR"/" -name "*.f")

# find the *.f90 files
file_list_f90=$(find $SOURCE_DIR"/" -name "*.f90")

file_list=$file_list_f" "$file_list_f90

OUTPUT_DIR=$SECOND_ORDER_DIR
LIST_VARIABLE_IN="gamma w"
LIST_SUBROUTINE="state_residuals func"
LIST_VARIABLE_OUT="gammab wb"

SUFFIX_TANGENT="_d"
SUFFIX_REVERSE="_b"
SUFFIX_TANGENT_W="_dw_d"

MSGLEVEL="20"

#FIXINTERFACE=""
FIXINTERFACE="-fixinterface"
```

```
# Copy the empty differentiated functions in the case of inactive input variables
# due to the fact that the ToT and/or ToR methods will refer to these functions.
# In other words, you MUST have these functions (empty or not) to properly link with
# the libraries
case $INPUT_FILE_EXTENSION in
  "f" ) EMPTYFILES_DIR=$WORKDIR"/Empty_files/fortran77" ;;
  "f90" ) EMPTYFILES_DIR=$WORKDIR"/Empty_files/fortran90" ;;
esac
cp $(find $EMPTYFILES_DIR"/second_order_derivatives/" -name "*."$INPUT_FILE_EXTENSION) \
   $SECOND_ORDER_DIR"/"


# Tangent on Reverse differentiation
for SUBROUTINE in $LIST_SUBROUTINE
do
  THIS_SUBROUTINE=$SUBROUTINE$SUFFIX_REVERSE
  $TAPENADE -d \
    -root $THIS_SUBROUTINE \
    -outvars """ $LIST_VARIABLE_OUT """ \
    -vars """ $LIST_VARIABLE_IN """ \
    -difffuncname $SUFFIX_TANGENT
    -ext $TAPENADEMISC_DIR"/PUSHPOPGeneralLib" \
    -extAD $TAPENADEMISC_DIR"/PUSHPOPADLib" \
    -O $OUTPUT_DIR \
    $FIXINTERFACE \
    -msglevel $MSGLEVEL \
    $file_list >> $LOGFILE
done


# Tangent on Tangent differentiation
for SUBROUTINE in $LIST_SUBROUTINE
do
  if [ "$SUBROUTINE" = "state_residuals" ]
  then
    VARIABLE_OUT="psid"
  elif [ "$SUBROUTINE" = "functional" ]
  then
    VARIABLE_OUT="jd"
  fi
  THIS_SUBROUTINE=$SUBROUTINE$SUFFIX_TANGENT
  $TAPENADE -d \
    -root $THIS_SUBROUTINE \
    -outvars """ $VARIABLE_OUT """ \
    -vars """ $LIST_VARIABLE_IN """ \
    -difffuncname $SUFFIX_TANGENT \
    -ext $TAPENADEMISC_DIR"/PUSHPOPGeneralLib" \
    -extAD $TAPENADEMISC_DIR"/PUSHPOPADLib" \
    -O $OUTPUT_DIR  \
    $FIXINTERFACE \
    -msglevel $MSGLEVEL \
    $file_list >> $LOGFILE
done


# Reverse on Tangent differentiation
for SUBROUTINE in $LIST_SUBROUTINE
do
  if [ "$SUBROUTINE" = "state_residuals" ]
  then
    VARIABLE_OUT="psid"
  elif [ "$SUBROUTINE" = "functional" ]
  then
    VARIABLE_OUT="jd"
  fi
  THIS_SUBROUTINE=$SUBROUTINE$SUFFIX_TANGENT_W
  $TAPENADE -b \
    -root $THIS_SUBROUTINE \
    -outvars """ $VARIABLE_OUT """ \
    -vars """ $LIST_VARIABLE_IN """ \
    -difffuncname $SUFFIX_REVERSE \
    -ext $TAPENADEMISC_DIR"/PUSHPOPGeneralLib" \
    -extAD $TAPENADEMISC_DIR"/PUSHPOPADLib" \
    -O $OUTPUT_DIR \
    $FIXINTERFACE \
    -msglevel $MSGLEVEL \
    $file_list >> $LOGFILE
done
```

C.3 Makefile

```
WORKDIR :=
PROGRAM_NAME := famosa

INPUT_FILE_NAME := functional_and_state_residuals
INPUT_FILE_EXTENSION := f
INPUT_FILE := $(INPUT_FILE_NAME).$(INPUT_FILE_EXTENSION)

PROGRAMDIR := $(PROGRAM_NAME)

RELEASE := Yes

TAPENADE_PATH := /user/masmarti/home/Tapenade/sources/topLevel
TAPENADE := $(TAPENADE_PATH)/mytapenade

SHELL := /bin/sh
```

```
AR := ar

CC := icc
CFLAGS = -shared-intel -mcmodel=large -parallel

FC := ifort
FFLAGS = -shared-intel -mcmodel=large -autodouble -132 -Warn -parallel

IPOFLAGS = -c
ifeq ($(RELEASE),Yes)
 CFLAGS += -O3 -xT -align
 FFLAGS +=  -O3 -xT -align all
 IPOFLAGS +=
else
  CFLAGS += -O0
  FFLAGS +=  -fp-model strict -O0 -check
endif

F90 := $(FC)
F90FLAGS := $(FFLAGS)

%.o : %.f
        $(FC) $(FFLAGS) $(IPOFLAGS) -o $@ $<

%.o : %.f90
        $(F90) $(F90FLAGS) $(IPOFLAGS) -o $@ $<

%.o : %.c
        $(CC) $(CFLAGS) $(IPOFLAGS) -o $@ $<


(%.o) : %.f
        $(FC) $(FFLAGS) $(IPOFLAGS) $< -o $*.o
        $(AR) r $@ $*.o

(%.o) : %.f90
        $(F90) $(F90FLAGS) $(IPOFLAGS) $< -o $*.o
        $(AR) r $@ $*.o

(%.o) : %.c
        $(CC) $(CFLAGS) $(IPOFLAGS) $< -o $*.o
        $(AR) r $@ $*.o

#-----------------------------------------------------------------------------
WORKDIR_DIFF := $(PROGRAMDIR)_diff
SOURCE_DIR := $(WORKDIR_DIFF)/src
FIRST_ORDER_DIR := $(WORKDIR_DIFF)/first_order_derivatives
SECOND_ORDER_DIR := $(WORKDIR_DIFF)/second_order_derivatives
LIB_DERIVATIVES_DIR := $(WORKDIR_DIFF)
```

```
LIB_FIRSTDERIVATIVES := $(LIB_DERIVATIVES_DIR)/libfirstderivatives.a
LIB_SECONDDERIVATIVES := $(LIB_DERIVATIVES_DIR)/libsecondderivatives.a

.PHONY: all
all:
        make derivatives
        make all_libraries

.PHONY: all_libraries
all_libraries:
        make library_derivatives
        make library_algorithms
        make library_tapenademisc

.PHONY: derivatives
derivatives: $(PROGRAMDIR)/$(INPUT_FILE)
        make firstderivatives
        make secondderivatives

.PHONY: firstderivatives
firstderivatives: firstorder_derivatives.sh
        . firstorder_derivatives.sh $(TAPENADE) \
            $(PROGRAM_NAME) \
            $(PROGRAMDIR) \
            $(WORKDIR_DIFF) \
            $(SOURCE_DIR) \
            $(FIRST_ORDER_DIR) \
            $(SECOND_ORDER_DIR) \
            $(INPUT_FILE) \
            $(INPUT_FILE_EXTENSION)

.PHONY: secondderivatives
secondderivatives: secondorder_derivatives.sh
        . secondorder_derivatives.sh $(TAPENADE) \
            $(PROGRAM_NAME) \
            $(PROGRAMDIR) \
            $(WORKDIR_DIFF) \
            $(SOURCE_DIR) \
            $(FIRST_ORDER_DIR) \
            $(SECOND_ORDER_DIR) \
            $(INPUT_FILE) \
            $(INPUT_FILE_EXTENSION)


$(LIB_FIRSTDERIVATIVES): first_order_derivatives_sources_f := \
        $(wildcard $(FIRST_ORDER_DIR)/*.f)
$(LIB_FIRSTDERIVATIVES): first_order_derivatives_objects_f := \
        $(patsubst %.f, %.o, $(first_order_derivatives_sources_f))
$(LIB_FIRSTDERIVATIVES): first_order_derivatives_sources_f90 := \
        $(wildcard $(FIRST_ORDER_DIR)/*.f90)
$(LIB_FIRSTDERIVATIVES): first_order_derivatives_objects_f90 := \
        $(patsubst %.f90, %.o, $(first_order_derivatives_sources_f90))
$(LIB_FIRSTDERIVATIVES): first_order_derivatives_objects := \
        $(first_order_derivatives_objects_f) $(first_order_derivatives_objects_f90)

$(LIB_SECONDDERIVATIVES): second_order_derivatives_sources_f := \
        $(wildcard $(SECOND_ORDER_DIR)/*.f)
$(LIB_SECONDDERIVATIVES): second_order_derivatives_objects_f := \
        $(patsubst %.f, %.o, $(second_order_derivatives_sources_f))
$(LIB_SECONDDERIVATIVES): second_order_derivatives_sources_f90 := \
        $(wildcard $(SECOND_ORDER_DIR)/*.f90)
$(LIB_SECONDDERIVATIVES): second_order_derivatives_objects_f90 := \
        $(patsubst %.f90, %.o, $(second_order_derivatives_sources_f90))
$(LIB_SECONDDERIVATIVES): second_order_derivatives_objects := \
        $(second_order_derivatives_objects_f) $(second_order_derivatives_objects_f90)


$(LIB_FIRSTDERIVATIVES):
        for object in $(first_order_derivatives_objects) ; do \
          make "$(LIB_FIRSTDERIVATIVES)($$object)" ; \
        done
        rm $(first_order_derivatives_objects)

$(LIB_SECONDDERIVATIVES):
        for object in $(second_order_derivatives_objects) ; do \
          make "$(LIB_SECONDDERIVATIVES)($$object)" ; \
        done
        rm $(second_order_derivatives_objects)

.PHONY: library_derivatives
library_derivatives: $(LIB_FIRSTDERIVATIVES) $(LIB_SECONDDERIVATIVES)
#----------------------------------------------------

#----------------------------------------------------
MATRIXFREE_DIR := Matrixfree_solvers
LIB_MATRIXFREE_DIR := $(MATRIXFREE_DIR)/Lib
LIB_MATRIXFREE := $(LIB_MATRIXFREE_DIR)/libmatrixfreesolvers.a

matrixfree_sources_f := $(wildcard $(MATRIXFREE_DIR)/*.f)
matrixfree_objects_f := $(patsubst %.f, %.o, $(matrixfree_sources_f))
matrixfree_sources_f90 := $(wildcard $(MATRIXFREE_DIR)/*.f90)
matrixfree_objects_f90 := $(patsubst %.f90, %.o, $(matrixfree_sources_f90))
matrixfree_objects := $(matrixfree_objects_f) $(matrixfree_objects_f90)

$(LIB_MATRIXFREE): $(LIB_MATRIXFREE)($(matrixfree_objects))

.PHONY: library_matrixfreesolvers
library_matrixfreesolvers: $(LIB_MATRIXFREE)
```

```
#-----------------------------------------------------

#-----------------------------------------------------
GRADIENTHESSIAN_DIR := Gradient_Hessian
LIB_GRADIENTHESSIAN_DIR := $(GRADIENTHESSIAN_DIR)/Lib
LIB_GRADIENTHESSIAN := $(LIB_GRADIENTHESSIAN_DIR)/libgradienthessian.a

gradienthessian_sources_f := $(wildcard $(GRADIENTHESSIAN_DIR)/*.f)
gradienthessian_objects_f := $(patsubst %.f, %.o, $(gradienthessian_sources_f))
gradienthessian_sources_f90 := $(wildcard $(GRADIENTHESSIAN_DIR)/*.f90)
gradienthessian_objects_f90 := $(patsubst %.f90, %.o, $(gradienthessian_sources_f90))
gradienthessian_objects := $(gradienthessian_objects_f) $(gradienthessian_objects_f90)

$(LIB_GRADIENTHESSIAN): $(LIB_GRADIENTHESSIAN)($(gradienthessian_objects))

.PHONY: library_gradienthessian
library_gradienthessian: $(LIB_GRADIENTHESSIAN)
#-----------------------------------------------------

#-----------------------------------------------------
ADJOINTCORRECTION_DIR := AdjointCorrection
LIB_ADJOINTCORRECTION_DIR := $(ADJOINTCORRECTION_DIR)/Lib
LIB_ADJOINTCORRECTION := $(LIB_ADJOINTCORRECTION_DIR)/libadjointcorrection.a

adjointcorrection_sources_f := $(wildcard $(ADJOINTCORRECTION_DIR)/*.f)
adjointcorrection_objects_f := $(patsubst %.f, %.o, $(adjointcorrection_sources_f))
adjointcorrection_sources_f90 := $(wildcard $(ADJOINTCORRECTION_DIR)/*.f90)
adjointcorrection_objects_f90 := $(patsubst %.f90, %.o, $(adjointcorrection_sources_f90))
adjointcorrection_objects := $(adjointcorrection_objects_f) $(adjointcorrection_objects_f90)

$(LIB_ADJOINTCORRECTION): $(LIB_ADJOINTCORRECTION)($(adjointcorrection_objects))

.PHONY: library_adjointcorrection
library_adjointcorrection: $(LIB_ADJOINTCORRECTION)
#-----------------------------------------------------


#-----------------------------------------------------
VALIDATION_DIR := Validation
LIB_VALIDATION_DIR := $(VALIDATION_DIR)/Lib
LIB_VALIDATION := $(LIB_VALIDATION_DIR)/libvalidation.a

validation_sources_f := $(wildcard $(VALIDATION_DIR)/*.f)
validation_objects_f := $(patsubst %.f, %.o, $(validation_sources_f))
validation_sources_f90 := $(wildcard $(VALIDATION_DIR)/*.f90)
validation_objects_f90 := $(patsubst %.f90, %.o, $(validation_sources_f90))
validation_objects := $(validation_objects_f) $(validation_objects_f90)

$(LIB_VALIDATION): $(LIB_VALIDATION)($(validation_objects))
```

```
.PHONY: library_validation
library_validation: $(LIB_VALIDATION)
#------------------------------------------------------

#----------------------------------------------------
library_algorithms:
        make library_matrixfreesolvers
        make library_gradienthessian
        make library_adjointcorrection
        make library_validation
#----------------------------------------------------

#-----------------------------------------------------------------------------
TAPENADEMISC_DIR := TapenadeMisc
LIB_TAPENADEMISC_DIR := $(TAPENADEMISC_DIR)/Lib
LIB_TAPENADEMISC := $(LIB_TAPENADEMISC_DIR)/libtapenademisc.a

tapenademisc_objects := $(TAPENADEMISC_DIR)/adBuffer.o \
                        $(TAPENADEMISC_DIR)/adStack.o \
                        $(TAPENADEMISC_DIR)/PUSHPOPDiff.o

$(LIB_TAPENADEMISC): $(LIB_TAPENADEMISC)($(tapenademisc_objects))

.PHONY: library_tapenademisc
library_tapenademisc: $(LIB_TAPENADEMISC)
#-----------------------------------------------------------------------------

#-----------------------------------------------------------------------------
FUNCTIONAL_AND_STATE_RESIDUALS_DIR := .
LIB_FUNCTIONAL_AND_STATE_RESIDUALS_DIR := $(FUNCTIONAL_AND_STATE_RESIDUALS_DIR)
LIB_FUNCTIONAL_AND_STATE_RESIDUALS := \
       $(LIB_FUNCTIONAL_AND_STATE_RESIDUALS_DIR)/libfuncttional_and_state_residuals.a
.PHONY: library_functional_and_state_residuals
library_functional_and_state_residuals: \
       $(LIB_FUNCTIONAL_AND_STATE_RESIDUALS)(functional_and_state_residuals.o)
#-----------------------------------------------------------------------------

#-----------------------------------------------------------------------------
LIB_SPARSKIT_DIR := /home/masmarti/MatrixLibs/SPARSKIT2
#-----------------------------------------------------------------------------

#-----------------------------------------------------------------------------
LIB_MKL_DIR= /user/masmarti/home/intel/mkl/9.1/lib/32
MKL_LIB= $(LIB_MKL_DIR)/libmkl_lapack.a \
         $(LIB_MKL_DIR)/libmkl_ia32.a \
         $(LIB_MKL_DIR)/libguide.so \
         /lib/libpthread.so.0 -lm
#-----------------------------------------------------------------------------
```

```
#-----------------------------------------------------------------------------
.Phony: Test
LIBRARY_DIRS := -L$(LIB_VALIDATION_DIR)/ \
                -L$(LIB_GRADIENTHESSIAN_DIR)/ \
                -L$(LIB_MATRIXFREE_DIR)/ \
                -L$(LIB_DERIVATIVES_DIR)/ \
                -L$(LIB_TAPENADEMISC_DIR) \
                -L$(LIB_SPARSKIT_DIR)

LIBRARY_FILES := $(LIB_VALIDATION)/ \
                 $(LIB_GRADIENTHESSIAN) \
                 $(LIB_MATRIXFREE) \
                 $(LIB_SECONDDERIVATIVES) \
                 $(LIB_FIRSTDERIVATIVES) \
                 $(LIB_TAPENADEMISC)

LIBRARY_DEPS := $(LIBRARY_FILES)

LIBRARIES:= -lvalidation \
            -lgradienthessian \
            -lmatrixfreesolvers \
            -lseconddderivatives \
            -lfirstderivatives \
            -ltapenademisc \
            -lskit \
            $(MKL_LIB)

Test: Test/flux_solver.o \
        Test/program_with_gradient.o \
        $(PROGRAMDIR)/$(INPUT_FILE_NAME).o \
        $(LIBRARY_DEPS)
        $(F90) $(F90FLAGS) -o Test/Test_hessian Test/program_with_gradient.o \
        Test/flux_solver.o \
        $(PROGRAMDIR)/$(INPUT_FILE_NAME).o \
        $(LIBRARY_DIRS) $(LIBRARIES)
#-----------------------------------------------------


.PHONY: cleanall
cleanall:
        -rm -r $(WORKDIR_DIFF)
        make clean
        make removelibs

.PHONY: clean
clean:
        -rm $(PROGRAMDIR)/*.o
        -rm $(PROGRAMDIR)/*.*~
```

```
.PHONY: removelibs
removelibs:
        -rm $(WORKDIR_DIFF)/*.a
        -rm $(LIB_VALIDATION_DIR)/*.a
        -rm $(LIB_GRADIENTHESSIAN_DIR)/*.a
        -rm $(LIB_MATRIXFREE_DIR)/*.a
        -rm $(LIB_TAPENADEMISC_DIR)/*.a
        -rm $(LIB_ADJOINTCORRECTION_DIR)/*.a
        make removeobjects

.PHONY: removeobjects
removeobjects:
        -rm $(VALIDATION_DIR)/*.o
        -rm $(GRADIENTHESSIAN_DIR)/*.o
        -rm $(MATRIXFREE_DIR)/*.o
        -rm $(TAPENADEMISC_DIR)/*.o
        -rm $(ADJOINTCORRECTION_DIR)/*.o
```

Bibliography

B. Abou El Majd, J.-A. Désidéri, and A. Janka. Nested and self-adaptive Bézier parametrization for shape optimization. In *International Conference on Control, Partial Differential Equations and Scientific Computing*, Bejiing, China. 13-16 September 2004.

B. Abou El Majd, J.-A. Désidéri, and R. Duvigneau. Multilevel strategies for parametric shape optimizations in Aerodynamics. *Revue Européenne de Mécanique Numérique - European Journal of Computational Mechanics*, 17(1-2):153–172, 2008.

N. Alexandrov, R. Lewis, C. Gumbert, L. Green, and P. Newman. Approximations and model management in aerodynamic optimisation with variable-fidelity models. *Journal of Aircraft*, 38(6):1093–1110, 2001.

M. Andreoli, A. Janka, and J.-A. Désidéri. Free-form deformation parametrization for multilevel 3D shape optimization in aerodynamics. Technical Report 5019, INRIA, November 2003.

H. Bandemer. *Mathematics of Uncertainty.* Springer, 2006.

J. W. Barrett, G. Moore, and K. W. Morton. Optimal recovery in the finite element method, part 2: Defect correction for ordinary differential equations. *IMA Journal of Numerical Analysis*, 8:527–540, 1988.

R. Becker and R. Rannacher. *An optimal control approach to error control and mesh adaptation.* Cambridge University Press, 2001.

R. Becker, H. Kapp, and R. Rannacher. Adaptive finite element methods for optimal control of partial differential equations: basic concepts. *SIAM Journal of Control and Optimization*, (39):113–132, 2000.

F. Beux. Shape optimization of an Euler flow in a nozzle. *Notes on Numerical Fluid Mechanics*, 55:115–131, 1994.

F. Beux and A. Dervieux. Exact-gradient shape optimization for a 2-D Euler flow. *Finite Elements in Analysis and Design*, 12(3-4):281–302, 1992.

F. Beux and A. Dervieux. A hierarchical approach for shape optimization. *Engineering Computations*, 11(6):25–38, 1994.

H.-G. Beyer and B. Sendhoff. Robust optimization – A comprehensive survey. *Comput. Methods Appl. Mech. Engrg.*, 196:3190–3218, 2007.

A. Borzi. *Multigrid methods for optimality systems.* Habilitation thesis. Univ. Graz, 2003.

G. Carpentieri, M. van Tooren, and B. Koren. Adjoint-based aerodynamic shape optimization on unstructured meshes. *Journal of Computational Physics*, 22(1):267–287, May 2007. ISSN 0021-9991.

L. Catalano, A. Dadone, and V. Dalosio. Progressive optimization on unstructured grid using multigrid-aided finite-difference sensitivities. *International Journal for Numerical Methods in Fluids*, 47(10-11):1383–1391, 2005.

L. Catalano, A. Dadone, and V. Dalosio. Turbine cascade design via multigrid-aided finite-difference progressive sensitivities. *Revue Européenne de Mécanique Numérique - European Journal of Computational Mechanics*, 17(1-2):203–220, 2008.

I. Chang, F. Torres, and C. Tung. Geometric analysis of wing sections. Technical Report 110346, NASA, April 1995.

P. H. Cournède, B. Koobus, and A. Dervieux. Positivity statements for a mixed-element-volume scheme on fixed and moving grids. *Revue Européenne de Mécanique Numérique - European Journal of Computational Mechanics*, 15(7-8):767–798, 2006.

F. Courty and A. Dervieux. Multilevel functional preconditioning for shape optimization. *International Journal of Computational Fluid Dynamics*, 20(7):481–490, 2006.

A. Dadone and B. Grossman. Progressive optimization of inverse fluid dynamic design problems. *Computaters and Fluids*, 29(1):1–32, 2000.

M. de' Michieli Vitturi and F. Beux. A discrete gradient-based approach for aerodynamic shape optimisation in turbulent viscous flows. *Finite Elements in Analysis and Design*, 43(1):68–80, 2006.

J.-A. Désidéri. *Hierarchical optimum-shape alghorithms using embedded Bézier parametrizations.* CIMNE, Barcelona, 2003.

J.-A. Désidéri. Two-level ideal alghorithm for parametric shape optimizations. *to appear in Journal of Numerical Mathematics*, 2007.

J.-A. Désidéri and P. W. Hemker. Analysis of the convergence of iterative implicit and defect-correction algorithms for hyperbolic problems. *SIAM Journal of Scientific Computation*, 16 (1):88–118, 1995.

J. Dongarra, V. Eijkhout, and A. Kalhan. Reverse Communication Interface for Linear Algebra Templates for Iterative Methods. Technical Report UT-CS-95-291, May 1995.

T. Dreyer, B. Maar, and V. Schulz. Multigrid optimization in applications. *J. Comp. Appl.*, 120:67–84, 2000.

R. Duvigneau. Aerodynamic Shape Optimization with Uncertain Operating Conditions Using Metamodels. Technical Report 6143, INRIA, March 2007.

R Dwight and J. Brézillon. Effect of various approximations of the discrete adjoint on gradient-based optimization. *AIAA Journal*, 44(12):3022–3071, December 2006.

G. Farin. *Curves and surfaces for computer-aided geometric design – A practical guide*. Academic Press, 1990.

D. Feng and T. Pulliam. Aerodynamic design optimization via reduced Hessian SQP with solution refining. Technical Report 95-24, Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, 1995.

L. Fézoui and A. Dervieux. Finite-element non oscillatory schemes for compressible flows. In *Computational Mathematics and Applications. Proceedings of* 8^{th} *France-U.S.S.R.-Italy Joint Symposium.*, number 730, Pavia, 1989. C.N.R.-Istituto di Analisi Numerica.

L. Fézoui and B. Stoufflet. A class of implicit upwind schemes for Euler simulations with unstructured meshes. *Journal of Computational Physics*, 84:174–206, 1989.

V. Frayssé, L. Giraud, S. Gratton, and J. Langou. A Set of GMRES Routines for Real and Complex Arithmetics on High Performance Computers. Technical Report TR/PA/03/3, CERFACS, 2003.

G-077-1998. *AIAA Guide for the Verification and Validation of Computational Fluid Dynamics Simulations*. American Institute of Aeronautics & Astronautics, 1998. ISBN 1563472856.

V. E. Garzon. *Probabilistic Aerothermal Design of Compressor Airfoils*. PhD thesis, MIT, 2003.

E. Gelman and J. Mandel. On multilevel iterative methods for optimization problems. *Mathematical Programming*, 48(1):1–17, 1990.

D. Ghate and M. B. Giles. *Inexpensive Monte Carlo uncertainty analysis*, pages 203–210. Recent Trends in Aerospace Design and Optimization. Tata McGraw-Hill, New Delhi, 2006.

D. Ghate and M. B. Giles. Efficient Hessian Calculation Using Automatic Differentiation. Number 2007-4059. AIAA, June 2007. 25th Applied Aerodynamics Conference, Miami (Florida).

R. Giering, T. Kaminski, and T. Slawig. Generating efficient derivative code wit TAF: adjoint and tangent linear Euler flow around an airfoil. *Future generation computer systems*, 21(8): 1345–1355, 2005.

M. B. Giles and N. A. Pierce. Adjoint error correction for integral outputs. In T. Barth and H. Deconinck, editors, *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*, volume 25 of *Lecture Notes in Computational Science and Engineering*, pages 47–96. Springer-Verlag, 2002.

M. B. Giles and N. A. Pierce. An introduction to the adjoint approach to design. *Flow, Turbulence and Combustion*, 65:393–415, 2000.

E. Godlewski and P. A. Raviart. *Numerical Approximation of Hyperbolic Systems of Conservation Laws.* Springer, 1996. ISBN 0-387-94529-6.

S. Gratton, A. Sartenaer, and P. Toint. Recursive trust-region methods for multilevel nonlinear optimization (part I): global convergence and complexity. Technical Report 04/06, Dep. Math. Univ. Namur, 2004.

A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*, volume 19 of *Frontiers in Applied Mathematics*. SIAM Philadelphia, 2000.

A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, and Walther A. ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM TOMS*, 22(2):131–167, 1996. The updated article is available online at http://www.math.tu-dresden.de/∼adol-c/adolc110.ps.

A. Griewank, N. R. Gauger, and Riehme J. Extension of fixed point PDE solvers for optimal design by single-step one-shot method. *Revue Européenne de Mécanique Numérique - European Journal of Computational Mechanics*, 17(1-2):87–103, 2008.

H. Guillard. Convergence analysis of a multi-level relaxation method. Technical Report 1884, INRIA, 1993.

H. Guillard and N. Marco. Some aspects of multigrid methods on non-structured meshes. In *Proceedings of the Conference of Copper Mountains on Multigrid Methods.* NASA, April 1995.

A. Hall. On an experimental determination of π. *Messeng. Math.*, (2):113–114, 1873.

L. Hascoët and V. Pascual. TAPENADE 2.1 user's guide. Technical Report 0300, INRIA, Sep 2004.

L. Hascoët, R.-M. Greborio, and V. Pascual. *Computing Adjoints by Automatic Differentiation with TAPENADE.* Springer, 2005.

C. Held, , and A. Dervieux. One-Shot airfoil optimization without adjoint. *Computaters and Fluids*, 31(8):1015–1049, 2002.

R. Hicks and P. Henne. Wing design by numerical optimisation. *Journal of Aircraft*, 15(7): 407–413, 1978.

L. Huyse. Free-form airfoil shape optimization under uncertainty using maximum expected value and second-order second-moment strategies. Technical Report 2001-211020, NASA, Jun 2001. ICASE Report No. 2001-18.

L. Huyse, S. L. Padula, R. Michael Lewis, and W. Li. Probabilistic approach to free-form airfoil shape optimization under uncertainty. *AIAA Journal*, 40(9):1764–1772, 2002.

R. B. Kearfoot. Interval copmputations: Introduction, uses and resources. *Euromath Bullettin*, 2(1):95–112, 1996.

D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357–397, 2004.

B. Koobus, N. Marco, and A. Dervieux. An additive multilevel preconditioning method. *Journal of Scientific Computing*, 12(3):233–251, 1997.

B. Koren. Defect correction and multigrid for an efficient and accurate computation of airfoil flows. *Journal of Computational Physics*, 76, 1988.

G. Kuruvila, S. Ta'asan, and M. Salas. Airfoil optimization by the one-shot method. *Von Karman Institute Lecture*, 1994.

J. S. Liu. *Monte Carlo Strategies in Scientific Computing.* Springer-Verlag, 2001. ISBN 0-387-95230-6.

N. Marco and A. Dervieux. Multilevel parametrization for aerodynamical optimization of 3D shapes. *Finite Elements in Analysis and Design*, 26:259–277, 1999.

R. Martin and H. Guillard. Second-order defect-correction scheme for unsteady problems. *Computer & Fluids*, 25(1):9–27, 1996.

M. Martinelli and F. Beux. Multilevel gradient-based methods in aerodynamic shape design. submitted to ESAIM: Proceedings, 2006.

N. Metropolis. The beginning of the Monte Carlo method. *Los Alamos Science*, (15):125–130, 1987.

A. Migdalas, P. pardalos, and P. Värbrand. *Multilevel optimization: algorithms and applications.* Luwer Academic, 1997.

B. Mohammadi. A new optimal shape design procedure for inviscid and viscous turbulent flows. *International Journal for Numerical Methods in Fluids*, 25:183–203, 1997.

B. Mohammadi and O. Pironneau. *Applied Shape Optimization for Fluids.* Oxford University press, 2001.

R. E. Moore. *Interval Arithmetic and Automatic Error Analysis in Digital Computing.* PhD thesis, Stanford University, 1962.

J. Moran. *An Introduction to Theoretical and Computational Aerodynamics.* John Wiley, 1984.

S. Nash. A multigrid approach to discretized optimization problems. *Journal of Optimization Methods and Software*, 14:99–116, 2000.

N. Nemec and D. Zingg. Newton-Krylov algorithm for aerodynamic design under the Navier-Stokes equations. *AIAA Journal*, 37:1146–1154, 2002.

J. Newman III, A. C. Taylor III, R. Barnwell, P. A. Newman, and G.-W. Hou. Overview of sensitivity analysis and shape optimization for complex aerodynamic configurations. *Journal of Aircraft*, 36(1):87–96, 1999.

J. C. Newman III, W. K. Anderson, and D. L. Whitfield. Multidisciplinary sensitivity derivatives using complex variables. Technical Report MSSU-COE-ERC-98-08, Mississippi State University, July 1998.

W. L. Oberkampf and F. G. Blottner. Issues in computational fluid dynamics code verification and validation. *AIAA Journal*, 36(5):687–695, 1998.

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

N. A. Pierce and M. B. Giles. Adjoint and defect error bounding and correction for functional estimates. *Journal of Computational Physics*, 200:769–794, 2004.

O. Pironneau and E. Polak. Consistent approximations and approximate functions and gradients in optimal control. *SIAM Journal on Control and Optimization*, 41(2):487–510, 2002.

M. M. Putko, P. A. Newman, A. C. Taylor III, and L. L. Green. Approach for uncertainty propagation and robust design in CFD using sensitivity derivatives. Technical Report 2528, AIAA, 2001.

J. Reuther and A. Jameson. Aerodynamic shape optimization of wing and wing-body configurations using control theory. *AIAA Paper*, (95-0123), 1995.

P. L. Roe. Approximate Riemann solvers, parameter vectors and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981.

P.L. Roe and J. Pike. Efficient construction and utilisation of approximate riemann solutions. In *Proceedings of* 6th *international symposium on computing methods in applied sciences and engineering*, Computing Methods in Applied Science and Engineering, pages 499–518, Versailles, France, 1985. Springer.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996. ISBN 0-534-94776-X.

Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Technical report, 1994. Avalaible online at http://www-users.cs.umn.edu/∼saad/software/SPARSKIT/paper.ps.

J. Samareh. Survey of shape parametrization techniques for high-fidelity multidisciplinary shape optimization. *AIAA Journal*, 39(5):387–397, 2001.

M. Schwabacher and A. Gelsey. Multilevel simulation and numerical optimization of complex engineering designs. *Journal of Aircraft*, 35(3):387–397, 1998.

F. C. Schweppe. *Uncertain Dynamical Systems*. Prentice Hall, Engelwood Cliffs, 1973.

V. Selmin. Geometry modelling and industrial parametrisation issues. *Revue Européenne de Mécanique Numérique - European Journal of Computational Mechanics*, 17(1-2):131–152, 2008.

L. L. Sherman, A. C. Taylor III, L. L. Green, and P. A. Newman. First and second-order aerodynamic sensitivity derivatives via automatic differentiation with incremental iterative methods. *Journal of Computational Physics*, 129:307–331, 1996.

A. N. Shirayev. *Probability*. Springer-Verlag, second edition, 1996. ISBN 0-387-94549-0.

R. D. Skeel. A theoretical framework for proving accuracy results for deferred corrections. *SIAM Journal of Numerical Analysis*, 19:171–196, 1981.

W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Rev.*, 10(1):110–112, 1998.

L. G. Stanley and D. L. Stewart. *Design Sensitivity Analysis*. SIAM, 2002. ISBN 0-89871-524-5.

J. Steger and J. F. Warming. Flux vector splitting of the inviscid gas dynamics equation with application to finite difference methods. *Journal of Computational Physics*, 40:263–293, 1981.

H. J. Stetter. The defect correction principle and discretization methods. *Numerical Mathematics*, 29:425–443, 1978.

L. Su and J. E. Renaud. Automatic Differentiation in Robust Optimization. *AIAA Journal*, 35 (6), 1997.

Z. Tang and J.-A. Désidéri. Towards self-adaptive parametrization of Bézier curves for airfoil aerodynamic design. Technical Report 4572, INRIA, 2002.

M. H. Tber, L. Hascoët, A. Vidard, and B. Dauvergne. Building the Tangent and Adjoint codes of the Ocean General Circulation Model OPA with the Automatic Differentiation tool TAPENADE. Technical report, INRIA, Sep 2007.

E. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer, 1999. ISBN 3-540-65966-8.

J. Utke. OpenAD: Algorithm implementation user guide. Technical Memorandum ANL/MCS–TM–274, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL., 2004. Avalaible online at ftp://info.mcs.anl.gov/pub/tech_reports/reports/TM-274.pdf.

B. van Leer. Flux-vector splitting for the Euler equations. *Eighth International Conference of Numerical Methods in Fluid Dynamics, Lecture Note in Physics*, 170:505–512, 1982.

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

B. van Leer, J. L. Thomas, P. L. Roe, and R. W. Newsome. A comparison of numerical flux formulas for the Euler and Navier-Stokes equations. Technical Report 87-1104, AIAA, 1987.

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

R. W. Walters and L. Huyse. Uncertainty analysis for fluid mechanics with applications. Technical Report 2002-211449, NASA, Feb 2002. ICASE Report No. 2002-1.