Modeling and Simulation in Science, Engineering and Technology

Series Editor

Advisory Editorial Board

Emmanuel Laporte
Patrick Le Tallec

Numerical Methods in Sensitivity Analysis and Shape Optimization

Springer Science+Business Media, LLC

Emmanuel Laporte
Délégation Générale
pour L'Armement DSA/SPAé
75509 Paris Cedex 15
France
emmanuel.laporte@dga.defense.gouv.fr

Patrick Le Tallec
Ecole Polytechnique
91 128 Palaiseau Cedex
France
Patrick.leTallec@polytechnique.fr

Library of Congress Cataloging-in-Publication Data

Numerical methods in sensitivity analysis and shape optimization / Patrick Le Tallec, Emmanuel Laporte, authors.
p. cm. – (Modeling and simulation in science, engineering & technology)
Includes bibliographical references and index.
Additional material to this book can be downloaded from http://extras.springer.com.
ISBN 978-1-4612-6598-6 ISBN 978-1-4612-0069-7 (eBook)
DOI 10.1007/978-1-4612-0069-7
1. Sensitivity theory (Mathematics) 2. Interior-point methods. 3. Mathematical optimization. 4. Numerical analysis. I. Le Tallec, Patrick. II. Laporte, Emmanuel, 1971- III. Series.

QA402.3 .N85 2002
003'.5–dc21 2002038614

AMS Subject Classifications: 90C51, 90C53, 65K10, 74P10, 90C90, 90C30, 90C06, 76M12, 65M50, 65M60, 74F10

Printed on acid-free paper.

Originally published by Birkhäuser Boston in 2003
Softcover reprint of the hardcover 1st edition 2003

ISBN 978-1-4612-6598-6 SPIN 10901649

Typeset by the authors.

9 8 7 6 5 4 3 2 1

Contents

Outline and Notation

Context

Sensitivity analysis and optimal shape design are key issues in today's engineering. The first challenge is to reduce design problems set in a complex engineering world to a simple mathematical problem. Such a task is hard and problem dependent. Let us consider for example the shape optimization of a missile or of an aircraft wing. The engineer must consider many flight configurations characterized, among other parameters, by their Mach, Reynolds numbers, or angle of attack, and then ensure that the proposed design behaves well in all these configurations, with respect to payloads, total mass, and flight characteristics. Doing this in a unique general mathematical framework turns out to be impossible. What is more realistic is to proceed in successive steps. First, one identifies the major dimensioning flight configurations, design variables and constraints on a first reference design. Global optimization tools such as simulated annealing, regularization techniques [4] or evolutionary algorithms can be used here to select a first draft and to define the topology of the product under study. The next step is to start from this reference design, and improve its performances or relax key constraints by tuning physically significant design parameters. Mathematical programming techniques [18] can be used at this second level to compute the sensitivity of the basic design to local changes of data and to improve this first draft within the topological data and constraints identified in the first design step or imposed by the designer based on previous experience. The mathematical framework in which to study such *local shape optimization problems* was in fact set up in the early 1960s [6, 42]. But the practical implementation of these ideas has been hampered for many years due to the difficulty of finding efficient minimizers and computing accurate gradients. The situation has changed recently due to progress in discretization and solution strategies [27], optimization algorithms [5], and automatic differentiation [39]. Indeed, within an optimal design framework, a practical optimization strategy must combine:

- an adequate discretization algorithm reducing the original physical problem to a standard constrained finite dimensional optimization problem, governed by a small set of simple geometric variables. For this purpose, one must in particular introduce specific tools to smoothly deform any admissible configuration and update its corresponding discretization grid or finite element mesh;

- an efficient solver for the complex state (partial differential) equations governing for a given shape the behavior and performance of the considered system;
- an efficient optimization algorithm. Interior point techniques are very adequate for this purpose since they have good convergence properties while respecting all design constraints during the whole optimization process;
- a fast calculation of gradients.

In fact, such optimal shape design strategies, looking for a local minimum of the approximate cost function among all local admissible designs, might not be fully relevant in engineering practice. On the one hand, the exact local minimization may lead to a very small decrease of the cost function in comparison to the added complexity induced by actual implementation of the corresponding optimal design. On the other hand, the mathematical formulation of the local optimization problem may omit untold practical constraints, which will be known to the designer but not taken into account by the computer. The local optimization problem can also be a part of a complex multipoint design methodology involving several conflicting optimization processes. *Sensitivity analysis*, as described in [29], is a common piece of the answers for all these situations. By computing gradients, sensitivity analysis measures the variation of performance induced by a small perturbation of the control parameters. Gradients are also a major ingredient in reliability analysis for computing a first order random model, identifying the most probable failure mechanisms from known stochastic distributions of the data. Finally, gradients are also needed in most algorithms used in mathematical programming, and more generally in most multipoint optimization techniques. Therefore, optimal shape design must be complemented by efficient techniques for calculating gradients. To improve the crude estimates obtained by finite differences, one can follow two fundamental approaches. If the functions to be differentiated are direct output of computer programmes, these gradients can be computed automatically by differentiating each line of these computer programmes. For complex cost functions used in shape optimization and involving elaborate solvers, automatic differentiation must be coupled with Lagrangian techniques.

Aims

In the above framework, our purpose is to give an up-to-date description of a methodology that can be efficiently used for the local optimal design or the sensitivity analysis of a system. We suppose that a reference shape has been decided and that the key design constraints have been identified. The systems under consideration will be governed by the general partial differential equations used in computational structural or fluid mechanics. The methodology to be described will cover the following aspects: control of the shape's deformation by spline interpolation, and mesh transformation strategies; constrained optimization introducing the (Euler–Lagrange) optimality conditions and solving them by interior point techniques; Lagrangian approach and automatic code differentiation for practical calculation of the gradients of the cost functions,

and of the design constraints. The methodology will be illustrated by several significant examples, such as inverse diffusion problems, drag reduction, or aeroelastic stability.

Our hope is that the monograph will help the reader to understand, use and validate the fundamental numerical tools used in sensitivity analysis or in shape optimization. This optimization process being very user dependent, any insight gained by the reader will help in adapting such tools to chosen optimization goals, and in controlling the relevance and efficiency of the resulting process. The goal here is not to get an absolute optimum, but to construct better, efficient and robust designs, or at least to understand the effects of local changes of design. For this purpose, the monograph describes and motivates the use of basic mathematical tools, so that the reader can understand the underlying structure of the problems under consideration. All chapters are complemented by examples or exercises illustrating the practical and numerical aspects of the proposed methods. Programming software building or updating computational grids, solving constrained optimization problems, performing automatic code differentiation, computing basic aerodynamic or aeroelastic solutions are also described. These programs can be obtained on the internet at www.inria.fr and are also included on the enclosed CD-ROM. Therefore, the reader has the opportunity to test and experiment with the practical behavior of important numerical tools, such as mesh generation, mesh deformation, code differentiation, or simple cfd simulations.

The materials covered in the present monograph have been presented in different courses taught by the authors to undergraduate, graduate students, and practicing engineers. The common background of the students (and the background that will be assumed for the reader) is a basic knowledge of differential calculus, elementary numerical analysis, and an interest in computational mechanics. Numerical tests performed as student group activities have been a key factor in the success and understanding of the different courses. In fact, we believe that a deep knowledge of the field can only be reached through such exercises. This justifies the presence of the enclosed CD-ROM which makes it possible for any reader to test the different tools on a personal computer.

Detailed contents

The approach followed by the monograph is progressive. The first chapter defines the mathematical framework. Using a relevant example, it identifies the design parameters, the equations of state governing the considered system, the cost function to be minimized and the algebraic constraints to be satisfied, and explains how to reduce the original design problem to the local constrained optimization of a regular cost function with respect to a reasonable number of design parameters. This is achieved by the introduction, at the end of the chapter, of shape parametrizations based on spline functions and of various mesh construction and deformation strategies. This approach, illustrated by different exercises, is flexible and avoids all hard theoretical issues related to the use of infinite dimensional design spaces.

The following three chapters introduce the basic mathematical programming techniques to be used to solve the above constrained minimization problem.

These chapters are rather self contained and can also be used as an introduction to the numerical solution of generic constrained optimization problems. The mathematical characterizations of the solutions of finite dimensional optimization problems are first recalled in Chapter 2, leading to the so-called Euler–Lagrange equations of optimality. Newton's algorithms are reviewed in Chapter 3, together with various implementation details, relevant to their applications to minimization problems: BFGS construction of approximate Hessians, line search strategies, or GMRES algorithms for the solution of the underlying algebraic problems. Finally, interior point algorithms are introduced in Chapter 4, as they turn out to be very efficient for solving the constrained minimization problems arising in optimal design, and for handling unilateral constraints.

A key point is then to calculate gradients. Computing the gradients of cost functions or of design constraints is needed in sensitivity analysis, in multipoint optimization, or simply in order to run any basic interior point algorithm. These gradients are difficult to get because computing a typical cost function or design constraint usually requires a complex algorithm involving sophisticated solvers for the solution of the underlying equations of state. Two complementary techniques are then needed. The first one, described in Chapter 5, is automatic code differentiation, constructing automatically a software for computing gradients or partial derivatives by differentiating the original numerical solver line by line. After a brief modelization of the structure of a computer code, these automatic differentiation techniques are introduced in both direct and reverse mode. The complementary technique, introduced in Chapter 6, uses adjoint states and Lagrangians to get at low cost the gradient of a functional depending on the solution of a nonlinear implicit equation.

The following chapter illustrates the different optimization and differentiation tools introduced in the monograph (space discretization, grid deformation, constrained optimization, gradient calculations) by solving model problems of increasing complexity: the inverse problems in diffusion, drag reduction, and aeroelastic stability of turbine blades.

In all these chapters, the state variables are eliminated from the problem, and the underlying problem is reduced to a small scale minimization problem, at the cost of the exact solution of a complex equation of state at each step of the proposed algorithms. The last chapter briefly introduces the so-called one shot methods, which avoid this cost of an exact solution of the equation of state by solving the equations of state and the equations of optimality in a single sweep. The proposed technique, still under development, treats the equations of state by Lagrange multipliers, and solves the extended system of Euler–Lagrange equations by generalized interior point techniques.

Notation

Throughout the text, we will use standard mathematical notation. The dot product $a \cdot b = \sum_{i=1}^{n} a_i b_i$ of two given vectors $a = (a_i)_{i=1,n}$ and $b = (b_i)_{i=1,n}$ of R^n will be denoted either by $a \cdot b$ or by $\langle a, b \rangle$. In matrix notation, we have $a \cdot b = a^t b$. The notation $B \cdot d$ (respectively $d \cdot B$) denotes the dot product $B \cdot d = \sum_{j=1}^{n} B_{ij} d_j$ (respectively $d \cdot B = \sum_{j=1}^{n} d_j B_{ji}$) of the second order tensor

$(B_{ij})_{i,j=1,n}$ by the vector $d = (d_j)_{j=1,n}$. In matrix notation, we have $B \cdot d = B\,d$. Finally, $a \cdot B \cdot b$ denotes the dot product of the second order tensor $(B_{ij})_{i,j=1,n}$ with vector a on the left and vector b on the right. In matrix notation, we have

$$a \cdot B \cdot b = a^t\, B\, b = \sum_{i,j=1}^{n} a_i B_{ij} b_j.$$

The gradient of a given function $f(x)$ defined on R^n will be denoted by ∇f. In component form, this gives

$$\nabla f_i = \frac{\partial f}{\partial x_i},\ i = 1, n.$$

Similarly the divergence of a vector or tensor u will be denoted either by $\nabla \cdot u$ or by div u. In component form, we have

$$\nabla \cdot u = \operatorname{div} u = \sum_{i=1,n} \frac{\partial u_i}{\partial x_i}.$$

Finally, vectors (respectively second-order tensors) defined in the physical space R^3 will be underlined once (respectively twice). For example, $\underline{u}$ will denote the velocity of a fluid particle, and $\underline{\underline{\sigma}}$ the Cauchy stress tensor. There will be a few obvious exceptions to this rule, in particular for the tensor product $\underline{u} \otimes \underline{u}$ defined by

$$(\underline{u} \otimes \underline{u})_{ij} = u_i u_j,\ i, j = 1, 3.$$

Acknowledgements

This work has been strongly supported by INRIA (Institut National de Recherche en Informatique et Automatique) in France, which has hosted both authors for several years and produced part of the free software distributed herein.

Numerical Methods
in Sensitivity Analysis
and Shape Optimization

Chapter 1

Basic Formulations

1.1 A generic example

Reducing an optimum design problem originating from the engineering world to a well-posed mathematical problem is hard and problem-dependent. Let us consider for example the shape optimization of a missile or of an aircraft wing. The flight configurations are numerous, characterized by their Mach or Reynolds numbers, angle of attack, and many other parameters [3]. The design must behave well in all these configurations with respect to payload, total mass, and aerodynamic characteristics. Given these requirements, the introduction of a unique general framework is impossible. What is more realistic is to identify the major dimensioning flight configurations, design parameters, and practical constraints on a first reference design and then construct the local optimization problem in order to improve the performances or to relax the key constraints in these dimensioning configurations, using a small number of physically significant design parameters and a well-behaved cost function.

Let us illustrate this approach on a basic example. A key and dimensioning problem in turbojet design is to ensure the aeroelastic stability of the blades that are close to the air intake.

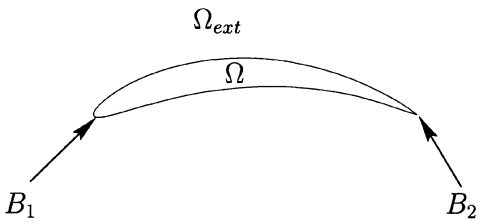


Figure 1.1. Two-dimensional section of a turbine blade

E. Laporte et al., *Numerical Methods in Sensitivity Analysis and Shape Optimization*

Design Parameters: The fundamental design parameter here is the blade's shape characterized by its contour $\partial\Omega$, with Ω denoting the domain inside the blade (Figure 1.1).

Geometric Constraints: This shape must then satisfy design constraints, such as minimal and maximal volume, locations of leading or trailing edges, excluded volumes, and so on. In a first approximation, these constraints can be expressed as a constraint on the volume, vol Ω, of the blade

$$\text{vol}\,\text{min} \leq \text{ vol } \Omega \leq \text{ vol}\,\text{max},$$

as a position constraint imposing a

$$\text{fixed trailing edge } B_2,$$

and as a localization constraint

$$\partial\Omega_{ext} \subset \Omega_{box},$$

where Ω_{box} represents the region of the space in which the blade profile is bound to stay.

Dynamic response: The dynamic response of the blade Ω is then characterized by its structural eigenmodes $(\omega_q, \underline{\xi}_q)$. These modes mathematically correspond to the self-sustained periodic solutions $\underline{\xi}(\underline{x},t) = \underline{\xi}_q(\underline{x})\cos(\omega_q t)$ of the equations of motion of the structure

$$\int_\Omega \rho\ddot{\underline{\xi}} \cdot \hat{\underline{v}} + \int_\Omega \underline{\underline{\sigma}}(\underline{\xi}) : \underline{\underline{\varepsilon}}(\hat{\underline{v}}) = 0, \forall \hat{\underline{v}} \in V_\Omega.$$

This weak form of the equation of balance of momentum

$$\rho\ddot{\underline{\xi}} - \text{div}(\underline{\underline{\sigma}}(\underline{\xi})) = 0$$

imposes that in such an eigenmode, inertia and elastic forces are in constant equilibrium. Above, $\underline{\xi}$ denotes the displacement field within the structure,

$$V_\Omega = \{\underline{v} : \Omega \to \mathbb{R}^3, \underline{v}|_{\partial\Omega_{int}} = 0, \underline{v} \in H^1(\Omega)\}$$

is the space of kinematically admissible velocity fields,

$$\underline{\underline{\varepsilon}}(\hat{\underline{v}}) = \frac{1}{2}\left[\frac{\partial\hat{\underline{v}}}{\partial\underline{x}} + \left(\frac{\partial\hat{\underline{v}}}{\partial\underline{x}}\right)^t\right]$$

is the tensor of strain rates inside the structure generated by the velocity field $\hat{\underline{v}}$, and $\underline{\underline{\sigma}}(\underline{\xi})$ denotes the Cauchy stress tensor. The vibration of a turbine blade is well described by linear elasticity, which linearly relates this stress tensor to the linearized strain tensor by

$$\underline{\underline{\sigma}}(\underline{\xi}) = \mathbf{E}\underline{\underline{\varepsilon}}(\underline{\xi})$$

with $\mathbf{E}(x)$ denoting the fourth-order elasticity tensor characterizing the local elastic response of the constitutive material.

Under this notation, making explicit the acceleration $\ddot{\underline{\xi}}_q = -\omega_q^2 \underline{\xi}_q$ and neglecting the influence of the surrounding fluid on the eigenvibrations of the blade, the equations characterizing the eigenmodes of the structure reduce to the eigenproblem

$$-\omega_q^2 \int_\Omega \rho \underline{\xi}_q \cdot \underline{\hat{v}} + \int_\Omega \mathbf{E} \underline{\underline{\varepsilon}}(\underline{\xi}_q) : \underline{\underline{\varepsilon}}(\underline{\hat{v}}) = 0, \forall \underline{\hat{v}} \in V_\Omega, \tag{1.1}$$

$$\int_\Omega \rho |\underline{\xi}_q|^2 = 1, \tag{1.2}$$

in which the unknowns are the pulsation $\omega_q \in \mathbb{R}$ and the modal deformation $\underline{\xi}_q \in V_\Omega$.

Aerodynamic analysis: The aerodynamic characteristics of a typical blade are obtained by computing the time periodic solution $W_q(\underline{x}, t)$ [of period $\frac{2\pi}{\omega_q}$] of the equations of motion governing the external fluid under the action of a given air inflow and of an imposed eigenvibration of the blade. The characteristics of the air inflow and the choice of the blade's vibration mode are specified by the designer and correspond to a critical working configuration. When we can neglect all viscous effects, the fluid is governed by the Euler's equations whose conservative form is [43]

$$\frac{\partial W_q}{\partial t} + \text{ div } F(W_q) = 0 \quad \text{on} \quad \Omega_{ext}, \tag{1.3}$$

$$W_q = W_\infty \quad \text{on} \quad \partial\Omega_\infty, \tag{1.4}$$

$$\underline{u} \cdot \underline{n} = \underline{\xi}_q(\underline{x}) \cdot \underline{n} \cos(\omega_q t) \quad \text{on} \quad \partial\Omega_{ext}, \tag{1.5}$$

$$W_q(T_q) = W_q(0) \quad \text{on} \quad \Omega_{ext}, \tag{1.6}$$

where the unknown vector W_q denotes the vector $W_q = \begin{bmatrix} \rho \\ \rho \underline{u} \\ E \end{bmatrix}$ of density, momentum and total energy, and where the different mass, momentum and energy fluxes are given by the well-known expression

$$F(W) = \begin{bmatrix} \rho \underline{u} \\ \rho \underline{u} \otimes \underline{u} + p \underline{\underline{Id}} \\ (E+p)\underline{u} \end{bmatrix},$$

the pressure p being defined by the state law $p = \dfrac{E - \frac{1}{2}\rho u^2}{\gamma - 1}$. Because of space periodicity around the turbine axis, the computational domain on which these fluid equations are solved can be reduced to a simple slab surrounding a single blade, with absorbing boundary conditions usually imposed at the outflow boundary.

Aeroelastic Stability: The aeroelastic stability of the blade system is guaranteed when the fluid pumps energy from the structure during the above periodic motion. Improving the aeroelastic stability can therefore be realized by finding the shape which minimizes the average energy $\int p\underline{n} \cdot \dot{\underline{\xi}}$ given during one time period $\frac{2\pi}{\omega_q}$ by the fluid to the structure when the structure vibrates along one of its prefered low frequency eigenmodes . Here $\underline{n}$ denotes the normal unit vector to the fluid solid interface $\partial\Omega_{ext}$ pointing towards the blade. If we select for example the first mode (corresponding to the smallest pulsation ω_1 and velocity field $\dot{\underline{\xi}} = -\omega_1 \sin(\omega_1 t)\underline{\xi}_1$), this aeroelastic stability problem reduces to the minimization problem

$$\min_{\partial\Omega} \frac{\omega_1}{2\pi} \int_0^{2\pi/\omega_1} \int_{\partial\Omega_{ext}} p\underline{n} \cdot \left[-\omega_1 \sin(\omega_1 t)\underline{\xi}_1\right] da\, dt,$$

where the unknown is the shape $\partial\Omega$. In this problem, the pulsation ω_1 and pressure p appearing in the cost function are obtained from the shape by solving respectively a structural eigenproblem (1.1)–(1.2) and an external flow problem (1.3)–(1.6).

1.2 Abstract formulation of a shape optimization problem

1.2.1 General framework and notation

In a general setting, a shape optimization problem as described above can be viewed as the constrained minimization of a cost function whose evaluation involves the solutions of partial differential equations set on a parameter dependent geometrical domain Ω.

The basic problem is to find an optimal shape Ω^* (the shape of the blade in the above example) in a set Ω_{adm} of admissible shapes, and satisfying nonlinear design constraints of the type $S(\Omega) \leq \varepsilon_0$, with given ε_0. This optimal shape minimizes an objective (cost) functional

$$j(\Omega^*) = \min j(\Omega) = \min J(\Omega, W_\Omega),$$

depending both on the selected shape and on the solution W_Ω of partial differential equations $f(\Omega, W) = 0$, characterizing the state of the system under study. In the above example, the solution $W_\Omega = \{(\omega_1, \underline{\xi}_1), W_1\}$ corresponds to the structural eigenmode of the structure and to the solution of the fluid equation, and the partial differential equations f denote the structural eigensystem (1.1)–(1.2) and fluid's equations (1.3)–(1.6). The cost functional is thus defined and computed through the flowchart

$$\Omega \xrightarrow{f} W_\Omega \rightarrow j(\Omega) = J(\Omega, W_\Omega).$$

In this general setting, a shape optimization problem takes the final abstract form

$$(P)\begin{cases}\text{Find the shape } \Omega^* \text{ such that}\\ \quad j(\Omega^*) = \min_{\Omega} j(\Omega) = \min_{\Omega} J(\Omega, W_\Omega),\\ \text{under the constraint } S(\Omega) \leq \varepsilon_0,\\ \text{and with the state } W_\Omega \text{ solution of the equation}\\ \quad f(\Omega, W_\Omega) = 0.\end{cases}$$

Above,

- Ω denotes the unknown shape and is a subset of $\mathbb{R}^3$: Ω is defined in general by a finite set of parameters $z \in \mathbb{R}^n$, called *the control variables,*
- W_Ω is *the state variable.* It characterizes the state of the system under study. In general, W_Ω is the unique solution of a set of partial differential equations, the so-called *state equations* written on Ω or around Ω,
- $j(\Omega) = j(\Omega, W)$ is *the objective or cost function* which must be minimized. Its expression is highly problem dependent, and is a function of both the domain Ω and the state variable W. Its evaluation therefore first requires the solution of the state problem $f(\Omega, W_\Omega) = 0$,
- $S(\Omega)$ defines the set of design constraints imposed on the shape Ω.

Under this abstract form, Problem (P) is ill posed. The space in which to look for the design shapes Ω is not defined, and therefore has no topological structure. Moreover, the influence of Ω on the state equation $f = 0$ or on the design constraints S is not explicit. In many cases, it is possible to give a precise mathematical structure to Problem (P). But, our goal here is not to obtain the absolute optimum, but to get a better, efficient and robust design Ω, or at least to understand the effects of a local change of design. This can be achieved with less difficulty by introducing an adequate discretization of problem (P).

1.2.2 Discretization

The reduction of the shape optimization Problem (P) to a standard problem of mathematical programming can be achieved in a rather flexible and efficient way by describing the computational domain Ω through its computational grid, whose evolution can be easily governed by a finite number of control parameters $z \in \mathbb{R}^n$. The choice of these control parameters may be problem and solver dependent, but once made, the optimization problem can be fully described through the numerical solvers to be used during its solution. The full description now consists of

1. a reference configuration $\Omega^o \in \Omega_{adm}$ of the body under study and the corresponding contour γ^0;
2. a map $\gamma\left(z, \gamma^0\right)$ defining the shape γ of the contour from its initial shape and from the normal motion of carefully chosen control points controlled by the vector z (cf. for example § 1.4.3);

3. a computational grid $\{\underline{X}(z) = (x_i)_{i=1,ns}, NU = (NU_{i,l})_{i=1,ndloc,l=1,nt}\}$ defined on the configuration Ω delimited by γ describing the position $\underline{X} \in \mathbb{R}^3$ of the ns grid nodes, and giving for each cell $l = 1$ to nt the set $NU_{.,l}$ of its $ndloc$ vertices. In this construction, the shape of the object is entirely characterized by its computational grid, and the parameters z controlling the shape are in fact only used to control the grid deformation;

4. the discretization scheme

$$f_h(\underline{X}, W_h) = 0 \quad \text{in} \quad \mathbb{R}^p$$

used for approximating the state equation $f = 0$ on the computational grid $\underline{X}$, and for defining the approximate discrete solution $W_h \in \mathbb{R}^p$ of the state equation. For example, after finite element discretization, the structural eigenproblem of our aeroelastic example takes the form [27, 52]

$$\text{Find } \omega, \underline{\xi} = \sum_{i=1}^{3\times ns} U^i \underline{\varphi}_i \text{ such that}$$

$$K(\underline{X}) \cdot U = \omega^2 M(\underline{X}) \cdot U,$$

$$< U, M(\underline{X}) \cdot U >= 1,$$

where the stiffness matrix K and mass matrix M are respectively given by

$$K(\underline{X})_{ij} = \int_\Omega \mathbf{E}\underline{\underline{\varepsilon}}(\underline{\varphi}_i) : \underline{\underline{\varepsilon}}(\underline{\varphi}_j),$$
$$M(\underline{X})_{ij} = \int_\Omega \rho \underline{\varphi}_i \cdot \underline{\varphi}_j.$$

Above, $\underline{\varphi}_{3(l-1)+q}$ denotes the piecewise polynomial shape function taking the value 1 in the direction $\underline{e}_q$ at node l and with zero values at all other degrees of freedom.

With this description, the design constraints $S(\Omega) \leq \varepsilon_0$ can be also expressed as functions of the present position of the grid points $\underline{X}$ and can be reduced to the generic form

$$g_{h_i}(z, \underline{X}) \leq 0, \forall 1 \leq i \leq m.$$

Altogether, the shape optimization problem is then reduced to the approximate problem

$$\boxed{(P_h) \quad \min_{z \in \mathbb{R}^n,\, g_h(z,\underline{X}) \leq 0} j_h(z) = J(\underline{X}, W_h(z))}$$

where $W_h(z)$ denotes the solution of the discrete state equation

$$(E_h) \begin{cases} f_h(\underline{X}, W_h) = 0 \quad \text{in} \quad \mathbb{R}^p, \\ \underline{X} = \underline{X}(z). \end{cases}$$

This new formulation is convenient for two main reasons:

- it has the form of a classical finite dimensional constrained optimization problem and therefore can be approached by standard techniques of mathematical programming, to be introduced in the forthcoming chapters,

- all functions appearing in this formulation can be explicitly calculated on a computer.

On the other hand, this formulation ignores the natural topology of the problem under study and is dependent on the quality of the discretization strategy which is used. This drawback can be overcome by using adaptive discretization techniques, as done in several recent research developments [31] and explained later.

1.3 Sensitivity analysis

Optimal shape design looks for a local minimum of the approximate cost function $j_h(z) = J(\underline{X}(z), W_h(z))$ among all local admissible designs. This simple goal might not be fully relevant in engineering practice. On one hand, the exact local minimization of j_h may lead to a very small decrease of the cost function in comparison to the added complexity induced by the actual implementation of the corresponding optimal design. On the other hand, the mathematical formulation of the local optimization problem may omit untold practical constraints, which will be known to the designer but not taken into account by the computer. The local optimization problem can also be a part of a complex multipoint design methodology involving several conflicting optimization processes.

Sensitivity analysis, as described in [29], is a common piece of the answers for all these situations. The idea is simply to estimate the potential gain or loss of the cost function induced by a local increase of the design parameter z_i, or more generally to get insight on the relevance of a given change of parameters with respect to the global design problem. This is achieved in practice simply by computing the directional derivatives $\nabla f(z) \cdot dz \approx \frac{1}{\epsilon}(f(z + \epsilon dz) - f(z))$ of the cost function $f(z) = j_h(z)$, of the design constraints $f(z) = g_h(z, \underline{X}(z))$, or of the combined Lagrangian $f(z) = \mathcal{L}(z, \mu) = j_h(z) + \langle \mu, g_h(z, \underline{X}(z)) \rangle$, along the different directions dz of the space of design parameters z.

In mathematical terms, sensitivity analysis around a given admissible configuration z reduces thus to a calculation of gradients, and corresponds to the sequence of operations

1. characterize the state of the system by computing the approximate solution $\underline{X}(z)$ and $W_h(z)$ of the discrete state equation (E_h);

2. compute the values $j_h(z) = J(\underline{X}(z), W_h(z))$ and $g_h(z, \underline{X}(z))$ of the cost function j_h and of the design constraints g_h for the current configuration z;

3. using the techniques to be described in Chapters 5 and 6, compute the

derivatives $\frac{d}{dz}j_h(z)$ and $\frac{d}{dz}g_h(z,\underline{X}(z))$ of this cost function and constraints;

4. if needed, compute the value of the Lagrange multiplier associated to the design constraints, and the derivative of the associated Lagrangian.

1.4 Shape parametrization

Whether considering optimization problem or sensitivity analysis, the first generic difficulty which must be overcome consists in the introduction of a map $\underline{X}(z)$ constructing a grid as a function of the control parameters z. This construction is described in the next section, together with different geometrical tools useful for generating or describing general shapes. The discussion will be reduced to two-dimensional configurations. The same principles apply to three-dimensional problems, but the technical details become more involved.

1.4.1 Cubic spline interpolation

One of the simplest ways of defining regular two-dimensional contours is to use cubic spline interpolation [8]. Given n points $t_i \in \mathbb{R}$, the values $f(t_i)$ of the interpolated function at these points, and of its derivative at the end points, this interpolation constructs a piecewise cubic function with continuous second derivatives, respecting the imposed values $f(t_i)$, $f'(t_1)$ and $f'(t_n)$. For example, Figure 1.2 represents the cubic spline f defined by:

$$\begin{cases} f(-1) = 1, \\ f(-0.5) = -1, \\ f(0) = 1, \\ f'(-1) = -1, \\ f'(0) = 1. \end{cases}$$

To construct in practice the spline interpolation f, one first defines for $i = 1, \ldots, n-1$ and $j = 1, \ldots, 4$ the function $f_{j,i}$ by

$$f_{j,i}(t) = \begin{cases} \frac{1}{j!}(t - t_i)^{j-1} \text{ if } t \in [t_i, t_{i+1}[, \\ 0 \text{ if not.} \end{cases}$$

The spline interpolation f can then be written as [8]

$$f(t) = \sum_{i=1}^{n-1} \sum_{j=1}^{4} c_{j,i} f_{j,i}(t).$$

Above, the $4(n-1)$ unknown coefficients $c_{j,i}$ are specified by the $n+2$ imposed

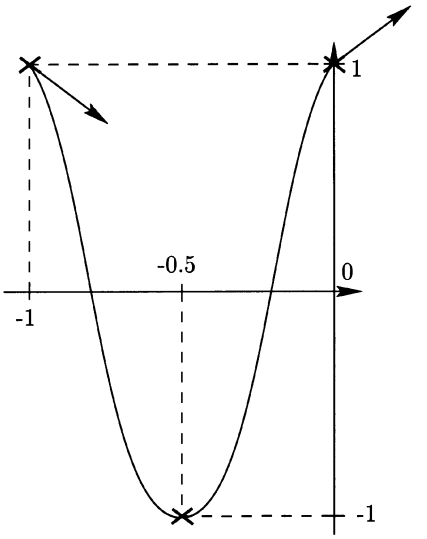


Figure 1.2. Data used for constructing a cubic spline

values

$$\sum_{i=1}^{n-1}\sum_{j=1}^{4} c_{j,i} f_{j,i}(t_k) = f(t_k), k = 1, n,$$

$$\sum_{i=1}^{n-1}\sum_{j=1}^{4} c_{j,i} f'_{j,i}(t_1) = f'(t_1),$$

$$\sum_{i=1}^{n-1}\sum_{j=1}^{4} c_{j,i} f'_{j,i}(t_n) = f'(t_n),$$

and by the $3(n-2)$ conditions imposing the continuity of f, of its first derivative f' and of its second derivative f'' at the intermediate points $t_k, k = 2, \ldots, n-1$. The solution $(c_{i,j})_{i,j}$ of this linear tridiagonal system of $4(n-1)$ equations is obtained by a direct Gaussian elimination as done in the subroutine `cubspl` described in Appendix A. Its input data are

- `n`, number of points t_i;
- `tau(1:n)`, real array of interpolation points $t_1, \ldots, t_n$, with $t_1 < t_2 < \ldots < t_n$;

- `c(4,n)`, real two-dimensional array of dimension $4 \times n$ containing in input the interpolating data:
  - $\mathtt{c}(1,i) = f(t_i)\,, \quad i = 1, \ldots, n;$
  - $\mathtt{c}(2,1) = f'(t_1)\,, \quad \mathtt{c}(2,n) = f'(t_n).$

The output solution coefficients are stored in the array `c(1:4,1:n-1)`, from which we define the interpolating function for $t \in [t_i, t_{i+1}]$ by

$$\mathtt{c}(t) = \mathtt{c}(1,i) + h \times (\mathtt{c}(2,i) + \frac{h}{2} \times (\mathtt{c}(3,i) + \frac{h}{3} \times \mathtt{c}(4,i)))$$

under the notation

$$h = t - t_i.$$

1.4.2 Wing profile parametrization by cubic splines

Cubic splines lead then to simple parametrizations of wing or blade profiles such as the one represented in Figure 1.3.

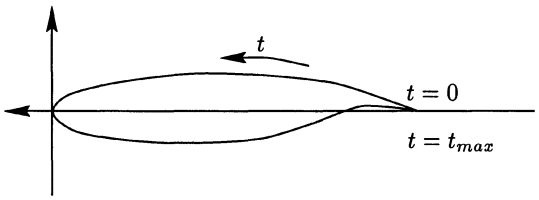


Figure 1.3. Generic wing profile

This is done by introducing a list of n points $(x(t_i), y(t_i))_{i=1,\ldots,n}$ and of two angles characterizing the profile at the trailing edge, with $x(t_1) = x(t_n)$ and $y(t_1) = y(t_n)$.

Cubic spline interpolation, operating separately on the coordinates $x(t)$ and $y(t)$, defines then a piecewise cubic regular two-dimensional closed curve $(x(t), y(t))_{t \in [0, t_{max}]}$ defined on the interval $(0, t_{max})$ completely characterized by its input data, namely the positions of the control points $(x(t_i), y(t_i))_{i=1,\ldots,n}$ and the values of the angles α and β (Figure 1.4). The values of the curve derivatives at end points are obtained from these angles through the relation

$$\frac{1}{\sqrt{\left(\frac{dx}{dt}(0)\right)^2 + \left(\frac{dy}{dt}(0)\right)^2}} \frac{dx}{dt}(0) = -\cos(\alpha)\,,$$

$$\frac{1}{\sqrt{\left(\frac{dx}{dt}(t_{max})\right)^2 + \left(\frac{dy}{dt}(t_{max})\right)^2}} \frac{dx}{dt}(t_{max}) = \cos(\beta)\,,$$

$$\frac{1}{\sqrt{\left(\frac{dx}{dt}(0)\right)^2+\left(\frac{dy}{dt}(0)\right)^2}}\frac{dy}{dt}(0)=\sin(\alpha),$$

$$\frac{1}{\sqrt{\left(\frac{dx}{dt}(t_{max})\right)^2+\left(\frac{dy}{dt}(t_{max})\right)^2}}\frac{dy}{dt}(t_{max})=-\sin(\beta).$$

Assuming that t is the curvilinear absissae, these formulas become

$$\frac{dx}{dt}(0)=-\cos(\alpha),\quad \frac{dx}{dt}(t_{max})=\cos(\beta),$$

$$\frac{dy}{dt}(0)=\sin(\alpha),\quad \frac{dy}{dt}(t_{max})=-\sin(\beta).$$

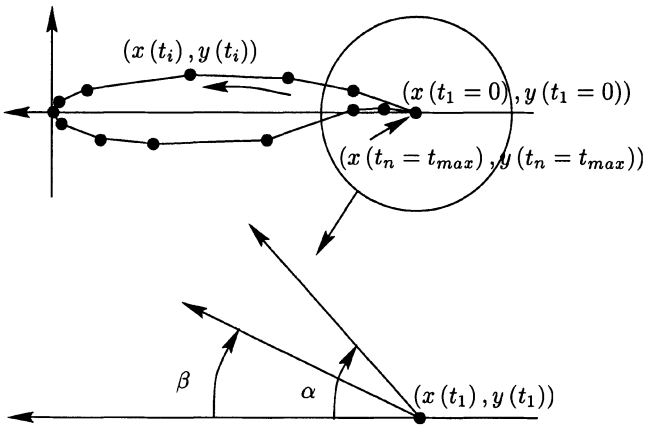


Figure 1.4. Wing profile parametrized by cubic splines

1.4.3 Parametrization of a shape deformation

We now suppose that we are given a two-dimensional reference shape characterized by the data of the inital position of q consecutive boundary vertices $M_1, M_2, \ldots, M_q$. These points are either the control points used in the spline construction described in the above paragraph, or are the boundary vertices of a computational grid obtained by a mesh generator such as described in the next paragraph. At each vertex M_i, the user knows or is able to construct a vector $\underline{n}_i$ approximatively perpendicular to the contour at this point and an approximate curvilinar abscissa s_i of this point along the contour.

The deformation of this shape or of the associated computational grid can now be characterized by the displacement of each vertex M_i along the normal

$\underline{n}_i$. This normal motion is then parametrized by a cubic spline $\psi(t)$ defined on n user-defined abscissae $s_1 = t_1 < t_2 < \cdots < t_n = s_q$, using $n-2$ free intermediate values $\psi(t_i) = z_i, i = 2, n-1$, two frozen end values $\psi(t_1) = \psi(t_n) = 0$ (the end points are fixed) and two free values for the end derivatives $\psi'(t_1) = z_1$, $\psi'(t_n) = z_n$ (free rotation of the contour around its end points). This defines a cubic spline

$$\psi(t) = \sum_{i=1}^{n-1} \sum_{j=1}^{4} c_{j,i}(z) f_{j,i}(t)$$

whose coefficients $c_{j,i}$ are computed in the subroutine `cubspl` and linearly depend on the above n real parameters $(z_i)_{i=1,n} = \psi'(t_1), \psi(t_i), i = 2, n-1, \psi'(t_n)$. The motion $\underline{\xi}_l$ of each boundary vertex M_l is then given by

$$\underline{\xi}_l = \psi(s_l)\underline{n}_l = \left(\sum_{i=1}^{n-1} \sum_{j=1}^{4} c_{j,i}(z) f_{j,i}(s_l) \right) \underline{n}_l. \tag{1.7}$$

Similarly, the motion of any intermediate point of abscissa t on the reference contour can be obtained by the formula $\underline{\xi}(t) = \psi(t)\underline{n}(t)$ (Figure 1.5).

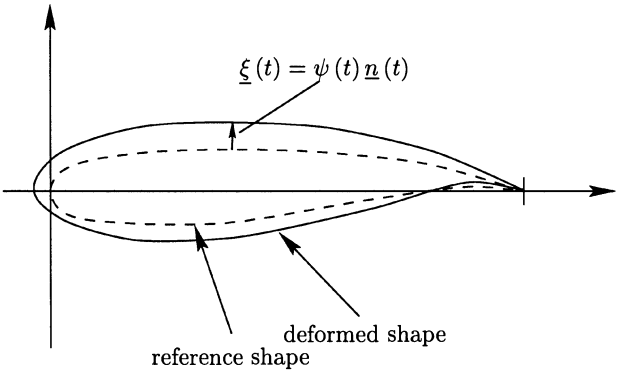


Figure 1.5. Motion of the boundary

1.5 Mesh construction and deformation

We now proceed to the construction and deformation of a computational grid $\underline{X}$ inside the domain Ω once the contour γ or the contour deformation $\underline{\xi}$ has been constructed as described in the previous section. This operation requires two distinct steps which are the construction of an initial mesh, and the deformation of an existing mesh.

1.5.1 Mesh construction

The *amdba* data structure

This format is one of the simplest available data structures for the description of a two or three-dimensional unstructured computational grid. In this format, a two-dimensional mesh is stored as an ASCII file containing the following data:

- ns, nt (total number of vertices, total number of elements (cells))
- 1, coor(1,1), coor(2,1), logfr(1)
- ...
- is, coor(1,is), coor(2,is), logfr(is) (vertex number, followed by its coordinates and its reference number)
- ...
- ns, coor(1,ns), coor(2,ns), logfr(ns)
- 1, nu(1,1), nu(2,1), nu(3,1), nd(1)
- ...
- it, nu(1,it), nu(2,it), nu(3,it), nd(it) (element number, followed by the number of each of its three vertices, and by the element reference number)
- ...
- nt, nu(1,nt), nu(2,nt), nu(3,nt), nd(nt)

The reference numbers for nodes and elements do not contribute to the geometric construction of the mesh but characterize the type of boundary conditions or physical domain characteristics relevant for the considered node or element.

The three steps for mesh construction

Constructing a mesh can be decomposed into three main steps as illustrated below in the construction of a mesh of a square torus (Figure 1.6). This mesh will be constructed by using the two-dimensional mesh generator and editor EMC2 (http://www-rocq.inria.fr/gamma/cdrom/www/emc2/eng.htm, also available in the enclosed CD-ROM), and its graphical environment.

1. Geometric construction: first, one must define the geometry of the object to be meshed. For a square torus, this is done by defining eight oriented segments. The segments belonging to the same boundary must be oriented in the same direction. The external and internal boundaries must have opposite orientation directions (Figure 1.7).

2. Mesh preparation: the next step defines the discretization itself. In particular, one must specify (See Figure 1.8):

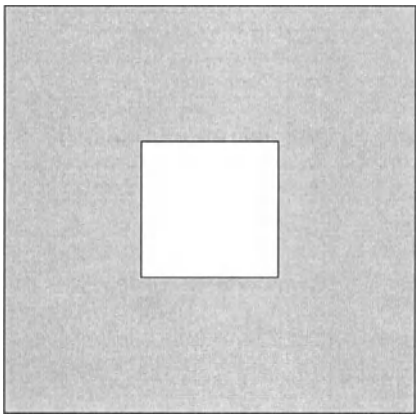


Figure 1.6. The square torus

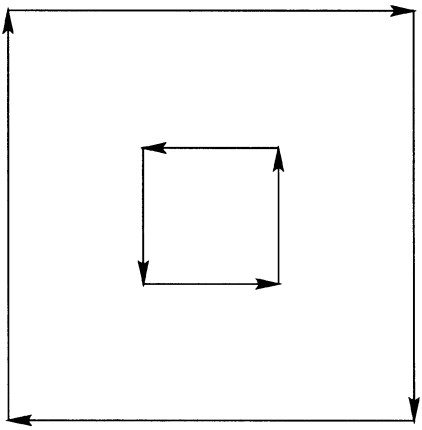


Figure 1.7. Geometric definition in the EMC2 mesh generator and graphical editor

- on each segment, the number of nodes to be generated on the segment and the reference number of the nodes belonging to the segment,
- and the interior of the domain to be meshed.

3. Mesh construction: the last step fills the interior domains with a Delaunay type mesh generator, and is monitored by a few control parameters (mesh refinement, regularization factor, and so on) (see Figure 1.9).

The realization of all these steps inside the automatic mesh generator `EMC2` is further illustrated in Figures 1.16, 1.17, 1.18 at the end of the chapter, describ-

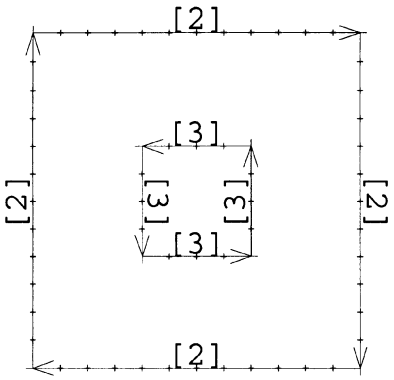


Figure 1.8. Mesh preparation within EMC2 mesh generator and graphical editor

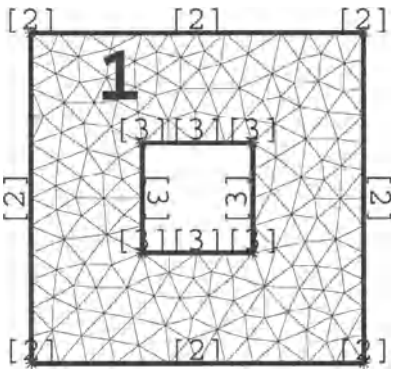


Figure 1.9. Final mesh obtained by the EMC2 mesh generator and graphical editor

ing what can be observed on the computer screen during the different steps.

1.5.2 A simple mesh deformation algorithm

We now describe a very simple minded algorithm for deforming an existing mesh $\{\underline{X}^o(i:1,ns), NU^o(i:1,ndloc; l=1,nt)\}$ by imposing a displacement of its contour as given in equation (1.7). The objective is to propagate this imposed boundary displacement inside the mesh by computing new admissible node positions $\underline{X} = \underline{X}(z)$ inside the domain once the boundary vertices have been moved to the position $(\underline{X}_l)_{l=1,q} = (\underline{X}_l^0 + \underline{\xi}(z, s_l))_{l=1,q}$ as indicated in the previous section. The proposed algorithm iteratively proceeds in two steps:

1. The first step computes on each element an element based displacement

by averaging the displacement of its nodes: for a triangle, this value is equal to one third of the sum of the displacement of its three vertices (see Figure 1.10).

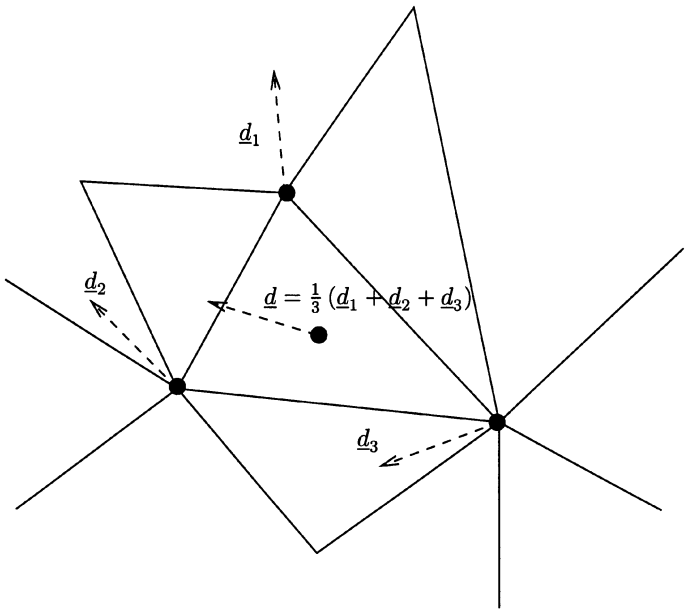


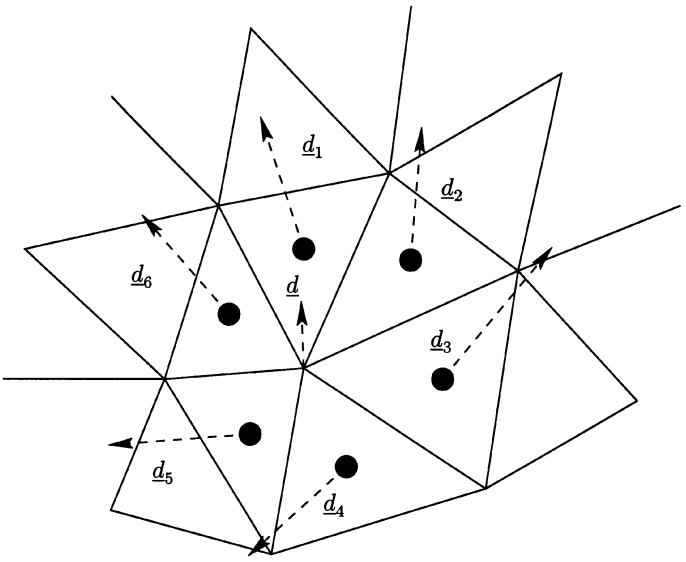
Figure 1.10. Averaging nodal displacements on a given triangle

2. The second step computes on each internal node an average displacement by summing all element displacements of the elements containing the node, and by dividing the result by the number of such elements (see Figure 1.11).

A few sweeps of this algorithm (typically a dozen iterations) constructs a reasonably smooth mesh from the imposed boundary displacement (see Figure 1.12). The cost of a mesh deformation with this algorithm is of the order of the chosen number of sweeps times the sum of the number of vertices with the number of elements.

1.5.3 A sophisticated mesh deformation algorithm

When the boundary displacement is too large or too distorted, the simple averaging strategy might fail. In such cases, one must use more sophisticated mesh deformation algorithms. The method presented below uses approximate Green's functions and is detailed in [33]. It gives an explicit expression of each nodal displacement based on the distance between the considered node and the boundary.



$$\underline{d} = \tfrac{1}{6}\left(\underline{d}_1 + \underline{d}_2 + \underline{d}_3 + \underline{d}_4 + \underline{d}_5 + \underline{d}_6\right)$$

Figure 1.11. Averaging element displacements to reconstruct nodal displacements

More precisely, if we denote by $I_{\partial\Omega}$ the set of indices corresponding to the boundary nodes, the proposed sophisticated method prescribes the displacement $\underline{\xi}(\underline{x}_i)$ of any internal node i to the following weighted average displacements $\underline{\xi}(\underline{x}_k)$ of the boundary nodes:

$$\underline{\xi}\left(\underline{x}_i\right) = \frac{1}{\alpha_i} \sum_{k \in I_{\partial\Omega}} w_k \alpha_{ki} \underline{\xi}\left(\underline{x}_k\right). \tag{1.8}$$

Here w_k is a scalar weight associated to each boundary node k,

$$\alpha_{ki} = \frac{1}{\left|\underline{x}_k - \underline{x}_i\right|^{\beta}}$$

is the local value of the discrete Green's function, with $\beta \geq 2$ an arbitrary parameter, and α_i is a normalization factor given by

$$\alpha_i = \sum_{k \in I_{\partial\hat{\Omega}}} w_k \alpha_{ki}.$$

In general, the coefficients w_k correspond to the area of the geometric boundary cell centered in $\underline{x}_k$. In two dimensions, this means that w_k is equal to the average

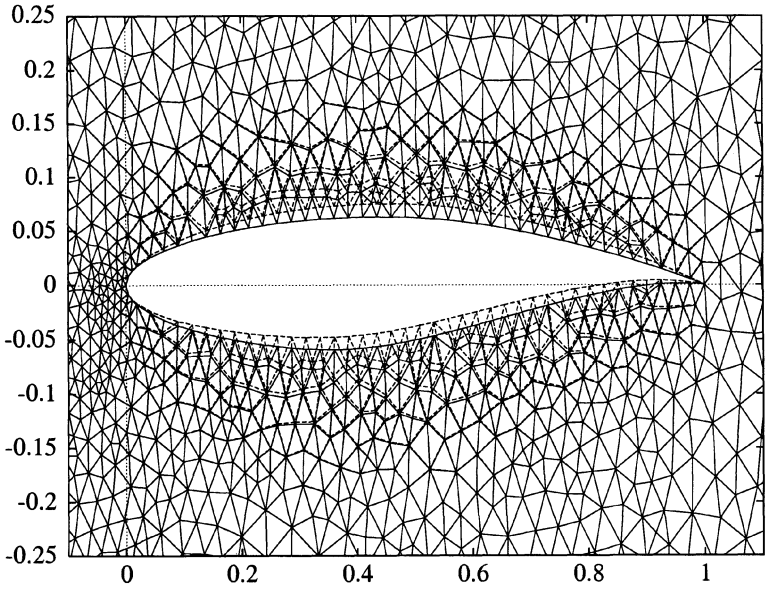


Figure 1.12. A typical deformed mesh obtained by averaging

length of the boundary edges containing the node k. With this choice, formula (1.8) is an approximation of the convolution integral

$$\frac{1}{\int_\gamma \frac{d\gamma}{|x-u|^\beta}} \int_\gamma \frac{\underline{\xi}(u)}{|x-u|^\beta}\, d\gamma, \tag{1.9}$$

with $\gamma = \partial\Omega$. More precisely, $\int_\gamma \frac{\underline{\xi}(u)}{|x-u|^\beta}\, d\gamma$ is the result of the convolution of the Green's function $\frac{1}{|x|^\beta}$ with the distribution $\underline{\xi}$ located on γ. The mesh deformation algorithm based on Formula (1.8) has therefore a clear theoretical interpretation. It is also robust. Unfortunately, it is rather expensive to compute. The cost of a mesh deformation with this algorithm is of the order of

number of internal nodes × number of boundary nodes.

A bidimensional example

Let us consider the case of a refined unstructured mesh around an NACA 65-010 profile. This mesh contains 20850 nodes and is partially described in Figure 1.13 [1]. Such a mesh is very hard to deform because of its fine grain and its large aspect ratio next to the body. It can nevertheless be successfully deformed by algorithm (1.8) with $\beta = 4$. The boundary deformation corresponds here to a

[1] This mesh was realized by Mugurel Stanciu at INRIA Rocquencourt.

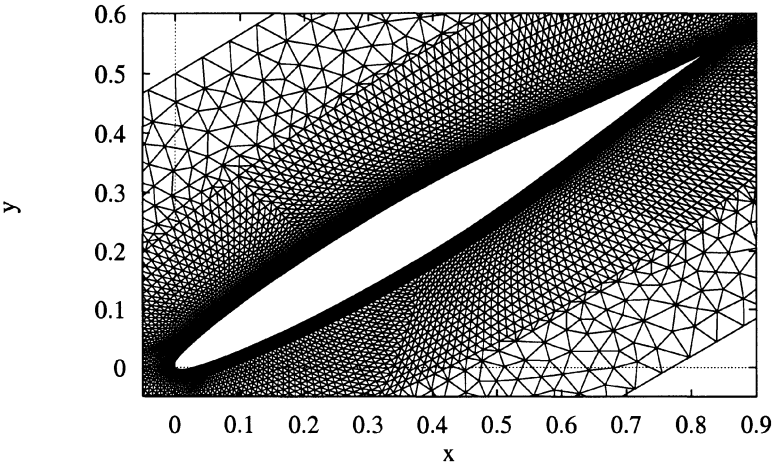


Figure 1.13. Detail of the refined unstructured mesh of the NACA 65-010 next to the boundary

factor 2 dilatation of the profile around its chord and is given by the formula

$$\underline{\xi}\left(x^1, x^2\right) = \begin{pmatrix} -0.544639\left(0.83867x^2 - 0.544639x^1\right) \\ 0.83867\left(0.83867x^2 - 0.544639x^1\right) \end{pmatrix}.$$

Figures 1.14, 1.15 are magnifications of the mesh close to the leading edge before and after deformation.

1.6 Exercises

The exercises below should help the reader to become familiar with basic two-dimensional mesh generation and deformation procedures. Corresponding data and solutions can be found in the enclosed CD-ROM.

1.6.1 Mesh generation around a wing profile

RAE

Build with EMC2 a mesh around an RAE wing profile. The profile is defined by 86 boundary vertices whose position is stored in the ASCII file RAE.pts. In this construction, you will set the nodal reference number to be 3 on the profile and 5 on the farfield boundary.

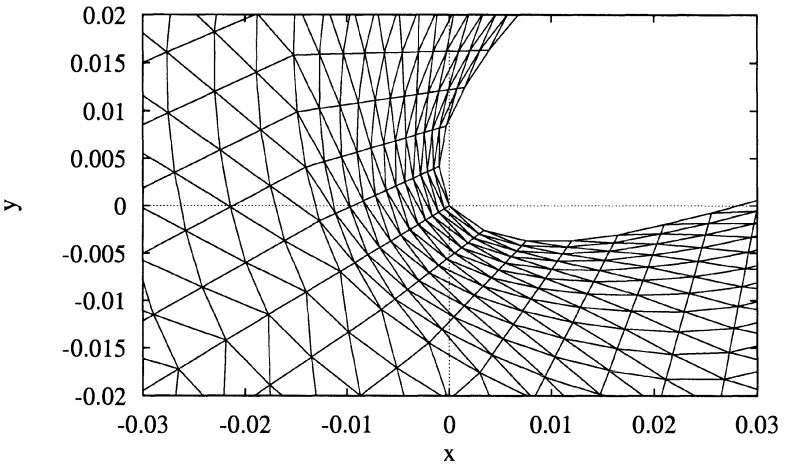


Figure 1.14. Mesh close to the leading edge before deformation

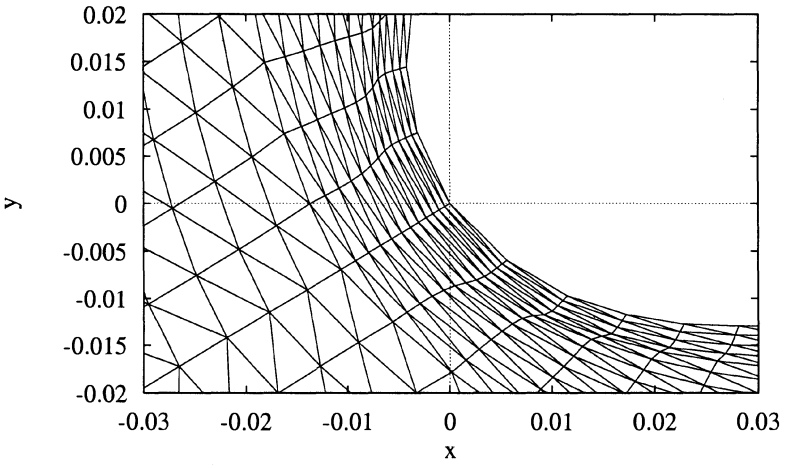


Figure 1.15. Mesh close to the leading edge after deformation

NACA

Build a mesh around a NACA wing profile using 40 vertices on the profile and the same nodal reference numbers as for the RAE profile.This construction should be split in the following subtasks:

- define the half profile by a spline interpolation based on the interpolating points given in the file NACA.pts;
- build on the half mesh around this spline;
- obtain the full mesh by symmetrization.

1.6.2 A software package for mesh deformation

Write a computer programme which reads a given mesh on the file `MESH_IN` using the data structure *amdba*, which moves the nodal points by averaging a given imposed displacement of the boundary nodes, and which stores the result in the file `MESH_OUT` using the same *amdba* format.

1.6.3 Cubic spline shape parametrization

1. Using the subroutine `cubspl`, write a computer programme which generates a large number of points (x,f(x)) on the graph of the cubic spline f defined by:

$$\begin{cases} f(-1) = 1 \\ f(-.5) = -1 \\ f(0) = 1 \end{cases} \quad \text{and} \quad \begin{cases} f'(-1) = -1 \\ f'(0) = 1 \end{cases} .$$

2. Using cubic spline interpolation, build a wing profile from an input data file `INPUT.rae` containing

   - the number of points `n`;
   - two angles `alpha` and `beta`;
   - the coordinates of the `n` points.

   For this purpose, write a programme which reads this input data file, builds the corresponding profile by cubic spline interpolation (using the subroutine `cubspl`) and generates the coordinates of a series of vertices belonging to the profile, coordinates which will be stored in the output file `OUTPUT`.

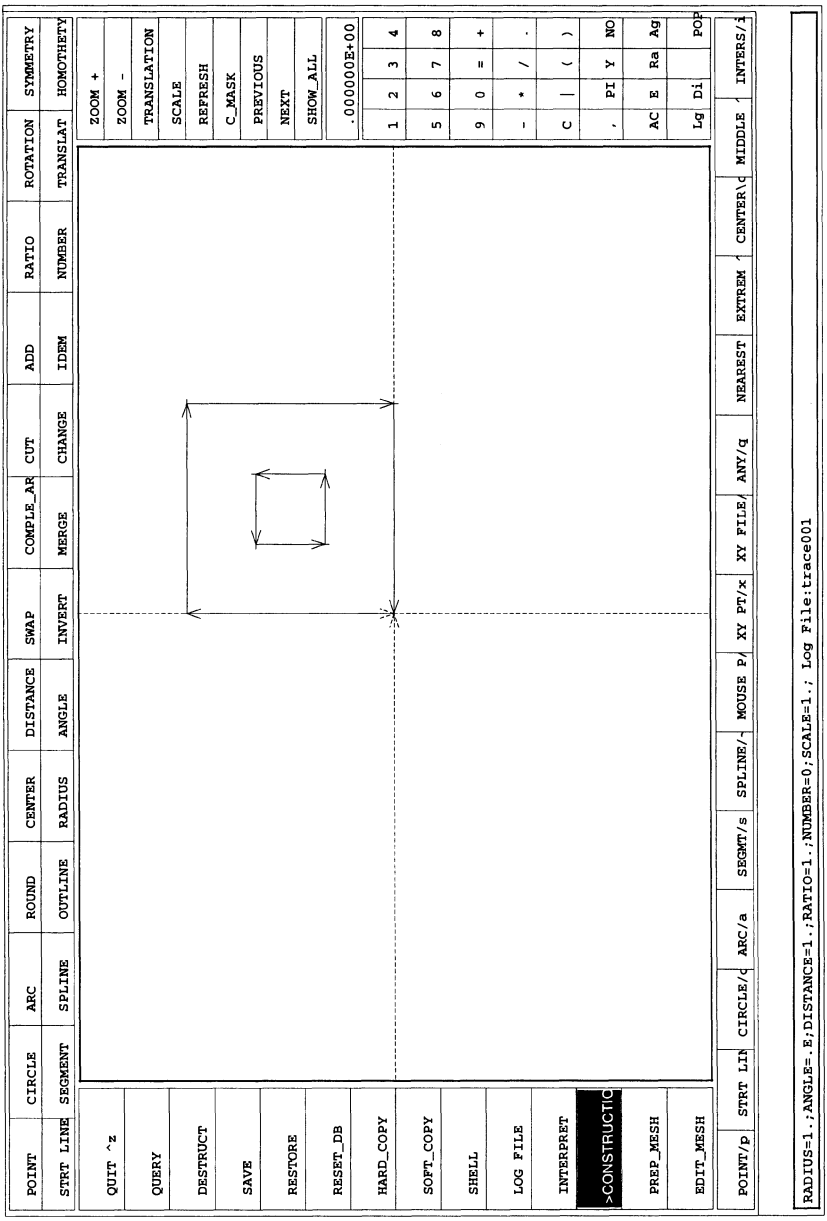


Figure 1.16. Geometric description of the mesh

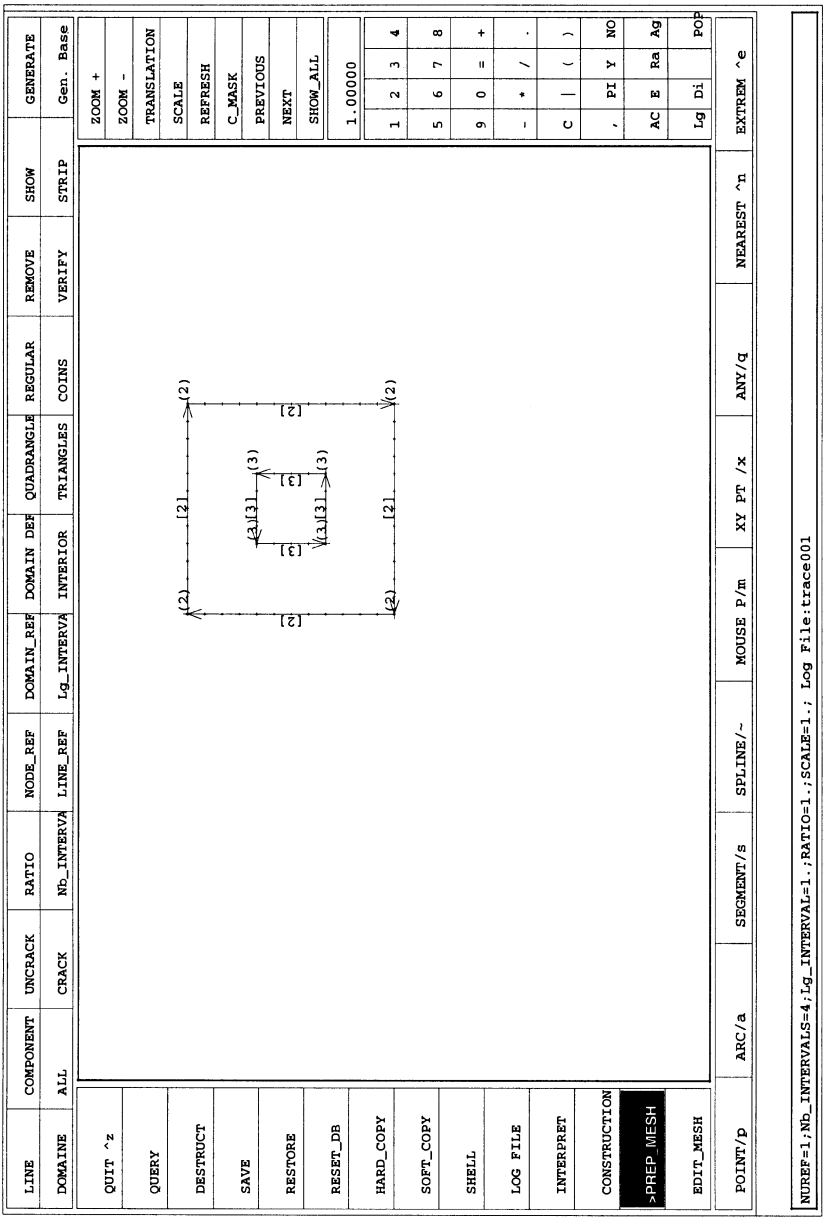


Figure 1.17. Mesh preparation

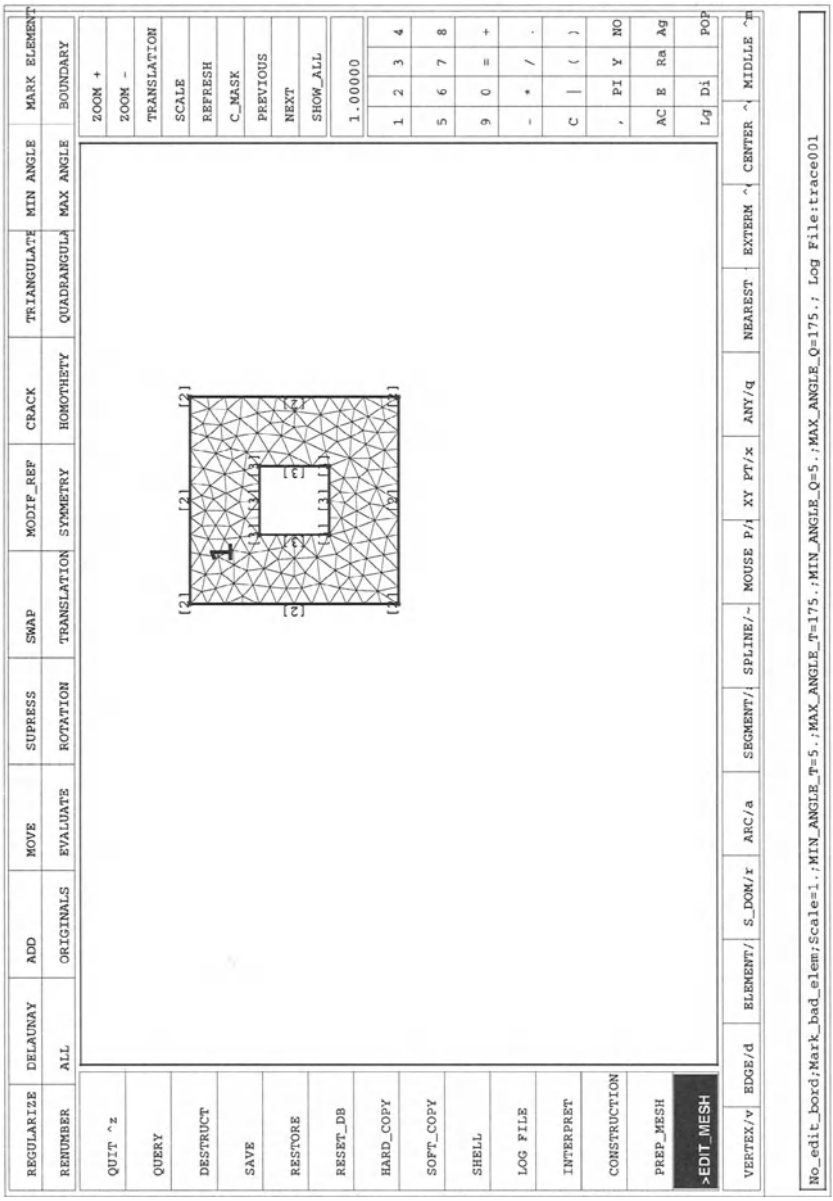


Figure 1.18. Final mesh

Chapter 2

Finite Dimensional Optimization

2.1 Basic problem and notation

In abstract form, shape optimization as introduced in the previous chapter consists in finding a global minimizer z^* of the cost function $j(z)$ on the set A of admissible designs:

$$\boxed{\begin{array}{c}\text{Find } z^* \in A \text{ such that} \\ j(z^*) \leq j(z), \forall z \in A.\end{array}} \tag{2.1}$$

The cost function j defined on A with values in $\mathbb{R}$ expresses the criteria that the designer wishes to optimize. It is usually given as a function of the grid and of the state

$$j(z) = J(\underline{X}(z), W_h(z)),$$

where the computational grid $\underline{X}(z)$ is an explicit function of the control z and where the state $W_h(z)$ is obtained implicitly by solving a state equation of the form

$$f_h(\underline{X}, W_h) = 0 \in \mathbb{R}^p.$$

This specific structure of the function j is not important hereafter since the optimality conditions to be derived in the coming chapter apply to any form of cost function j.

If the cost function j or the set A of admissible designs are not convex, finding a global minimizer may be very difficult. In such situations, the designer may reduce his ambitions and only look for local minimizers of the cost function around a given reference configuration as defined in the following definition.

Definition 2.1 *A point z^* in the set A endowed with the norm $\|.\|$ is a local minimizer of the cost function j on the set A if and only if there exists a radius*

E. Laporte et al., *Numerical Methods in Sensitivity Analysis and Shape Optimization*

$\delta > 0$ such that z^* minimizes the cost function j on the ball of radius δ centered in z^*:

$$j(z^*) \leq j(z), \forall z \in A \quad \textit{with} \quad \|z - z^*\| \leq \delta.$$

The purpose of the present chapter is now to review the basic conditions of optimality satisfied by the local minimizers of (2.1).

In order to characterize and to compute these local minima in a general abstract mathematical setting, we introduce some notation. First, we will consider the problem after discretization and assume that the set A of admissible designs corresponds to the subset of the parameter space $\mathbb{R}^n$ defined by

$$A = \{z \in \mathbb{R}^n, h_i(z) = 0, \forall i = 1, q, g_j(z) \leq 0, \forall j = 1, m\}. \tag{2.2}$$

We will also assume that the cost function j, the constraints h_i or g_l which are functions of the design parameters z and defined on $\mathbb{R}^n$ with values in $\mathbb{R}$ are differentiable. In this framework, we introduce the Lagrangian $\mathcal{L}(z, \mu, \lambda)$: $\mathbb{R}^n \times \mathbb{R}^q \times \mathbb{R}^m \to \mathbb{R}$ associated to the constrained minimization problem (2.1), and given by

$$\mathcal{L}(z, \mu, \lambda) = j(z) + \sum_{i=1}^{q} \mu_i h_i(z) + \sum_{l=1}^{m} \lambda_l g_l(z). \tag{2.3}$$

To compute gradients, we will identify $\mathbb{R}^n$ to its dual $(\mathbb{R}^n)^*$, and define the gradient of the real differentiable function $j(z)$ as the vector of $\mathbb{R}^n$ of coordinates

$$(\nabla j)_i = \frac{\partial j}{\partial z_i}, i = 1, n.$$

Similarly, the Hessian of j will be defined as the symmetric matrix

$$(\nabla^2 j)_{ij} = \frac{\partial^2 j}{\partial z_i \partial z_j}, \forall i = 1, n, \forall j = 1, n.$$

The action of these derivatives along given directions d of $\mathbb{R}^n$ will then be defined as the dot products

$$\nabla j \cdot d = \sum_{i=1}^{n} (\nabla j)_i d_i,$$

$$(\nabla^2 j \cdot d)_i = \sum_{k=1}^{n} (\nabla^2 j)_{ik} d_k.$$

2.2 Necessary conditions of optimality

Most numerical methods compute local minimizers by solving the Euler–Lagrange optimality conditions satisfied by these minima. These optimality conditions are based on the notion of feasible search directions.

Definition 2.2 *A vector d in $\mathbb{R}^n$ is a feasible search direction in the set A at point $z \in A$ if and only if there exists a positive number $\theta > 0$ such that the whole segment*

$$[z, z+\theta d] = \{x \in \mathbb{R}^n, x = z + td, 0 \leq t \leq \theta\}$$

is included in the set A.

By construction, the action of the gradient of the cost function j on the direction d characterizes the rate of variation of this cost function along direction d. For any local minimum z^* and for any feasible direction d, this variation must be positive since the cost function cannot decrease next to a minimum along any admissible direction. This immediately yields a first necessary condition of optimality satisfied by all local minima

Theorem 2.1 *Let $z^* \in A$ be a local minimizer of the cost function j on the set A, and let $\nabla j(z^*)$ be its gradient at point z^*. Then, all feasible search directions d in A at z^* satisfy*

$$\nabla j(z^*) \cdot d \geq 0, \forall d \text{ feasible.}$$

Remark 2.1 Introducing a second-order Taylor expansion of the cost function j from its Hessian $H = \nabla^2 j$ $(H_{ij} = \partial^2 j / \partial z_i \partial z_j)$ yields a finer characterization of the local minima of j on A. It can be easily proved that at any local minimizer z^* of j on A where the cost function j has second derivatives, all feasible search directions must satisfy the optimality conditions

$$\begin{aligned} &\text{either } \nabla j(z^*) \cdot d > 0, \\ &\text{or } \nabla j(z^*) \cdot d = 0 \quad \text{and} \quad d \cdot \nabla^2 j \cdot d \geq 0. \end{aligned}$$

The notion of feasible search directions is well adapted to problems which only involve inequality constraints in the definition of the set A of admissible designs. It is too restrictive in the presence of equality constraints where the local minima z^* must be looked for on the surfaces of equation $h_i(z) = 0$. In such cases, the notion of feasible search direction is to be replaced by the notion of feasible tangent vector

Definition 2.3 *A nonzero vector $d \in \mathbb{R}^n$ is a feasible tangent vector to the set A in z if and only if there exists a differentiable half curve on A,*

$$\varphi : [0, \theta] \to A,$$

with $\varphi(0) = z$ and whose tangent at origin $\frac{d\varphi}{ds}|_{s=0}$ is equal to the vector d.

A simple exercise in differential calculus leads then to the generalization of the above optimality condition.

Theorem 2.2 *Let z^* be a local minimizer of the cost function j on the set A of admissible design, and let $\nabla j(z^*)$ be the gradient of j in z^*. Then each feasible tangent vector to A in z^* must satisfy the optimality condition*

$$\nabla j(z^*) \cdot d \geq 0, \text{ for all feasible tangent vectors } d.$$

2.3 Optimality conditions of Euler–Lagrange

2.3.1 Equality constraints

The above optimality conditions take a much simpler form when the set A is locally defined by smooth constraints [2, 48]. Let us first consider the case without inequality constraints. For such problems, the set A looks locally like a smooth surface of equation $h_i(z) = 0, \forall i = 1, q$. We assume that the minimizer z^* is a regular point of A.

Assumption 2.1 *Let z^* be given in A and suppose that z^* is a regular point of A, meaning that there exists q differentiable real functions $h_i(z)$ defined on $\mathbb{R}^n$ such that:*

- *the vectors $\nabla h_i(z^*)$ are linearly independent in $\mathbb{R}^n$,*
- *any element z in a neighborhood of z^* belonging to A satisfies $h_i(z) = 0, \forall i = 1, q$ and conversely.*

A direct application of the implicit function theorem then yields the following lemma:

Lemma 2.1 *If z^* is a regular point of A satisfying the above asssumption, then a given vector d in $\mathbb{R}^n$ is a feasible tangent vector to A in z^* if and only if*

$$\nabla h_i(z^*) \cdot d = 0, \forall i = 1, q.$$

A direct consequence of this lemma is then:

Theorem 2.3 *If z^* is a regular point of A satisfying the above assumption, then z^* is a local minimizer of the cost function j on the A of admissible designs only if there exists q real numbers $\mu_1^*, \mu_2^*, \ldots, \mu_q^*$ (the so-called Lagrange multipliers) satisfying the Euler–Lagrange equations*

$$\nabla j(z^*) + \sum_{i=1}^{q} \mu_i^* \nabla h_i(z^*) = 0 \quad in \quad \mathbb{R}^n, \tag{2.4}$$

$$h_i(z^*) = 0, \forall i = 1, q. \tag{2.5}$$

Proof The proof is very simple but quite fundamental. Let d be a vector of $\mathbb{R}^n$ such that

$$\nabla h_i(z^*) \cdot d = 0, \forall i = 1, q.$$

From the above lemma, d is a feasible tangent vector and therefore satisfies the necessary optimality condition

$$\nabla j(z^*) \cdot d \geq 0.$$

But by construction, $(-d)$ satisfies the same equation

$$\nabla h_i(z^*) \cdot (-d) = 0, \forall i = 1, q,$$

and therefore the same reasoning yields

$$\nabla j(z^*) \cdot (-d) \geq 0.$$

Combining both inequalities leads to the equality

$$\nabla j(z^*) \cdot d = 0, \forall d \in Ker\ B,$$

where B is the linear map defined by

$$\begin{array}{rcl} B : \mathbb{R}^n & \to & \mathbb{R}^q, \\ d & \to & B \cdot d = (\nabla h_i(z^*) \cdot d)_{i=1,q} \end{array}$$

and $Ker\ B = \{d \in \mathbb{R}^n, B \cdot d = 0\}$ is its kernel. In linear algebra, the above equality means that the gradient of j belongs to the space orthogonal to the kernel of B,

$$\nabla j(z^*) \in Ker\ B^{\perp}.$$

From the closed range theorem ($Ker\ B^{\perp} = Im\ B^t$), this implies that $\nabla j(z^*)$ belongs to the range of B^t, that is that there exists $\mu^* = (\mu_1^*, \mu_2^*, \cdots, \mu_q^*) \in \mathbb{R}^q$ such that

$$\nabla j(z^*) = B^t(-\mu^*) = -\sum_{i=1}^{q} \mu_i^* \nabla h_i(z^*).$$

Our theorem is therefore proved by observing that, in addition, the equalities $h_i(z^*) = 0$ are a direct consequence of the definition of the set A of admissible designs. □

Remark 2.2 As before, using the second derivatives of the cost function yields a finer characterization of the local minima of j, to be proved later in the general case with equality and inequality constraints.

- If z^* is a local minimum of j, and if all involved functions are twice differentiable, then z^* satisfies the Euler–Lagrange equations together with the inequality

$$d \cdot (\nabla^2 j(z^*) + \sum_{i=1}^{q} \mu_i^* \nabla^2 h_i(z^*)) \cdot d \geq 0, \forall d \in Ker\ B.$$

- Conversely, if z^* satisfies the Euler–Lagrange equations and if

$$\nabla^2 j(z^*) + \sum_{i=1}^{q} \mu_i^* \nabla^2 h_i(z^*)$$

is positive definite on $Ker\ B$, then z^* is a local minimizer of j on A.

2.3.2 Inequality constraints

The extension of Lagrange's theorem to the case with inequality constraints is based on the notion of active constraints.

Definition 2.4 *Let A be the set of admissible designs defined in $\mathbb{R}^n$ by the continuous equalities and inequalities*

$$A = \{z \in \mathbb{R}^n, h_i(z) = 0, \forall i = 1, q, \quad g_l(z) \leq 0, \forall l = 1, m\}.$$

The set $I(z^*)$ of active constraints at point $z^* \in A$ is defined as the set of indices $l \in \{1, \ldots, m\}$ such that $g_l(z^*) = 0$. In other words,

$$I(z^*) = \{l \in \{1, \ldots, m\}, g_l(z^*) = 0\}.$$

By continuity, the constraints g_l which are not active at point z^* will remain strictly negative in a neighborhood of z^*. Therefore, they will automatically be satisfied locally, and can thus be ignored.This argument reduces then the case with inequality constraints to the case with equality constraints and leads to the Kuhn–Tucker optimality conditions.

Theorem 2.4 *Let z^* be a local minimizer of the cost function j on the set A of admissible designs. Let $I(z^*)$ be the set of active constraints in z^*. If j, h_i and g_l are locally differentiable around z^* and if the $q + card\ I(z^*)$ vectors*

$$\{\nabla h_1(z^*), \ldots, \nabla h_q(z^*), \nabla g_l(z^*)_{l \in I(z^*)}\}$$

are linearly independent in $\mathbb{R}^n$, then there exist q Lagrange multipliers $\mu^* \in \mathbb{R}^q$ and m Lagrange multipliers $\lambda^* \in \mathbb{R}^m$ such that

$$\frac{\partial \mathcal{L}}{\partial z}(z^*, \mu^*, \lambda^*) = \nabla j(z^*) + \sum_{i=1}^{q} \mu_i^* \nabla h_i(z^*) + \sum_{l=1}^{m} \lambda_l^* \nabla g_l(z^*) = 0, \quad (2.6)$$

$$h_i(z^*) = 0, \forall i = 1, q, \quad (2.7)$$

$$\lambda_l^* g_l(z^*) = 0, \forall l = 1, m, \quad (2.8)$$

$$\lambda_l^* \geq 0, g_l(z^*) \leq 0, \forall l = 1, m. \quad (2.9)$$

Proof As observed above, there exists a ball B_δ or radius δ centered in z^* all of whose internal points satisfy the inactive constraints:

$$g_l(z) \leq 0, \forall l \notin I(z^*), \forall z \text{ with } \|z - z^*\| \leq \delta.$$

The set $A_I \cap B_\delta$ with A_I defined by

$$A_I = \{z \in \mathbb{R}^n, h_i(z) = 0, \forall i = 1, q, \quad g_l(z) = 0, \forall l \in I(z^*)\}$$

is thus included in A. By construction, z^* being a local minimizer of j on A is also a minimizer of j on A_I. Therefore, from the Lagrange theorem, there exist

real multipliers $(\mu_i^*)_{i=1,q}$ and $(\lambda_l^*)_{l\in I(z^*)}$ such that

$$\begin{aligned}
&\nabla j(z^*) + \sum_{i=1}^{q} \mu_i^* \nabla h_i(z^*) + \sum_{l\in I(z^*)} \lambda_l^* \nabla g_l(z^*) = 0,\\
&h_i(z^*) = 0, \forall i = 1, q,\\
&g_l(z^*) = 0, \forall l \in I(z^*).
\end{aligned}$$

Setting $\lambda_l^* = 0$ if $l \notin I(z^*)$, and from the construction of $z^* \in A$, this system can also be written as

$$\begin{aligned}
&\nabla j(z^*) + \sum_{i=1}^{q} \mu_i^* \nabla h_i(z^*) + \sum_{l=1}^{m} \lambda_l^* \nabla g_l(z^*) = 0 \quad \text{in} \quad \mathbb{R}^m,\\
&h_i(z^*) = 0, \forall i = 1, q,\\
&g_l(z^*) = 0, \forall l \in I(z^*),\\
&\lambda_l^* = 0, g_l(z^*) \leq 0, \forall l \notin I(z^*).
\end{aligned}$$

Moreover, by construction, for any $l \in I(z^*)$, any vector d_l such that

$$\begin{aligned}
&\nabla h_i(z^*) \cdot d_l = 0, \forall i = 1, q,\\
&\nabla g_k(z^*) \cdot d_l = 0, \forall k \neq l \in I(z^*),\\
&\nabla g_l(z^*) \cdot d_l = -1,
\end{aligned}$$

is a feasible tangent vector to A in z^*. Such a vector exists from the assumed linear independence of $\nabla h_i(z^*)$ and $\nabla g_l(z^*)$. This therefore implies that the variation of j along d_l is positive,

$$\nabla j(z^*) \cdot d_l \geq 0,$$

which, plugged in the first equation of the above system, yields

$$\lambda_l^* = \nabla j(z^*) \cdot d_l \geq 0, \forall l \in I(z^*),$$

and our theorem is complete. □

Remark 2.3 The above proof gives a very powerful interpretation of the Lagrange multiplier λ_l^*, as the rate of variation of the cost function j along the direction d_l which enters the admissible set normally to the constraint g_l and tangentially to all other constraints.

As before, the first order optimality conditions (2.6)–(2.9) can be replaced by finer characterizations if the functions are twice differentiable.

Theorem 2.5 *Let (z^*, μ^*, λ^*) be a solution of the Kuhn–Tucker optimality conditions (2.6)–(2.9). We again suppose that the $m + card\ I(z^*)$ vectors*

$$\{\nabla h_1(z^*), \ldots, \nabla h_q(z^*), \nabla g_l(z^*)_{l\in I(z^*)}\}$$

are linearly independent in $\mathbb{R}^n$, and we also suppose that the Lagrangian $\mathcal{L}$ defined in (2.3) is twice differentiable in z at point (z^*, μ^*, λ^*). We finally introduce the kernel

$$Ker\, B_I = \{d \in \mathbb{R}^n, \nabla h_i(z^*) \cdot d = 0, \forall i = 1, q,\ \nabla g_l(z^*) \cdot d = 0, \forall l \in I(z^*)\}.$$

Under these assumptions and notation, if z^* is a local minimizer of the cost function j on the set A, then the Hessian of $\mathcal{L}$ is positive on $Ker\, B_I$,

$$d \cdot \frac{\partial^2 \mathcal{L}}{\partial z^2}(z^*, \mu^*, \lambda^*) \cdot d \geq 0,\ \forall d \in Ker\, B_I. \tag{2.10}$$

Conversely, if the Hessian of $\mathcal{L}$ is strictly positive on $Ker\, B_I$,

$$d \cdot \frac{\partial^2 \mathcal{L}}{\partial z^2}(z^*, \mu^*, \lambda^*) \cdot d > 0,\ \forall d \neq 0 \in Ker\, B_I, \tag{2.11}$$

and if the Lagrange multipliers λ_l^* are strictly positive for all $l \in I(z^*)$, then z^* is a strict local minimizer of j on A.

Proof Let (z^*, μ^*, λ^*) be a solution of the optimality conditions (2.6)–(2.9) such that the $q + card\, I(z^*)$ vectors $\{\nabla h_1(z^*), \ldots, \nabla h_q(z^*), \nabla g_l(z^*)_{l \in I(z^*)}\}$ are linearly independent in $\mathbb{R}^n$. Let us introduce the extended set

$$A_I = \{z \in \mathbb{R}^n, h_i(z) = 0, \forall i = 1, q,\ g_l(z) = 0, \forall l \in I(z^*)\}.$$

By continuity, there exists a ball B_δ of radius δ and centered in z^* such that

$$g_l(z) < 0, \forall l \notin I(z^*), \forall z \in B_\delta.$$

Let us now consider an arbitrary vector $d \in Ker\, B_I$. From the implicit function theorem, there exists a curve $\varphi(s)$ on $A_I \cap B_\delta \subset A$ originating from z^* ($\varphi(0) = z^*$) and tangent to d in z^* ($\frac{d\varphi}{ds}(0) = d$). (Conversely, we could prove that the tangent vector at z^* to any smooth curve on $A_I \cap B_\delta$ must belong to $Ker\, B_I$.) By construction of A_I, this means that we have

$$h_i(\varphi(s)) = g_l(\varphi(s)) = 0 \text{ locally in } s, \forall i = 1, q, \forall l \in I(z^*).$$

Hence, a double differentiation with respect to s yields

$$\begin{aligned}
0 &= \left(\frac{d^2}{ds^2} h_i(\varphi(s))\right)_{|s=0} \\
&= \frac{d\varphi}{ds}(0) \cdot \nabla^2 h_i(z^*) \cdot \frac{d\varphi}{ds}(0) + \nabla h_i(z^*) \cdot \frac{d^2\varphi}{ds^2}(0), \\
0 &= \left(\frac{d^2}{ds^2} g_l(\varphi(s))\right)_{|s=0} \\
&= \frac{d\varphi}{ds}(0) \cdot \nabla^2 g_l(z^*) \cdot \frac{d\varphi}{ds}(0) + \nabla g_l(z^*) \cdot \frac{d^2\varphi}{ds^2}(0).
\end{aligned}$$

By summation, and since λ_l^* is zero when l does not belong to the set $I(z^*)$ of active constraints, we get

$$-\left(\sum_{i=1}^{q}\mu_i^*\nabla h_i(z^*)+\sum_{l=1}^{m}\lambda_l^*\nabla g_l(z^*)\right)\cdot\frac{d^2\varphi}{ds^2}(0)$$
$$=d\cdot\left(\sum_{i=1}^{q}\mu_i^*\nabla^2 h_i(z^*)+\sum_{l=1}^{m}\lambda_l^*\nabla^2 g_l(z^*)\right)\cdot d.$$

Using equation (2.6), this yields

$$\nabla j(z^*)\cdot\frac{d^2\varphi}{ds^2}(0)=d\cdot\frac{\partial^2\mathcal{L}}{\partial z^2}(z^*,\mu^*,\lambda^*)\cdot d-d\cdot\nabla^2 j(z^*)\cdot d.$$

Regrouping all terms in j, we observe that the second rate of change of the cost function along the curve $\varphi(s)$ is finally given by

$$\left(\frac{d^2}{ds^2}j(\varphi(s))\right)_{|s=0}=d\cdot\frac{\partial^2\mathcal{L}}{\partial z^2}(z^*,\mu^*,\lambda^*)\cdot d. \tag{2.12}$$

If z^* is a local minimum of j on A, it will be a local minimum of j on the curve $\varphi(s)$ implying

$$0\leq\left(\frac{d^2}{ds^2}j(\varphi(s))\right)_{|s=0}=d\cdot\frac{\partial^2\mathcal{L}}{\partial z^2}(z^*,\mu^*,\lambda^*)\cdot d,$$

which is the desired condition (2.10).

Conversely, if the Hessian of $\mathcal{L}$ is strictly positive on $Ker\,B_I$, then the above equality (2.12) implies that we have $\left(\frac{d^2}{ds^2}j(\varphi(s))\right)_{|s=0}>0$ on all curves of A_I going through z^*. Combined with (2.6) which yields

$$\left(\frac{d}{ds}j(\varphi(s))\right)_{|s=0}=\nabla j(z^*)\cdot d=\frac{\partial\mathcal{L}}{\partial z}(z^*,\mu^*,\lambda^*)\cdot d=0,$$

with $d\in Ker\,B_I$ the tangent vector to the curve $\varphi(s)$ at point $s=0$. This guarantees that z^* is a local minimizer of j on all curves of A_I. If we now consider a curve which is not tangent to A_I, that is a curve which enters inside A with $\nabla g_l(z^*)\cdot\frac{d\varphi}{ds}(0)<0$ for some $l\in I(z^*)$, then equation (2.6) and the strict positivity of λ_l^* will yield

$$\left(\frac{d}{ds}j(\varphi(s))\right)_{|s=0}=\nabla j(z^*)\cdot d=\frac{\partial\mathcal{L}}{\partial z}(z^*,\mu^*,\lambda^*)\cdot d-\sum_l\lambda_l^*\nabla g_l(z^*)\cdot d>0,$$

implying again that z^* is a strict local minimizer of j on the part of the curve φ which belongs to A. Altogether, this proves that z^* is a local minimizer of j on A and our theorem is complete. □

2.4 Exercises

The exercises below should help the reader to become familiar with standard optimality conditions.

Microeconomics

A company wants to produce 1000 units of a given product in two production plants. Let $h_1(x)$ (respectively $h_2(x)$) represent the cost of producing x units in the first (respectively the second) plant. Prove that the production x_1 of the first plant minimizing the company cost is either $x_1 = 0$, $x_1 = 1000$ or is a solution of the scalar equation

$$h_1'(x_1) = h_2'(1000 - x_1).$$

Boltzmann distribution in statistical physics

A common problem in statistical physics is to find a density distribution $x = (x_i)_{i=1,n}$ minimizing the entropy $j(x) = \sum_{i=1}^n x_i \ln(x_i)$ under the constraints

$$\begin{aligned} &x_i \geq 0, \text{ positive distributions,} \\ &\sum_{i=1}^n x_i = \bar{x}, \text{ given total mass,} \\ &\sum_{i=1}^n x_i E_i = \bar{E}, \text{ given total enegy,} \end{aligned}$$

with $0 < E_1 < E_2 < \ldots < E_n$. To completely solve this problem, we introduce the admissible and strictly admissible subsets

$$\begin{aligned} A &= \{x \in \mathbb{R}^n, x_i \geq 0, \sum_{i=1}^n x_i = \bar{x}, \sum_{i=1}^n x_i E_i = \bar{E}\}, \\ \mathring{\mathbf{A}} &= \{x \in \mathbb{R}^n, x_i > 0, \sum_{i=1}^n x_i = \bar{x}, \sum_{i=1}^n x_i E_i = \bar{E}\}, \end{aligned}$$

and assume that $\mathring{\mathbf{A}}$ is not empty.

Prove that j has a unique local minimizer on A. Prove also that all local minimizers x^* of j on $\mathring{\mathbf{A}}$ are solutions of the nonlinear system

$$\ln x_i^* + \alpha^* + \beta^* E_i = 0, \forall i, \tag{2.13}$$

$$\sum_{i=1}^n x_i^* = \bar{x}, \tag{2.14}$$

$$\sum_{i=1}^n x_i^* E_i = \bar{E}. \tag{2.15}$$

Prove then that the system (2.13)–(2.15) has a unique solution (x^*, α^*, β^*) in $\mathring{\mathbf{A}} \times \mathbb{R}^2$, and that the corresponding vector x^* is a local minimizer of j on $\mathring{\mathbf{A}}$, hence the unique local (and global) minimizer of j on A.

Chapter 3

Newton's Algorithms

3.1 The problem to solve

In the absence of any inequality constraints g_l on the design parameters z, the shape optimization problem (P_h) introduced in the first chapter reduces to the constrained minimization problem

$$\boxed{\min_{z \in \mathbb{R}^n,\, h_i(z)=0} j(z).} \tag{3.1}$$

The cost function j and design constraints $h_i(z)$ are defined as real functions $j(z) = J(\underline{X}(z), W_h(z))$ and $h_i(z) = h_i(\underline{X}(z), W_h(z))$ of the control (design) variables z and of the state variables W_h, and the computational grid $\underline{X}(z)$ or the state $W_h(z)$ are obtained by solving the state equation

$$(E_h) \begin{cases} f_h(\underline{X}, W_h) = 0 \quad \text{in} \quad \mathbb{R}^p, \\ \underline{X} = \underline{X}(z). \end{cases}$$

The variables $(\underline{X}(z), W_h(z))$ could be considered as independent variables and the state equation treated as additional equality constraints. This philosophy will be briefly followed at the end of this monograph. However, because of the very large dimension $p + 2ns$ of the equation of state, this point of view is usually not followed in shape optimization or in sensitivity analysis. On the contrary, the basic methodology outlined in the following chapters will consider that the cost function j and design constraints h_i are functions of z only, and that the state equation is just an intermediate step needed for the evaluation of j or of h_i.

In this framework, and as seen in the previous chapter, the optimal solution z^* of the shape optimization problem (3.1) must satisfy the (Euler–Lagrange) optimality equations

$$\nabla j(z^*) + \sum_{i=1}^{q} \mu_i^* \nabla h_i(z^*) = 0 \quad \text{in} \quad \mathbb{R}^n, \tag{3.2}$$

$$h_i(z^*) = 0, \forall i = 1, q, \tag{3.3}$$

E. Laporte et al., *Numerical Methods in Sensitivity Analysis and Shape Optimization*

expressing that the gradient of the cost function must be orthogonal to all feasible tangent vectors. In abstract form, the optimality system can be rewritten as the nonlinear system in $\mathbb{R}^n \times \mathbb{R}^q$,

$$F(z^*, \mu^*) = 0 \in \mathbb{R}^n \times \mathbb{R}^q. \tag{3.4}$$

Computing F might be difficult and expensive. In particular, j and F are functions of the state variables W_h, and therefore any evaluation of j, of the constraints, or of their gradients requires us first to solve exactly the state equation. We will nevertheless suppose for the time being that it is accessible, and we will see in the following chapters how to compute F in practice. The purpose of this chapter is then to describe a basic Newton algorithm which can be used for the solution of the minimization problem (3.1) or of the optimality equation (3.4), with its different variants (quasi-Newton, line search, GMRES).

3.2 Newton's algorithm

As described at length in [40], Newton's algorithm is a general and efficient algorithm for solving a regular nonlinear equation of the type $F(z) = 0$, where $F : \mathbb{R}^N \to \mathbb{R}^N$ is a given differentiable function. This algorithm reduces the solution of this nonlinear problem to the sequence of operations:

For z^k given and while $\|F(z^k)\| > \varepsilon$, do:

- compute the Jacobian $A^k = \frac{dF}{dz}(z^k)$;
- replace $F(z)$ by its first-order approximation $F(z) \approx F(z^k) + A^k \cdot (z - z^k)$, which means that the original problem is replaced by the solution of the linear system
$$A^k \cdot dz = -F(z^k); \quad (S)$$
- set $z^{k+1} = z^k + dz$ and reiterate the process until convergence.

The strict application of Newton's algorithm requires us to compute the Jacobian matrix $A^k = \dfrac{dF}{dz}(z^k)$ and to solve a linear system associated to this matrix. This might be too demanding. Quasi-Newton variants are obtained by replacing the solution of the linear system (S) by the solution of a simpler approximate system.

Such Newton or quasi-Newton algorithms have usually a very fast convergence rate next to the solution, as indicated by the following theorem [10, 9, 5]:

Theorem 3.1 *Let us consider a quasi-Newton algorithm whose generic iteration satisfies*

$$z^+ = z + d(z) + \varepsilon(z),\ z \in B(z^*, \rho),$$

where the Newton update $d(z)$ is given by

$$d(z) = -(\frac{dF}{dz})^{-1} \cdot F(z).$$

Suppose that

1. *the function* $F : \mathbb{R}^N \to \mathbb{R}^N$ *is twice continuously differentiable,*
2. *there exists a solution* z^* *to the equation* $F(z) = 0$,
3. *for all* z *in the ball* $B(z^*, \rho)$, *the Jacobian inverse* $(\frac{\partial F}{\partial z})^{-1}$ *is well defined,*
4. *the perturbation* $\varepsilon(z)$ *satisfies*

$$\|\varepsilon(z)\| \leq c_1 \|d(z)\|^2, \forall z \in B(z^*, \rho) \quad (\mathcal{H}).$$

Then there exists a constant $c_2 > 0$ *such that*

$$\|z^+ - z^*\| \leq c_2 \|z - z^*\|^2, \forall z \in B(z^*, \rho).$$

In other words, in a neighborhood of the solution , the convergence of the quasi-Newton algorithm is quadratic, meaning that the error is squared at each iteration.

Proof By construction, we have

$$\begin{aligned} \|z^+ - z^*\| &\leq \|z^+ - (z + d(z))\| + \|z^* - (z + d(z))\| \\ &\leq c_1 \|d(z)\|^2 + \|(z^* - z) + (\frac{\partial F}{\partial z})^{-1} \cdot F(z)\| \\ &\leq c_1 \|d(z)\|^2 + \|(\frac{\partial F}{\partial z})^{-1} . (\frac{\partial F}{\partial z} \cdot (z^* - z) + F(z) - F(z^*))\|. \end{aligned}$$

A second-order Taylor expansion then yields

$$\|F(z^*) - F(z) - \frac{\partial F}{\partial z} \cdot (z^* - z)\| \leq \frac{1}{2} \sup_{z \in B} \|D^2 F\| . \|z^* - z\|^2.$$

On the other hand, the Lipschitz continuity of F implies

$$\begin{aligned} \|d(z)\| &= \|(\frac{\partial F}{\partial z})^{-1} \cdot (F(z^*) - F(z))\| \\ &\leq \|(\frac{\partial F}{\partial z})^{-1} \| L \| z^* - z\|. \end{aligned}$$

By addition, we finally get

$$\|z^+ - z^*\| \leq \left[c_1 L^2 \|(\frac{\partial F}{\partial z})^{-1}\|^2 + \frac{1}{2} \|(\frac{\partial F}{\partial z})^{-1}\| \|D^2 F\|_{\infty, B} \right] \|z^* - z\|^2.$$

□

Remark 3.1 The key assumption in the above convergence proof concerns the accuracy of the solution of the linear system (S), as imposed by assumption $(\mathcal{H})$ on the perturbation $\varepsilon(z)$. But in any case, the convergence proof is only local, and does not guarantee the global convergence of the algorithm. In practice, this means that Newton's algorithms must be complemented by adequate continuation techniques [28] or line search strategies [9], as explained later.

3.3 Unconstrained optimization

3.3.1 Basic algorithms

The application of Newton's algorithm to unconstrained optimization problems is rather straightforward [9, 18]. For the case $q = 0$ (no constraint), the optimality condition takes the simple form

$$F(z^*) := \frac{dj}{dz}(z^*) = 0, \tag{3.5}$$

and its solution by a Newton algorithm leads to the following sequence of operations.

For z^k given and while $\|\frac{dj}{dz}(z^k)\| > \varepsilon$, do:

- compute the Jacobian $A^k = \frac{d^2j}{dz^2}(z^k)$;
- solve the linear system $A^k \cdot dz = -\frac{dj}{dz}(z^k)$; (S)
- set $z^{k+1} = z^k + dz$ and reiterate the process until convergence.

3.3.2 Quasi-Newton variant

A simpler variant of Newton's algorithm is to replace the exact Jacobian A^k by a simpler approximation B^k. As an example, we describe below a BFGS type strategy for constructing such an approximation as proposed by Powell [45]. This strategy is usually initialized by choosing B^0 equal to the identity matrix, and builds then a succession of positive definite approximations B^k of the Hessian matrix A^k. It is efficient if this Hessian matrix A^k is well conditioned and does not vary too much between successive iterations.

The Quasi-Newton Matrix Updating

Introduce the variable update

$$\delta = z^{k+1} - z^k$$

and the corresponding functional update

$$\gamma = \frac{dj}{dz}(z^{k+1}) - \frac{dj}{dz}(z^k).$$

The algorithm consists in updating at each step the approximate Hessian B by a rank 2 modification in order to satisfy $B \cdot \delta = \gamma$, yielding

$$B^{k+1} := B^k + \frac{\gamma \otimes \gamma}{\delta \cdot \gamma} - \frac{(B^k \cdot \delta) \otimes (B^k \cdot \delta)}{\delta \cdot B^k \cdot \delta},$$

that is in matrix form

$$B^{k+1} := B^k + \frac{\gamma\gamma^t}{\delta^t\gamma} - \frac{B^k\delta\delta^tB^k}{\delta^tB^k\delta}.$$

The algorithm iterates on k until convergence as follows (with z^0 given and $B^0 = Id$):

- solve the linear system $B^k \cdot dz = -\frac{dj}{dz}(z^k)$;
- set $z^{k+1} = z^k + t_k dz$ with t_k as predicted by the selected line search strategy (see below for more details);
- compute $\delta = t_k dz$ and $\gamma = \frac{dj}{dz}(z^{k+1}) - \frac{dj}{dz}(z^k)$;
- set

$$B^{k+1} := B^k + \frac{\gamma\gamma^t}{\delta^t\gamma} - \frac{B^k\delta\delta^t B^k}{\delta^t B^k \delta}.$$

3.3.3 GMRES solution

Another possibility for implementation of Newton's algorithm is to solve the linear system (S) by a conjugate gradient [19] or by a GMRES algorithm. For these two choices, explicit knowledge of the Jacobian A is not required. What is needed is only the calculation of the directional derivative

$$w_{i+1} = A \cdot v^i = \frac{dF}{dz}(z^k) \cdot v^i \approx \frac{1}{h}(F(z^k + hv^i) - F(z^k)). \tag{3.6}$$

This directional derivative can be obtained either by automatic differentiation or approximated as above by finite differences, using a given small increment h along any direction v^i. The practical calculation of $A \cdot v^i$ will be described in more detail later when dealing with automatic differentiation.

In any case, at each step of the Newton algorithm, that is for a given z^k, the GMRES algorithm as initially developed by Y. Saad and M. Schultz [47] will compute the vector dz as the minimizer of the residual

$$\left\| F(z^k) + \frac{\|dz\|}{h}\left[F(z^k + \frac{h}{\|dz\|}dz) - F(z^k)\right]\right\|_2 \approx \|F(z^k + dz)\|_2 \tag{3.7}$$

on the Krylov space

$$W_{K-1} = \{v, v = \sum_{j=0}^{K-1} y_j A_h^j \cdot r^o\}, \tag{3.8}$$

generated by the initial residual

$$r^o = -F(z^k), \tag{3.9}$$

multiplied by different powers A_h^j of the linearized operator A_h defined by

$$A_h \cdot v = \frac{1}{h}[F(z^k + hv) - F(z^k)]. \tag{3.10}$$

This Krylov space contains the dominating eigenmodes of the Jacobian matrix A, and thus the approximate solution obtained by the GMRES algorithm is supposed to contain the main components of the exact solution. A detailed

description of the GMRES algorithm and some indication of its convergence properties is given in a special section at the end of this chapter. We detail below the Newton–GMRES algorithm obtained by coupling the basic Newton algorithm with a GMRES linear solver. This leads to the following procedure, (where the notation $\langle u, v\rangle = u \cdot v$ denotes the usual dot product of vectors u and v in $\mathbb{R}^n$):

External Newton loop

For z^k given , and while $\|F(z^k)\| > \varepsilon$, do

i) calculation of the initial residual

$$res^o = -F(z^k); \tag{3.11}$$

$$v^1 = res^o/\|res^o\|. \tag{3.12}$$

ii) internal GMRES loop (construction of an orthonormal basis of W_{K-1} by a Gram–Schmidt algorithm, and of the matrix H_K of the linearized operator in this basis).

For i increasing from 1 to K, compute:

$$w^i = \frac{1}{h}[F(z^k + hv^i) - F(z^k)] \text{ or } w^i = \frac{dF}{dz}(z^k) \cdot v^i; \tag{3.13}$$

$$H_{ji} = \left\langle w^i, v^j \right\rangle, \forall j = 1, i; \tag{3.14}$$

$$\tilde{w}^i = w^i - \sum_{j \le i} H_{ji} v^j; \tag{3.15}$$

$$H_{i+1,i} = \|\tilde{w}^i\|; \tag{3.16}$$

$$v^{i+1} = \tilde{w}^i/\|\tilde{w}^i\|. \tag{3.17}$$

End GMRES loop

iii) Newton's update.

$$QR \text{ Factorization of the matrix } H_K = (H_{ij}); \tag{3.18}$$

$$\text{solution of the linear system } Ry = Q^T \|res^o\| \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}; \tag{3.19}$$

$$\text{set } dz = \sum_{i=1}^{K} y_i v^i; \tag{3.20}$$

$$\text{set } z^{k+1} = z^k + t_k dz \text{ with } t_k \text{ given by a line search.} \tag{3.21}$$

Remark 3.2 The convergence of the GMRES algorithm is much better when the condition number of the Jacobian matrix $\frac{dF}{dz}$ of F is close to unity. If an easily computable approximation S of the Jacobian inverse $\frac{dF}{dz}^{-1}$ is available, it is then very efficient to precondition the GMRES loop by S. This amounts to replacing in all steps the function F by the preconditioned function $S \cdot F$.

Remark 3.3 Compared to the previous BFGS algorithm or to other similar quasi-Newton algorithms, the Newton–GMRES procedure reconstructs a different approximation H_K of the Hessian at each Newton step. It is therefore less sensitive to the variations of this Jacobian matrix during the iterations.

Remark 3.4 The Newton–GMRES procedure is efficient when one can compute accurately the gradient $F(v)$ and the products $\frac{dF}{dz} \cdot v$ and if the condition number of the preconditioned matrix $S \cdot \frac{dF}{dz}$ is bounded. It behaves poorly in the presence of penalized constraints which badly affect this condition number.

3.3.4 Line search

The updating strategy

$$z^{k+1} = z^k + dz$$

used in the above implementations of Newton's algorithm is optimal at the asymptotic limit, but might not be so during the first iterations. Therefore, it is much better to replace the simpler updating strategy by a line search formula of the type [9, 5]

$$z^{k+1} = z^k + t_k dz.$$

The first idea will be to use the step t_k which minimizes the cost $j(z^k + tdz)$ on the line $L = \{z = z^k + tdz, t \in \mathbb{R}\}$. Instead of making such an exact minimization on t, it is in fact more efficient to employ inexact line search techniques. The simplest technique is Armijo's Line Search Scheme [1] which defines the step length t as the first number of the sequence $\{1, \nu, \nu^2, \nu^3, ...\}$ satisfying

$$j(z^k + tdz) \leq j(z^k) + t\eta_1 \nabla j(z^k) \cdot dz \tag{3.22}$$

where both η_1 and ν are user defined numbers belonging to the open interval $(0, 1)$. The above scheme selects in the sequence the first term achieving a sufficient decrease of the cost function j. Line search strategies will be further detailed in the general case of constrained optimization.

3.4 Constrained optimization

3.4.1 Basic algorithm

Let us now consider the general constrained minimization problem (3.1). Solving the corresponding optimality conditions (3.2)–(3.3)

$$F := \begin{bmatrix} \nabla j + \sum_h \mu_i \nabla h_i \end{bmatrix} = 0$$

by a Newton algorithm leads to the following sequence of operations.

For (z^k, μ^k) given and while $||F(z^k, \mu^k)|| > \varepsilon$, do:

- compute an approximation H^k of the Jacobian $\nabla^2 j(z^k) + \sum_{i=1}^{q} \mu_i^k \nabla^2 h_i(z^k)$;
- solve the symmetric linear system in $(dz, d\mu)$,

$$H^k \cdot dz + \sum_{i=1}^{q} d\mu_i \nabla h_i(z^k) = -\nabla j\left(z^k\right) - \sum_{i=1}^{q} \mu_i^k \nabla h_i\left(z^k\right), \quad (3.23)$$

$$\nabla h_i(z^k) \cdot dz = -h_i(z^k), \forall i = 1, q; \quad (3.24)$$

- set $z^{k+1} = z^k + dz$, $\mu^{k+1} = \mu^k + d\mu$ and reiterate the process until convergence.

3.4.2 Han's penalized cost function

We have seen earlier that the update $z^{k+1} = z^k + dz$, $\mu^{k+1} = \mu^k + d\mu$ proposed by Newton's algorithm is very efficient during the last steps of the algorithm and guarantees local quadratic convergence. On the other hand, the relevance of dz is not so clear when we are far away from the final solution z^*. A more detailed analysis can in fact prove that if the approximate Hessian H^k is positive (which is the case next to the solution as implied by the second-order optimality conditions of the previous chapter), then the direction dz is a proper direction of descent, meaning that following this direction will lead to a decrease of the cost function. More precisely, if we introduce a sequence of positive numbers $(c_i)_{i=1,q}$ such that

$$c_i > |\mu_i^k + d\mu_i|, \quad c_i > |\mu_i^*|, \forall i = 1, q, \quad (3.25)$$

and if we build the associated penalized cost function (often called Han's penalized cost function)

$$j_c(z) = j(z) + \sum_{i=1}^{q} c_i |h_i(z)|,$$

we have

Theorem 3.2 *Let z^* be a local minimizer of the cost function j over A, and let μ^* be the corresponding Lagrange multipliers. Let $(dz, d\mu)$ be the solution of the Newton linear system (3.23)-(3.24). Under assumption (3.25), z^* is a local minimizer of the penalized cost function j_c over the whole space $\mathbb{R}^n$. If in addition the approximate Hessian H^k is positive, the vector dz is a direction of descent for this penalized cost function.*

Proof To first prove that z^* is a local minimizer of j_c, we compute the rate of variation of j_c along any direction d, using the optimality condition (3.2). This yields

$$\begin{aligned} j_c(z^* + \varepsilon d) - j_c(z^*) &= \varepsilon \nabla j(z^*) \cdot d + \sum_{i=1}^{q} \varepsilon c_i \operatorname{sgn}(\nabla h_i(z^*) \cdot d) \nabla h_i(z^*) \cdot d + o(\varepsilon), \\ &= \sum_{i=1}^{q} \varepsilon \Big[c_i \operatorname{sgn}(\nabla h_i(z^*) \cdot d) - \mu_i^* \Big] \nabla h_i(z^*) \cdot d + o(\varepsilon), \\ &= \sum_{i=1}^{q} \varepsilon \Big[c_i - \mu_i^* \operatorname{sgn}(\nabla h_i(z^*) \cdot d) \Big] |\nabla h_i(z^*) \cdot d| + o(\varepsilon), \\ &> 0, \quad \forall \varepsilon > 0, \forall d \in \mathbb{R}^n. \end{aligned}$$

Similarly, taking into account equations (3.23)–(3.24) and recalling the notation $\mathcal{L}(z, \mu) = j(z) + \sum_{i=1}^{q} \mu_i h_i(z)$, the rate of variation of j_c along direction dz at point z^k is given by

$$\begin{aligned} \nabla j_c(z^k) \cdot dz &= \nabla j(z^k) \cdot dz + \sum_{i=1}^{q} c_i \operatorname{sgn}(h_i(z^k)) \nabla h_i(z^k) \cdot dz, \\ &= \frac{\partial \mathcal{L}}{\partial z}(z^k, \mu^k + d\mu) \cdot dz \\ &\quad + \sum_{i=1}^{q} \Big[c_i \operatorname{sgn}(h_i(z^k)) - (\mu_i^k + d\mu_i) \Big] \nabla h_i(z^k) \cdot dz, \\ &= -\frac{\partial \mathcal{L}}{\partial z}(z^k, \mu^k + d\mu) \cdot (H^k)^{-1} \cdot \frac{\partial \mathcal{L}}{\partial z}(z^k, \mu^k + d\mu) \\ &\quad - \sum_{i=1}^{q} \Big[c_i \operatorname{sgn}(h_i(z^k)) - (\mu_i^k + d\mu_i) \Big] h_i(z^k), \\ &= -\frac{\partial \mathcal{L}}{\partial z}(z^k, \mu^k + d\mu) \cdot (H^k)^{-1} \cdot \frac{\partial \mathcal{L}}{\partial z}(z^k, \mu^k + d\mu) \\ &\quad - \sum_{i=1}^{q} \Big[c_i - (\mu_i^k + d\mu_i) \operatorname{sgn}(h_i(z^k)) \Big] |h_i(z^k)|, \\ &\leq 0. \end{aligned}$$

□

3.4.3 Line search

The above theorem leads to a very simple and robust line search strategy for upgrading the performances of the original Newton's algorithm in constrained optimization. Since dz is a direction of descent of j_c and since we are looking for a control z^* which is a local minimizer of j_c, it makes sense to keep dz as a direction of update. Moreover, since we are looking for a minimizer of j_c, the best upgrade $z^k + t_k dz$ of z along this direction seems to be the one which leads

to the lowest value of j_c, that is the solution of the minimization problem

$$t_k = \text{Argmin}_{t>0} j_c(z^k + tdz).$$

This simple idea leads to the following global Newton's algorithm for constrained optimization

ALGORITHM 1

For (z^k, μ^k) given and while $\|F(z^k, \mu^k)\| = \left\| \begin{matrix} \nabla j\left(z^k\right) + \sum_i \mu_i^k \nabla h_i\left(z^k\right) \\ h\left(z^k\right) \end{matrix} \right\| > \varepsilon$, do:

- compute a positive approximation H^k of the Jacobian of F_1,

$$A^k = \nabla^2 j(z^k) + \sum_{i=1}^{q} \mu_i^k \nabla^2 h_i(z^k);$$

- solve the symmetric linear system in $(dz, d\mu)$,

$$H^k \cdot dz + \sum_{i=1}^{q} d\mu_i \nabla h_i(z^k) = -\nabla j\left(z^k\right) - \sum_{i=1}^{q} \mu_i^k \nabla h_i\left(z^k\right), \quad (3.26)$$

$$\nabla h_i(z^k) \cdot dz = -h_i(z^k), \forall i = 1, q; \quad (3.27)$$

- compute $c_i = \max(|\mu_i^k|, |\mu_i^k + d\mu_i|) + c_{min}$;
- compute an approximate solution t_k of the one-dimensional minimization problem

$$t_k = \text{Argmin}_{t>0} \left(j(z^k + tdz) + \sum_{i=1}^{q} c_i |h_i(z^k + tdz)| \right);$$

- set $z^{k+1} = z^k + t_k dz$, $\mu^{k+1} = \mu^k + t_k d\mu$ and reiterate the process until convergence.

Remark 3.5 The simplest positive approximation of the Hessian is the identity matrix $H^k = Id$. This choice corresponds to the so-called steepest descent algorithms. A slightly more complicated choice is to use the BFGS update reviewed in the previous section. The more elaborate choice is to take the full Hessian, to perform a symmetric Gaussian factorization, and if needed, to modify the diagonal pivots in order to restore positivity. These different implementations are available in most commercial and freeware optimization softwares.

Remark 3.6 The one-dimensional minimization problem in t_k can be solved by iterative cubic interpolation [18, 5], using the stopping criteria as proposed by Wolfe [51]. A simpler strategy is Armijo's Line Search Scheme applied to

the penalized function j_c which defines the step length t_k as the first number of the sequence $\{1, \nu, \nu^2, \nu^3, ...\}$ satisfying

$$j_c(z^k + tdz) \leq j_c(z^k) + t\eta_1 \nabla j_c(z^k) \cdot dz. \tag{3.28}$$

Remark 3.7 The choice of the penalizing factor c_i requires some care. Indeed, we want to recover the quadratic convergence of the Newton algorithm next to the solution. This means in particular that t_k must be close to 1 when we approach the solution. But, if the value of c_i is too large, the minimum of $j_c(z^k + tdz)$ will not be reached for $t = 1$ close to the solution, destroying the quadratic convergence of the algorithm, and leading to the so-called Maratos effect [5].

3.5 Complement: gradient based algorithms

Another basic algorithm in shape optimization is the steepest descent [25, 35]. It can be viewed as a particular case of the global quasi-Newton algorithm **ALGORITHM 1** corresponding to the choice $H^k = Id$. This simple method is well adapted for unconstrained optimization problems, but it can also handle optimization problems with inequality constraints if one can construct a simple projection operator $Proj_{z_{adm}}$ from $\mathbb{R}^n$ on to the set Z_{adm} of admissible designs. This algorithm reduces the global minimization problem to the succession of one-dimensional minimization problems $(k = 1, 2, \ldots)$

$$(P_k) \quad \min_{t>0} j[Proj_{z_{adm}}(z^k - t\nabla j(z^k))], \quad Z_{adm} = \{z, g(z) \leq 0\}.$$

Each problem (P_k) consists in fact in approximating the cost function by its first-order linear approximation, which amounts in the unconstrained case to the minimization of the original cost function j on its line of steepest descent going through the point z^k. Each problem (P_k) is usually solved by a secant algorithm. The gradient $\nabla j(z^{(k)})$ can be computed by a finite difference technique

$$(\nabla j)_i \approx \frac{[j(z + \varepsilon e_i) - j(z)]}{\varepsilon}$$

or by the automatic differentiation techniques to be presented in Chapter 5. As for the projection step, it can take different forms depending on the expression of the constraints. In the particular case where Z_{adm} is equal to the set of positive vectors $(g(z) = -z \leq 0)$, this projection $Proj_{z_{adm}}(v)$ consists simply in replacing $v = (v_i)_{i=1,n}$ by its positive part $v_+ = \big(max(0, v_i)\big)_{i=1,n}$.

There are many generalizations of this steepest descent algorithm. In particular, Fleury and Braibant [15] have proposed to impose convexity at each step by using a first-order expansion of the cost function with respect to either the original variable if the corresponding derivative is positive or to its inverse if this derivative is negative. By similarly linearizing the design constraints , the problem can thus be reduced to a separable local approximate convex minimization problem easy to solve by standard duality techniques. A variant of this technique replaces in (P_k) the cost function by the following convex multidimensional approximation

$$(P'_k) \quad \min_z \left[j(z^k) + \sum_{i=1}^{n} \Big(\frac{C_i}{R_i - z_i} + \frac{D_i}{z_i - S_i} + A_i + B_i z_i \Big) \right].$$

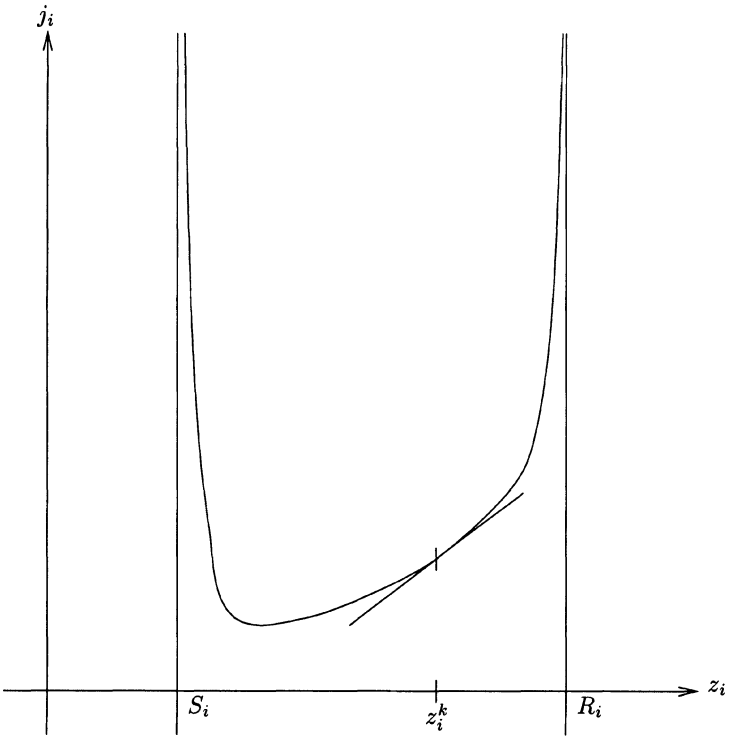


Figure 3.1. Approximation of a cost function by moving asymptotes

In this moving asymptotes technique described in Svanberg [49] (Figure 3.1), the position of the asymptotes $R_i \in]z_i^{(k)}, +\infty]$, $S_i \in [-\infty, z_i^{(k)}[$ and the values of the coefficients A_i, B_i, C_i and D_i are chosen in order to satisfy the following requirements:

1. the product of intervals $\Pi_i[R_i, S_i]$ must be included in Z_{adm},
2. the approximating function $j_i(z_i) = \frac{C_i}{R_i - z_i} + \frac{D_i}{z_i - S_i} + A_i + B_i z_i$ is convex on the interval $[R_i, S_i]$,
3. the function j_i is a good local approximation of the cost function j, and satisfies in particular

$$
\begin{aligned}
& j_i(z_i^k) = 0, \\
& \frac{dj_i}{dz_i}(z_i^k) = \frac{\partial j}{\partial z_i}(z^k), \\
& \frac{d^2 j_i}{dz_i^2}(z_i^k) = \frac{\partial^2 j}{\partial z_i^2}(z^k).
\end{aligned}
$$

The steepest descent algorithm and variants are simple and robust techniques, but they may be slow to converge in multidimensional optimization problems. In fact, one

can prove that such algorithms have at best a linear convergence rate given by

$$\|z^k - z^*\| \approx \left[\frac{cond(D^2 j) - 1}{cond(D^2 j) + 1}\right]^k \|z^0 - z^*\|.$$

3.6 Complement: detailed description of the GMRES algorithm

3.6.1 Description

The purpose of the GMRES algorithm is to extend the conjugate gradient techniques to the solution of linear problems $A \cdot x = b$ involving nonsymmetric matrices A, while keeping the key ingredient of these conjugate gradient techniques, namely their optimality on the Krylov space. For this purpose, from an initial given residual $r^o = b - Au^o$, the algorithm constructs a sequence of vectors u^K such that

$$\begin{cases} u^K \text{ minimizes the residual } \|A \cdot u^K - A \cdot u\| \text{ on the Krylov space,} \\ W_{K-1} = \{u^o + \sum_{i=0}^{K-1} \alpha_i A^i \cdot r^o\}. \end{cases}$$

This sequence is constructed by building in parallel

- an orthonormal basis of the space $W_{K-1} - u^o$,
- the matrix H_K of A in this new basis,
- the QR factorization of this new matrix,

from which we can explicitly compute the minimizer u^K.

3.6.2 The main steps

First phase

The first step consists in building an orthonormal basis on the Krylov space $W_{K-1} - u^o$ by a Gram–Schmidt orthogonalization procedure:

$$v^1 = r^o / \|r^o\|,$$
$$v^{i+1} = \lambda_i [A \cdot v^i - \sum_{j \le i} v^j \langle A \cdot v^i, v^j \rangle], \|v^{i+1}\| = 1.$$

Second phase

The second phase computes the matrix H of A in the basis (v_i).

By construction, we have

$$\begin{aligned} A \cdot v^i &= \frac{1}{\lambda_i} v^{i+1} + \sum_{j \le i} \langle A \cdot v^i, v^j \rangle v^j \\ &:= H_{i+1,i}\, v^{i+1} + \sum_{j \le i} H_{ji} v^j. \end{aligned}$$

Therefore, the matrix H is an upper Hessenberg matrix

$$H = \begin{bmatrix} H_{1,1} & H_{1,2} & H_{1,3} & \cdot & \cdot \\ H_{2,1} & H_{2,2} & H_{2,3} & \cdot & \cdot \\ 0 & H_{3,2} & H_{3,3} & \cdot & \cdot \\ 0 & 0 & H_{4,3} & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot \end{bmatrix}.$$

In particular, we will denote by H_K the $(K+1, K)$ upper Hessenberg submatrix of H whose columns correspond to the components in the basis $(v^1, \ldots, v^{K+1})$ of the images by A of the K basis vectors $(v^1, \ldots, v^K)$ of the Krylov space $W_{K-1} - u^o$.

Third phase

The last step computes the minimizer u^K of the L^2 residual in the linear space spanned by the set $(v^1, \ldots, v^K)$. By construction, one must minimize

$$\|A \cdot v - A \cdot u\|_2$$

on the space

$$W_{K-1} = \{v, v = u^o + \sum_{i=1}^{K} y_i \, v^i, \quad y \in \mathbb{R}^K\}.$$

After development, we have

$$\begin{aligned} A \cdot u - A \cdot v &= b - A \cdot u^o - \sum_{i=1}^{K} y_i \, A \cdot v^i \\ &= \|r^o\| v^1 - \sum_{i=1}^{K} \sum_{j=1}^{i+1} H_{j,i} \, y_i \, v^j. \end{aligned}$$

Since $(v^1, \ldots, v^{K+1})$ is an orthonormal basis by construction, the norm of $A \cdot u - A \cdot v$ is equal to the sum of its components squared, that is

$$\|A \cdot u - A \cdot v\|_2^2 = (\|r^o\| - \sum_{i=1}^{K} H_{1,i} \, y_i)^2 + \sum_{j=2}^{K+1} \left(\sum_{i=j-1}^{K} H_{j,i} \, y_i \right)^2.$$

This can be rewritten

$$\|A \cdot u - A \cdot v\|_2^2 = \left\| \|r^o\| e^1 - H_K \, y \right\|_2^2,$$

where $e^1 \in \mathbb{R}^{K+1}$ denotes the unit vector $(e^1)^t = (1, 0, 0, \ldots, 0)$.

Let us now suppose that we know a QR decomposition of the matrix H_K,

$$H_K = Q_K \, R_K,$$

where Q_K is an orthogonal matrix of order $K+1$ and R_K is equal to an upper triangular matrix $\hat{R}_K$ of order K completed by a line of zeros. Using the othogonality of Q_K, we then have

$$\begin{aligned} \|A \cdot u - A \cdot v\|_2^2 &= \left\| \|r^o\| e^1 - Q_K \, R_K \, y \right\|_2^2 \\ &= \left\| Q_K^T \|r^o\| e^1 - Q_K^T \, Q_K \, R_K \, y \right\|_2^2 \\ &= \left\| Q_K^T \|r^o\| e^1 - R_K \, y \right\|_2^2. \end{aligned}$$

Minimizing this quantity with respect to v, that is minimizing the right-hand side with respect to the vector y, is now explicit. The solution is given as

$$y^* = \hat{R}_K^{-1} \hat{Q}_K^T \|r^o\| \hat{e}^1, \left(\hat{e}_K^1\right)^t = (1, 0, \ldots, 0) \in \mathbb{R}^K,$$

with $\hat{Q}_K$ denoting the $K \times K$ submatrix of Q_K, and the corresponding minimum is equal to

$$\inf_v \|A \cdot u - A \cdot v\|_2^2 = \left\| Q_K^T \|r^o\| e^1 - R_K \hat{R}_K^{-1} \hat{Q}_K^T \|r^o\| \hat{e}^1 \right\|_2^2 = \left(Q_K^T \|r^o\| e^1\right)_{K+1}^2.$$

By construction, the associated minimizer u^K in v has for expression

$$u^K = \sum_{i=1}^{K} y_i^* \, v^i + u^o$$

and corresponds to a residual (error) given by

$$res_K = (Q^T \|r^o\| e^1)_{K+1}.$$

What needs to be done in the third phase is finally to build the $Q_K R_K$ factorization $H_K = Q_K R_K$, to solve the triangular linear system

$$\hat{R}_K \, y^* = \hat{Q}_K^T \|r^o\| \hat{e}^1$$

and to set

$$u^K = u^o + \sum_{i=1}^{K} y_i^* \, v^i,$$

the corresponding error being given by

$$\|e^K\| = \left(Q_K^T \|r^o\| e^1\right)_{K+1}.$$

3.6.3 Final algorithm

For completeness, we describe below the practical implementation of the GMRES algorithm when applied to the solution of the linear system

$$A \cdot x = b \in \mathbb{R}^n.$$

Initialization

- The matrix A, the initial solution x_0 and the right-hand side b are given. We then compute
- $r^0 = b - A \cdot x^0$ = initial residual,
- $sm(1) = \|r^0\|$,
- $v_1 = r^0 / \|r^0\|$ = first basis vector.

Loop on j

(extension of the Krylov space and updating of the projection H of A on the Krylov space)

- **Computing the $j+1$ basis vector**
  - $w^j = A \cdot v^j$ = image of the j basis vector,
  - $H_{ij} = \langle w^j, v^i \rangle, \forall i \leq j$,
  - $w^j \longleftarrow w^j - \sum_{i \leq j} H_{ij} v^i$ = vector w^j after orthogonalization,
  - $H_{j+1,j} = \|w^j\|$,
  - $v^{j+1} = w^j / \|w^j\|$.
- **QR factorization of the matrix A in basis v^j**
  - For i increasing from 1 to $j-1$, do
$$\begin{pmatrix} H_{ij} \\ H_{i+1,j} \end{pmatrix} \longleftarrow \begin{pmatrix} c_i & -s_i \\ s_i & c_i \end{pmatrix} \begin{pmatrix} H_{ij} \\ H_{i+1,j} \end{pmatrix},$$
  - $r = H_{jj}$,
  - $h = H_{j+1,j}$,
  - $c_j = r/(r^2+h^2)^{1/2}$,
  - $s_j = -h/(r^2+h^2)^{1/2}$,
  - $H_{jj} \longleftarrow c_j r - s_j h$.
- **Multiplication of the right-hand side by Q**
  - $sm(j+1) = s_j sm(j)$;
  - $sm(j) \longleftarrow c_j sm(j)$;
  - If $\|sm(j+1)\| < \varepsilon$, the Krylov space is sufficiently large, and we compute the solution x by solving the triangular system $R \cdot y = sm$ (where R has been stored in the upper part of the matrix H), and by setting $x = x^j = x^0 + \sum_{i \leq j} y_i v^i$.
  - If not, we set $j = j+1$ and we go back to the first step of the loop in j.

End of GMRES algorithm

3.6.4 Convergence analysis

The convergence properties of the GMRES algorithm can be summarized in the next theorems [11]

Theorem 3.3 *If A can be diagonalized into $A = XDX^{-1}$, and if x^j denotes the solution computed at step j of the above GMRES algorithm, then we have*

$$\|b - A \cdot x^j\| \leq \|X\| \|X^{-1}\| \varepsilon^{(j)} \|b - A \cdot x_0\|$$

under the notation

$$\varepsilon^{(j)} = \min_{P \in P_j, P(0)=1} \left\{ \max_{i \leq j} \|P(d_{ii})\| \right\},$$

the minimization being done on the set P_j of real polynomials of degree less than or equal to j.

Theorem 3.4 *If A is definite, positive, with symmetric part M, then we have in particular*

$$\|b - A \cdot x_j\| \leq \left\{1 - \frac{\lambda_{min}(M)^2}{\lambda_{max}(A^t A)}\right\}^{j/2} \|b - A \cdot x_0\|.$$

3.7 Exercises

The exercises below should help the reader to become familiar with Newton's algorithm.

Square root extraction

Apply Newton's algorithm to the solution of the algebraic problem $z^2 = a$. What is its speed of convergence ?

Division

Using Newton's algorithm, compute $z = \frac{1}{a}$ without performing a single division. (Hint: use the function $F(z) = \frac{1}{z} - a$.)

Chapter 4

Constrained Optimization

4.1 Optimality conditions

In general, the design variables $z \in \mathbb{R}^n$ in the shape optimization problem (P_h) must also satisfy several inequality constraints of the type $g_l(z) \leq 0, \forall l = 1, m$. A general shape optimization problem takes then the form of the following constrained minimization problem

$$\boxed{\min_{\substack{z \in \mathbb{R}^n \\ h_i(z)=0, g_l(z) \leq 0}} j(z).} \tag{4.1}$$

The cost function j and the design constraints h_i or g_l are again given as functions of the computational grid $\underline{X}(z)$ and of the state $W_h(z)$

$$\begin{aligned} j(z) &= J(\underline{X}(z), W_h(z)), \\ h_i(z) &= h_i(\underline{X}(z), W_h(z)), \\ g_l(z) &= g_l(\underline{X}(z), W_h(z)), \end{aligned}$$

which are obtained by solving the state equation

$$(E_h) \left\{ \begin{array}{l} f_h(\underline{X}, W_h) = 0 \quad \text{in} \quad \mathbb{R}^p, \\ \underline{X} = \underline{X}(z). \end{array} \right.$$

As before, this chapter will treat the variables $(\underline{X}(z), W_h(z))$ as dependent variables, that is as intermediate functions of z, implying that the cost function j and constraints h_i or g_l are only functions of z and that the state equation is just an intermediate step needed for the evaluation of these functions.

In this framework, the necessary optimality conditions of Kuhn–Tucker introduced in Chapter 2 and studied at length in [2, 48] take the classical form

$$\nabla j(z^*) + \sum_{i=1}^{q} \mu_i^* \nabla h_i(z^*) + \sum_{l=1}^{m} \lambda_l^* \nabla g_l(z^*) = 0 \quad \text{in} \quad \mathbb{R}^n, \tag{4.2}$$

$$h_i(z^*) = 0, \forall i = 1, q, \tag{4.3}$$

$$\lambda_l^* g_l(z^*) = 0, \forall l = 1, m. \tag{4.4}$$

E. Laporte et al., *Numerical Methods in Sensitivity Analysis and Shape Optimization*

The optimum pair (z^*, μ^*, λ^*), composed of the minimizer z^* and of the Lagrange multipliers (μ^*, λ^*) must also satisfy the admissibility condition

$$(z^*, \mu^*, \lambda^*) \in C = \{(z, \mu, \lambda) \in \mathbb{R}^n \times \mathbb{R}^q \times \mathbb{R}^m, g(z) \leq 0, \lambda \geq 0\}.$$

The purpose of this chapter is to propose a practical numerical strategy for the solution of problem (4.1) by interior point techniques.

4.2 Interior point algorithms

4.2.1 Introduction

Interior point methods turn out to be quite adequate for solving the above optimality conditions. On the one hand, they are natural efficient generalizations of Newton's method in the presence of inequality constraints. On the other hand, forcing the whole optimization process to satisfy the design constraints forbids unfeasible intermediate designs which would be impossible to compute.

There is nowadays a vast literature on interior point methods. We refer to Strang [48] for a first introduction, to Herskovits [21] for the introduction of general nonlinear algorithms, to Tits and coauthors [41, 50] for its development in the case of linear constraints, to Potra [44] for the study of predictor-corrector versions of such algorithms, or to El Bakry and al. [12] for a general formulation of these methods in a primal dual framework.

4.2.2 Basic philosophy

The traditional idea in constrained optimization is to eliminate either the multiplier or the inequality constraint since one of them has to be zero at the optimum. More precisely, as done in Chapter 2, we could introduce the set of active constraints

$$I^* = I(z^*) = \{l \in \{1, m\} \text{ such that } g_l(z^*) = 0\}$$

and the active surface of constraints

$$A_I = \{z \in \mathbb{R}^n, h_i(z) = 0, \forall i = 1, q, g_l(z) = 0, \forall l \in I(z^*)\},$$

and observe that any local minimizer z^* of j on A is also a local minimizer of j over A_I. The latter problem is a minimization problem which only involves equality constraints and which can therefore be solved by the Newton's algorithms of the previous chapter. Such a strategy looks attractive, but is not efficient in practice because the identification of the set I^* of active constraints is difficult. Different updates of the set I are then necessary (one must add the l constraint to I if the constraint $g_l(z) \leq 0$ is violated by the minimizer z_I^* of j on A_I, and one must remove l from I if the corresponding multiplier $\lambda_l^*(z_I^*)$ is negative), and this induces a lot of spurious oscillations of the overall algorithm.

Interior point algorithms take a completely different approach. The first main idea of interior point algorithms is to use a primal dual approach, that is to keep both the Lagrange multipliers and the constraints in the equations to be solved. This means in particular that the quadratic equations

$$\lambda_l^* \cdot g_l(z^*) = 0, \forall l = 1, m$$

are kept in the optimality system. The second main idea is to solve this full optimality system by a Newton algorithm, which is modified in order to impose that at each iteration the pairs $(z, \mu, \lambda)^k$ stay strictly inside the admissibility domain C. This has three main advantages:

1. the algorithm keeps the quadratic convergence properties of a Newton method,
2. the iterates $(z, \lambda)^k$ stay away from the boundary of C, which avoids most problems related to corners, coefficient discontinuities or singularities in the matrix of the linearized Newton system,
3. the solution z^k at each iteration is admissible, which avoids unphysical situations in the convergence process and makes it possible to successfully stop the algorithm well before full convergence.

4.2.3 A simplified interior point algorithm

For simplicity, we first suppose that the cost function j is quadratic

$$j(z) = \frac{1}{2} z \cdot H \cdot z + c \cdot z,$$

that there are no equality constraints ($q = 0$), and that the inequality constraints are linear and are given by

$$g(z) = -z \leq 0.$$

In this framework, as proposed and analyzed in Tits and Zhou [50], based on a preliminary work of Herskovits [21], we can construct a very simple interior point algorithm in two steps.

1. The first step is a standard Newtonian prediction computing the direction of descent $(dz, d\lambda)$ by solving the linearized system in dz and $d\lambda$,

$$H \cdot dz - d\lambda = -H \cdot z^k - c + \lambda^k, \tag{4.5}$$
$$\lambda_j^k dz_j + z_j^k d\lambda_j = -\lambda_j^k z_j^k, \forall j = 1, n. \tag{4.6}$$

   By elimination of $d\lambda_j$ by $d\lambda_j = -\lambda_j^k (z_j^k + dz_j)/z_j^k$, this system reduces to a single problem

$$K \cdot dz = -H \cdot z^k - c$$

   in dz with penalized matrix $K_{ij} = H_{ij} + \frac{\lambda_j^k}{z_j^k} \delta_{ij}$.

2. The second step updates the solution (z, λ) by

$$z^{k+1} = z^k + dz + (t-1)dz,$$
$$\lambda_j^{k+1} = \lambda_j^k + d\lambda_j + \varepsilon_j.$$

This update adds small corrections $(t-1)dz$ and ε_j to the Newton update $(dz, d\lambda)$ in order to preserve the strict admissibility of the solution pair (z, λ), imposing in fact

$$-z^{k+1} \leq -O(\|dz\|) < 0, \quad \lambda^{k+1} \geq O(\|dz\|^2) > 0.$$

In particular, the correction on dz imposes that the update $z^k + dz + (t-1)dz$ be strictly positive, which is achieved by choosing a real number $0 < \beta < 1$ fixed once for all , and by setting

$$\begin{aligned} \bar{t} &= \min_{dz_j<0}(-z_j^k/dz_j) \\ &= \text{maximal step to have } z^k + tdz \geq 0, \\ t &= \min(\max(\beta\bar{t}, \bar{t} - \|dz\|), 1) \\ &= \text{largest step compatible with the Newton update and preserving} \\ &\quad \text{the strict admissibility of } z^k + tdz. \end{aligned}$$

The correction on λ_j is mainly given by

$$\varepsilon_j = \max(\varepsilon\|dz\|^2 - \lambda_j^k - d\lambda_j, 0).$$

It truncates the multiplier λ_j from below by the positive number $\varepsilon\|dz\|^2$. More precisely, when one tends towards the saturation of the constraint (which activates the multiplier $(\lambda + d\lambda)_i > \varepsilon\|dz\|^2$), the value predicted by the Newton algorithm is kept (it is also possible to bound this value from above by introducing a maximum threshold $\lambda_j^{k+1} = \min(\lambda_{max}, (\lambda^k + d\lambda)_j)$). On the other hand, if the constraint is relaxed, the multiplier is also relaxed and is set to the minimal value $\varepsilon\|dz\|^2$. A particular case arises when one stays very close to the constraint $(\lambda_j^k + d\lambda_j \leq -z_j < 0)$ where it would be adviseable to replace this value $\varepsilon\|dz\|^2$ by a reference initial value λ_j^o. In any case, the update of λ only affects the construction of the penalty matrix K of the next Newton iteration, and the real important trick is in fact to bound the product $\lambda_j z_j$ both from below and from above.

With this line search procedure, the above Newton algorithm is transformed into an interior point algorithm since all updates are kept strictly inside C by construction. It is rather robust and converges quadratically as proved in Tits et Zhou [50] using the convergence theorem of the previous chapter.

4.3 Interior point algorithms with deflection

4.3.1 Introduction

For general convex constraint functions g_l, the direction dz of update proposed by the Newton step may point to the outside of the admissible domain C,

meaning that we will have $g_l(z + tdz) > 0$ for almost all positive values of t (Figure 4.1) . It is then impossible to find any admissible point $(z^k + tdz, \lambda)$ on the line of research, and the line search automatically fails. Moreover, a global analysis of convergence indicates that the most efficient algorithms are obtained when the different updates (z^k, λ^k) approach all active boundaries of C at the same speed, that is when all products $\lambda_l^k g_l(z^k)$ stay close to a given average value $\omega_k \approx \|dz\|$. Geometrically speaking, this means that the iterates must stay close to the central axis of C, leaving more admissible space for the next line search. The necessity of having admissible search directions combined with the interest of having iterates close to the center of C make it useful to modify the Newton direction of search in order to deflect it towards the center of C.

After adding this deflection, an interior point iteration has three main steps. The **first step** is a (quasi-Newton) iteration which solves the **full linearized optimality system**

$$
\begin{aligned}
&B \cdot d + \sum_{i=1}^{q} \mu_i \nabla h_i(z^k) + \sum_{l=1}^{m} \lambda_l \nabla g_l(z^k) = -\nabla j(z^k), \\
&\nabla h_i(z^k) \cdot d = -h_i(z^k), \forall i = 1, q, \\
&\lambda_l^k \nabla g_l(z^k) \cdot d + \lambda_l g_l(z^k) = 0, \forall l = 1, m.
\end{aligned}
$$

The unknowns are the update d of the unknown z and the updated value (μ, λ) of the Lagrange multipliers. The matrix B introduced above is an approximation of the Hessian

$$
\nabla^2 j(z^k) + \sum_{i=1}^{q} \mu_i^k \nabla^2 h_i(z^k) + \sum_{l=1}^{m} \lambda_l^k \nabla^2 g_l(z^k).
$$

In this first step, as observed earlier, the tentative descent direction d may point to the outside of the admissible domain C. The **second step** is thus a deflection step which modifies this direction d in order to build an updated direction $d + \rho\tilde{d}$ that is a strict direction of descent for the cost j but which points towards the inside of domain C. This update is obtained by solving the same Newton system as above, but with a new right-hand side made of m strictly positive numbers $(\omega_l)_{l=1,m}$,

$$
\begin{aligned}
&B \cdot \tilde{d} + \sum_{i=1}^{q} \tilde{\mu}_i \nabla h_i(z^k) + \sum_{l=1}^{m} \tilde{\lambda}_l \nabla g_l(z^k) = 0, \\
&\nabla h_i(z^k) \cdot \tilde{d} = 0, \forall i = 1, q, \\
&\frac{d(\lambda_l g_l)}{d(z, \lambda)} \cdot (\tilde{d}, \tilde{\lambda}) := \lambda_l^k \nabla g_l(z^k) \cdot \tilde{d} + \tilde{\lambda}_l g_l(z^k) = -\omega_l, \forall l = 1, m.
\end{aligned}
$$

By construction, the proposed right-hand side forces the products $\lambda_l g_l$ towards negative values. In particular, assuming for simplicity that we have no equality constraints and eliminating $\tilde{\lambda}$ from the last equations, the solution of the above

linear system reduces to the algebraic system in $\tilde{d}$,

$$\left(B - \sum_l \frac{\lambda_l^k}{g_l(z^k)} \nabla g_l \otimes \nabla g_l\right) \cdot \tilde{d} = \sum_l \frac{\omega_l}{g_l(z^k)} \nabla g_l.$$

By construction, the coefficients $\frac{\omega_l}{g_l(z^k)}$ are negative, and the matrix

$$\left(B - \sum_l \frac{\lambda_l^k}{g_l(z^k)} \nabla g_l \otimes \nabla g_l\right)$$

is positive. This means that the direction of deflection $\tilde{d}$ looks like a linear combination of the constraints gradients with strictly negative coefficients (Figure 4.1), and therefore points towards the interior of C.

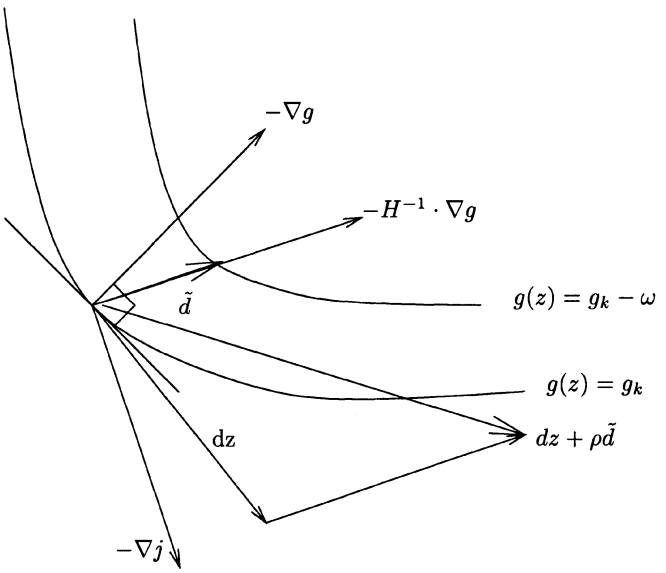


Figure 4.1. Unfeasibility of the Newton's direction of descent dz for convex constraints and construction of a deflected direction

The **last step** is finally a standard line search which minimizes the Han penalized cost function j_c on the direction of descent $z^k + t(d + \rho\tilde{d}) \in C$,

$$\min_{\substack{t \in \mathbb{R} \\ z^k + t(d+\rho\tilde{d}) \in C}} j_c(z^k + t(d + \rho\tilde{d}))$$

and updates the Lagrange multiplier λ^k.

We will now describe different adaptations of this basic algorithm, and for simplicity, we will first restrict ourselves to situations which do not involve any equality constraints ($q = 0$).

Remark 4.1 At each step, the constraint $g_l(z^k)$ is strictly negative by construction. Therefore, the multipliers λ_l can be eliminated from the linearized optimality system. Indeed, after division by $g_l(z^{(k)})$, the last equations in this system take the form

$$\lambda_l = -\frac{\lambda_l^k}{g_l(z^k)} \nabla g_l(z^k) \cdot d.$$

Assuming for simplicity that there are no equality constraints ($q = 0$), the linearized system can then be reduced to a linear system $K \cdot d = -\nabla j_\varepsilon(z^k)$, whose matrix

$$\begin{aligned} K &= \nabla^2 \left[j(z) - \sum_l \epsilon_l^k \log(-g_l(z)) \right] \\ &= \left[\nabla^2 j(z^k) + \sum_l \lambda_l^k \nabla^2 g_l(z^k) \right] - \sum_l \frac{\lambda_l^k}{g_l(z^k)} \nabla g_l(z^k) \otimes \nabla g_l(z^k), \end{aligned}$$

is associated to the penalized function $j_\varepsilon(z) = j(z) - \sum_l \epsilon_l^k \log(-g_l(z))$ and to the penalization factors $\epsilon_l^k = -\lambda_l^{(k)} g_l(z^{(k)}) > 0$. In other words, as often observed in the literature on interior point algorithms, we get the same system as if we had introduced a logarithmic penalization of the inequality constraint $g_l(z^{(k)}) \leq 0$.

4.3.2 Central trajectories

The central trajectory is a curve of C which is of particular interest in interior point methods because it stays away from the boundary of C and because it contains the optimal solution at its limit. By definition, the central trajectory is the curve C_o defined in C by

$$\begin{aligned} C_o = \{(z,\lambda), g(z) < 0, \lambda > 0, \nabla j(z) + \sum_{l=1}^{m} \lambda_l \nabla g_l(z) = 0, \\ \exists \omega_0 > 0 \in \mathbb{R}, \text{ such that } g_l(z)\lambda_l = -\omega_0, \forall l = 1, m\}. \end{aligned}$$

A central trajectory algorithm uses path following techniques to follow the curve C_o from the center of C to the boundary $\omega_0 = 0$ where the solution lies. The deflection technique is used here to impose that the intermediate solutions (z^K, λ^K) of the nonlinear problem remain on the central trajectory during the algorithm, leading to the following algorithm.

- $\omega_0^K > 0$ given in $\mathbb{R}$ and (z^{K-1}, λ^{K-1}) known in C.
- Using a Newton algorithm initialized by (z^{K-1}, λ^{K-1}) compute the solution $(z^K, \lambda^K) \in C$ of the optimality system

$$\begin{aligned} \nabla j(z^K) + \sum_{l=1}^{m} \lambda_l^K \nabla g_l(z^K) = 0, \\ \lambda_l^K g_l(z^K) = -\omega_0^K, \forall l = 1, m. \end{aligned}$$

After elimination of λ_l^K, this step amounts to the calculation of the minimum z^K of the logarithm barrier penalized function

$$z^K = \text{Argmin}_{z\in\mathbb{R}^n}\left(j(z) - \sum_{l=1}^{m} \omega_0^K \ln(-g_l(z))\right).$$

- Set $\omega_0^{K+1} = \alpha\omega_0^K$, $K = K+1$ and reiterate.

The above penalization strategy using logarithmic barrier functions has been known and used for a long time. The resulting algorithm is very simple, and rather efficient. The problem is to properly control the updating strategy of the regularizing factor ω_0, in order to make it smoothly converge to zero. Many variants have been recently proposed to overcome this problem (see [5] and the citations therein), most of them keeping the Lagrange multipliers λ^K as independent variables (primal dual approach).

4.3.3 Centered trajectory algorithms

The central trajectory may be difficult to follow. But it can be replaced by larger α centered sets $\mathcal{N}_\alpha$, still staying away from the boundary of C and still containing the solution at the limit. These sets are the cones of C defined by

$$\mathcal{N}_\alpha = \{(z,\lambda) \in C, |g_l(z)\lambda_l - \frac{g(z)\cdot\lambda}{m}| \leq -\alpha\frac{g(z)\cdot\lambda}{m}, \forall l = 1,m\}.$$

Centered trajectories algorithms solve then the optimality conditions

$$\nabla j(z^*) + \sum_{l=1}^{m} \lambda_l^* \nabla g_l(z^*) = 0,$$
$$g(z^*)\lambda^* = 0, (\text{ i.e., } g_l(z^*)\lambda_l^* = 0, \forall l = 1,m)$$

by a Newton algorithm whose iterates are forced to stay within the centered cone $\mathcal{N}_\alpha$. This variant has been developed in the OB1 software from the National Institute of Standards and Technology, and analysed by Potra [44]. Its steps are:

- (z^k, λ^k) known in C;
- compute the direction of descent dz and $d\lambda$ by performing a Newton iteration

$$(\nabla^2 j + \sum_{l=1}^{m} \lambda_l^k \nabla^2 g_l)(z^k)\cdot dz + \sum_{l=1}^{m}(\lambda^k + d\lambda)_l \nabla g_l(z^k) = -\nabla j(z^k),$$
$$\lambda_l^k \nabla g_l(z^k)\cdot dz + (\lambda_l^k + d\lambda_l) g_l(z^k) = 0, \forall l = 1,m.$$

- find the largest possible relaxation parameter $\bar{\theta} \in [0,1]$ so that

$$(\bar{z}, \bar{\lambda}) = (z^k + \bar{\theta} dz, \lambda^k + \bar{\theta} d\lambda) \in \mathcal{N}_{1/2};$$

- correct $(\bar{z}, \bar{\lambda})$ by a deflection vector $(\tilde{d}, \tilde{\lambda})$ satisfying (centering step)

$$\left(\nabla^2 j + \sum_{l=1}^{m} \lambda_l^k \nabla^2 g_l\right)(z^k) \cdot \tilde{d} + \sum_{l=1}^{m} \tilde{\lambda}_l \nabla g_l(z^k) = 0 \qquad \text{(no influence on } dj\text{)},$$

$$(\bar{z} + \tilde{d}, \bar{\lambda} + \tilde{\lambda}) \in \mathcal{N}_{1/4} \qquad \text{(centering)},$$

$$g(\bar{z} + \tilde{z}) \cdot (\bar{\lambda} + \tilde{\lambda}) \simeq (1 - \bar{\theta}) g(z^k) \cdot \lambda^k \qquad \text{(path following towards the solution)}.$$

The difficulty in the above algorithm is the centering step which requires a certain know-how.

4.3.4 Herskovits' algorithm without equality constraints

The general idea of Herskovits is to build a unique direction of descent by combining the Newton direction of descent and the direction of deflection. It can be also further generalized by replacing in the Newton step the Hessian $\frac{d^2 j}{dz^2} + \sum_l \lambda_l \frac{d^2 g}{dz^2}$ by an adequate positive definite approximation B constructed for example by a BFGS updating procedure. The general Herskovits algorithm [23, 24] then updates the solution (z, λ) through the following steps:

i) **Newton's prediction**

Solve the linearized equations of optimality

$$B \cdot dz + \sum_{l=1}^{m} d\lambda_l \nabla g_l(z^k) = -\frac{\partial \mathcal{L}}{\partial z}(z^k, \lambda^k) = -\nabla j\left(z^k\right) - \sum_{l=1}^{m} \lambda_l^k \nabla g_l\left(z^k\right),$$

$$\lambda_l^k \nabla g_l(z^k) \cdot dz + d\lambda_l g_l(z^k) = -\lambda_l^k g_l(z^k),\ l = 1, m.$$

ii) **deflection**

In order for the direction of descent to point towards the inside of the admissible set C, positive numbers $\omega_l > 0$ are chosen and a *normal* correction $(\tilde{d}, \tilde{\lambda})$ is constructed by solving

$$B \cdot \tilde{d} + \sum_{l=1}^{m} \tilde{\lambda}_l \nabla g_l(z^k) = 0,$$

$$\lambda_l^k \nabla g_l(z^k) \cdot \tilde{d} + \tilde{\lambda}_l g_l(z^k) = -\omega_l \lambda_l^k,\ l = 1, m.$$

From the first equation , the Kuhn–Tucker residual is not affected by $(\tilde{d}, \tilde{\lambda})$. For an active constraint $(g_l(z^k) \approx 0)$, the second equation implies

$$\nabla g_l(z^k) \cdot \tilde{d} = -\omega_l < 0$$

and thus descending along $\tilde{d}$ makes the constraint g_l decrease and bends the solution towards the interior of domain C. For an inactive constraint $(\lambda_l^k \approx 0)$, the second equation implies $\tilde{\lambda}_l = 0$, which means that the inactive constraint g_l has no influence on the deflection $\tilde{d}$.

iii) **Direction of descent**

The direction of descent $(dz, d\lambda)$ is finally given by combining the Newton direction of descent and the above direction of deflection

$$(\bar{dz}, \bar{d\lambda}) = (dz, d\lambda) + \rho(\tilde{d}, \tilde{\lambda})$$

where ρ is the largest possible number preserving a significant decrease of the cost function j along the direction $(\bar{dz}, \bar{d\lambda})$ and is bounded above by $\alpha\|dz\|^2$ in order to preserve the Newtonian quadratic speed of convergence next to the solution. It is obtained by solving the one-dimensional minimization problem

$$\begin{aligned} &\max \rho, \\ &0 < \rho \leq \alpha\|dz\|^2, \\ &\nabla j \cdot (dz + \rho\tilde{d}) \leq \frac{1}{2}\nabla j \cdot dz. \end{aligned} \tag{4.7}$$

iv) **Line search**

The solution (z, λ) is finally updated by the formula

$$\begin{aligned} z^{k+1} &= z^k + t\bar{dz}, \\ \lambda_l^{k+1} &= \lambda_l^k + \bar{d\lambda}_l + \varepsilon_l, \end{aligned}$$

with t an approximate solution of the one-dimensional problem

$$\begin{aligned} &\min_{t>0} j(z^k + t\bar{dz}) \text{ with} \\ &g_l(z^k + t\bar{dz}) < 0 \quad \text{if} \quad \lambda_l^k + \bar{d\lambda}_l \geq 0, \\ &g_l(z^k + t\bar{dz}) \leq g_l(z^k) \quad \text{if not,} \end{aligned}$$

and with ε_l defined as in the previous section by $\varepsilon_l = \max\left(\varepsilon\,\|dz\|^2 - \lambda_l^k - \bar{d\lambda}_l, 0\right)$. This line search guarantees that our algorithm is indeed an interior point algorithm since the above update stays by construction strictly inside the admissible domain C. The inequality $g_l(z^k + t\bar{dz}) \leq g_l(z^k)$ if $\lambda_l^k + \bar{d\lambda}_l < 0$ forbids the saturation of constraints which would correspond to negative values of the dual parameters λ and avoids converging towards unfeasible solutions of the optimality conditions.

In practice, the above one-dimensional problem is usually solved by an Armijo technique computing the first element in the sequence $\{1, \nu, \nu^2, \nu^3, \ldots\}$ such that

$$
\begin{aligned}
& j(z^k + t\bar{dz}) \leq j(z^k) + \eta t \nabla j(z^k) \cdot \bar{dz}, \\
& g_l(z^k + t\bar{dz}) < 0 \quad \text{if} \quad \lambda_l^k + \bar{d\lambda}_l \geq 0, \\
& g_l(z^k + t\bar{dz}) \leq g_l(z^k) \text{ if not.}
\end{aligned}
$$

Such a t should exist because by construction, $\bar{dz}$ is a direction of descent both for the cost function and for the constraint g_l.

Next to the solution, the Newton prediction should be accurate, and the values of the coefficients t are expected to be close to 1 at convergence. In fact, a value close to 1 guarantees a quadratic convergence of the algorithm by a direct application of the general convergence theorem of the previous chapter. Such a behavior is usually respected in practice. The detailed implementation of the above algorithm is described in [24].

4.3.5 Introduction of equality constraints

Introducing equality constraints does not modify the philosophy of the above Herskovits algorithm, but adds difficulties to the line search. With equality constraints, the problem to solve becomes

$$
\begin{aligned}
& \text{Find } z^* \in A \text{ such that} \\
& j(z^*) \leq j(z), \forall z \in A,
\end{aligned}
$$

where the set A of admissible designs is now given by

$$
A = \{z \in \mathbb{R}^n, h_i(z) = 0, \forall i = 1, q, g_l(z) \leq 0, \forall l = 1, m\}.
$$

The solution again must satisfy the (Kuhn–Tucker) optimality conditions of Chapter 2,

$$
\begin{aligned}
& \nabla j(z^*) + \sum_{i=1}^{q} \mu_i \nabla h_i(z^*) + \sum_{l=1}^{m} \lambda_l \nabla g_l(z^*) = 0, \\
& h_i(z^*) = 0, \forall i = 1, q, \\
& \lambda_l g_l(z^*) = 0, \forall l = 1, m, \\
& \lambda_l \geq 0, g_l(z^*) \leq 0, \forall l = 1, m.
\end{aligned}
$$

These equations are very similar to the equations solved in the previous section, which means that the interior point algorithm can be extended to this new case. The major changes concern the Newton prediction step in which the first equation in dz must be replaced by a system with unknown $(dz, d\mu)$

yielding

$$B \cdot dz + \sum_{i=1}^{q} d\mu_i \nabla h_i(z^k) + \sum_{l=1}^{m} d\lambda_l \nabla g_l(z^k) = -\frac{\partial \mathcal{L}}{\partial z}(z^k, \mu^k, \lambda^k)$$
$$= -\nabla j\left(z^k\right) - \sum_{i=1}^{q} \mu_i^k \nabla h_i\left(z^k\right) - \sum_{l=1}^{m} \lambda_l^k \nabla g_l\left(z^k\right),$$
$$\nabla h_i(z^k) \cdot dz = -h_i(z^k), \forall i = 1, q,$$
$$\lambda_l^k \nabla g_l(z^k) \cdot dz + d\lambda_l g_l(z^k) = -\lambda_l^k g_l(z^k),\ \forall l = 1, m,$$

the deflection step which is now given by

$$B \cdot \tilde{d} + \sum_{i=1}^{q} \tilde{\mu}_i \nabla h_i(z^k) + \sum_{l=1}^{m} \tilde{\lambda}_l \nabla g_l(z^k) = 0,$$
$$\nabla h_i(z^k) \cdot \tilde{d} = 0, \forall i = 1, q,$$
$$\lambda_l^k \nabla g_l(z^k) \cdot \tilde{d} + \tilde{\lambda}_l g_l(z^k) = -\omega_l \lambda_l^k,\ \forall l = 1, m,$$

and the line search where the orginal search function j must be replaced by a Han penalized function $j_c(z) = j(z) + \sum_{i=1}^{q} c_i |h_i(z)|$. In the above algorithm, the matrix B is again a positive definite approximation of the Hessian $\nabla^2 \mathcal{L}(z^k, \mu^k, \lambda^k)$, with $\mathcal{L}(z, \mu, \lambda) = j\left(z\right) + \sum_{i=1}^{q} \mu_i h_i\left(z\right) + \sum_{l=1}^{m} \lambda_l g_l\left(z\right)$.

4.4 Using an interior point algorithm

4.4.1 Code's description

Global Structure

A typical computer programme implementing the above interior point algorithm [23, 24] has the following structure:

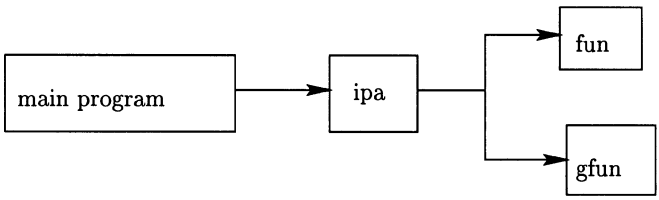


Figure 4.2. Global structure of the programme

`ipa` (meaning Interior Point Algorithm) is the main programme applying the algorithm to the cost functions and constraints defined by the user in `fun` and `gfun` (see the following explanations). The user only modifies the programme `ipa` when he wishes to change certain execution parameters.

Key Subroutines

```
ipa     - main subroutine implementing the Herskovits's
               interior point algorithm in nonlinear programming.
vbox    - check box constraints on input data
dire    - computes the  direction of descent
step0   - defines the initial step in the line search
newstep - checks the convergence criteria of the line search,
                and if needed proposes a new step
updateb - updates B by a Quasi Newton procedure
updlam  - updates the dual variables lambda
```

Integer input variables

```
nvar    - number of parameters
ncstr   - number of constraints
neq     - total number of equality constraints
neqlin  - number of linear equality constraints
                (neqlin.le.neq !!!)
lenvlb  - numb. of variables bounded from below (box constraints)
lenvub  - numb. of variables bounded from above (box constraints)
```

Calculated integer variables

```
ntcstr  = ncstr+lenvlb+lenvub  total number of constraints
nineq   = ntcstr-neq total number of inequality constraints
nlmat   = ntcstr+nvar total number of unknowns
```

Input data on the box constraints

```
lvlb(i=1:nvar) = 1: x(i) is bounded from below
lvlb(i=1:nvar) = 0: x(i) is not bounded from below
vlb(i=1:nvar) is the lower bound on x(i)
lvub(i=1:nvar) = 1: x(i) is bounded from above
lvub(i=1:nvar) = 0: x(i) is not bounded from above
vub(i=1:nvar) is the upper bound on x(i)
```

Input/output data

```
double precision x(nvar): input/ initial value of the parameters,
                          output/ optimal value of the parameters
double precision f: output/ final value of the cost function
double precision g(ncstr): output/ final value of the constraint
                                   vector
double precision lambda0(nineq):
          output/ final value of the Lagrange multipliers
                     for the inequality constraints
double precision mu0(neq):
```

```
output / final value of the Lagrange multipliers
              for the equality constraints
```

Stopping criteria

The stopping criteria are one of the following:

- the update dz =d0 of z is too small: d0**2<epsd02;
- the variation of the cost function in the three last iterations is less than epsdf;
- the square value of the Lagrangian residual is less than epsglag;
- the number of iterations is equal to itermax.

Linear search

```
ilin = 1  - the search uses Wolfe's technique with cubic
            interpolation
ilin = 0  - the search uses Armijo's technique
```

Stopping criteria for the linear search

The algorithm stops with an error message when the number of iterations in the linear search reaches itlinmax..

User's defined subroutines for the cost function and constraints

- subroutine computing cost and constraints values:

```
subroutine fun(f,g,x,nprov,rutil,iutil,lrutil,liutil)

external subroutine  fun.f  - computes f and g


double precision f        - cost function
double precision g(ncstr) - constraints vector:

g(1),...,g(neqlin)     - linear equality constraints
g(neqlin+1),...,g(neq) - nonlinear equality constraints
g(neq+1),...,g(ncstr)  - inequality constraints
                                    (except box constraints)
nprov         - control parameter exchanged between the main
                programme and the subroutines fun and gfun
iutil(liutil) - integer working array to be used if needed
                in the subroutines ipa, fun or gfun
rutil(lrutil) - double precision real working array to be
                used if needed in the subroutines ipa, fun
                or gfun
```

- subroutine computing gradients:

```
subroutine gfun(gradf,gradg,x,nprov,rutil,iutil,lrutil,
                liutil)

external subroutine gfun.f
                          - computes the gradients of f and g

double precision gradf(nvar) is the gradient of f
double precision dg(nvar,ncstr) is the  gradient of g

nprov          - control parameter exchanged between the main
                 programme and the subroutines fun and gfun
iutil(liutil) - integer working array to be used if needed
                   in the subroutines ipa, fun or gfun
rutil(lrutil) - double precision real working array to be
                 used if needed in the subroutines ipa, fun
                 or gfun
```

4.4.2 Example

The subroutines in Appendix B correspond to the following example: find $x^* = (x_1^*, x_2^*, x_3^*) \in \mathbb{R}^3$ such that

$$f(x^*) = \min_{x \in V} f(x)$$

and

$$g_1(x^*) = 0, \quad g_2(x^*) \leq 0,$$

where the space V, cost function f and constraints g are given by

$$V = \left\{(x_1, x_2, x_3) \in \mathbb{R}^3;\, x_1 \geq 0, x_2 \geq 0, x_3 \geq 0\right\},$$

$$f(x_1, x_2, x_3) = (x_1 + 3x_2 + x_3)^2 + 4(x_1 - x_2)^2,$$

$$g_1(x_1, x_2, x_3) = x_1 + x_2 + x_3 - 1,$$

$$g_2(x_1, x_2, x_3) = x_1^3 - 6x_2 - 4x_3 + 3.$$

4.4.3 Application to optimum design problems

The implementation of the above algorithm in an optimal design problem amounts to the development of the two subroutines `fun` (computing the cost function function $j(z) = J(\underline{X}(z), W_h(z))$ and the constraints $g_l(z)$) and `gfun` (computing the gradients).

By construction of the cost function, the structure of the subroutine `fun` is quite simple and corresponds to the following sequence of operations.

1. Calculation of the grid deformation $\underline{X} = \underline{X}(z)$.

2. Call of the numerical solver of the state equation $f_h(\underline{X}, W) = 0$.

3. Call of the computer programme computing the cost function

$$j(z) = J(\underline{X}, W).$$

4. Call of the computer programme computing the design constraints

$$g_l(z) = g_l(z, \underline{X}, W).$$

The subroutine `fun` therefore only chains existing solvers. Observe nevertheless that one call to `fun` executes one solution of the state equation $f_h(\underline{X}, W) = 0$, which might therefore be quite expensive.

The calculation of the gradients $\frac{dj}{dz}$ and $\frac{dg_l}{dz}$ is more elaborate, and will be explained in the following chapters. The flowchart of `gfun` will be in fact given at the end of Chapter 6.

4.4.4 Exercises

A first problem with polynomials

Let us consider the following problem: find $x^* = (x_1^*, x_2^*, x_3^*) \in \mathbb{R}^3$ such that

$$f(x^*) = \min_{x \in V} f(x)$$

with

$$g_1(x^*) \leq 0, \quad g_2(x^*) \leq 0,$$

under the notation

$$V = \left\{(x_1, x_2, x_3) \in \mathbb{R}^3;\ 0 \leq x_1 \leq 20, 0 \leq x_2 \leq 11, 0 \leq x_3 \leq 42\right\},$$

$$f(x_1, x_2, x_3) = -x_1 x_2 x_3,$$

$$g_1(x_1, x_2, x_3) = -x_1 - 2x_2 - 2x_3$$

$$g_2(x_1, x_2, x_3) = -72 + x_1 + 2x_2 + 2x_3.$$

Show that this problem has a unique solution which one will compute by hand. Write then the corresponding user's defined subroutines `fun` and `gfun` and solve the problem with an interior point method like Herskovits'. Compare the results obtained by the two approaches.

Minimal perimeter

Let us define a polygon P by `npu` points on its upper part and `npl` points on its lower part (see Figure 4.3). The goal is to find a polygon P with minimal perimeter and fixed area:

$$\text{area } (P) = \frac{\pi}{4}. \tag{4.8}$$

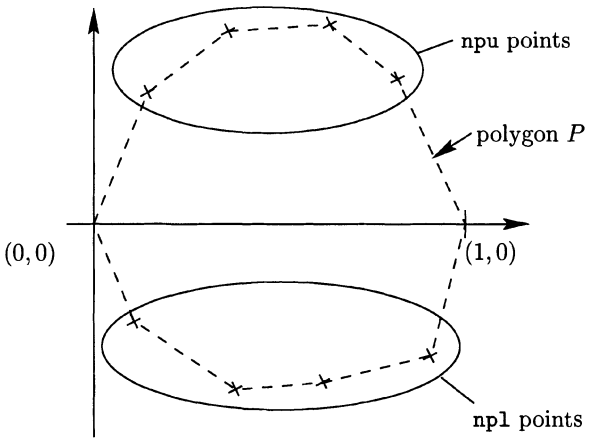


Figure 4.3. Data of the perimeter minimization problem

Starting from a given admissible solution, solve this problem by the above interior point algorithm. Represent the initial and the final solution by storing these solutions on two distinct files and by drawing the corresponding polygons using the plotter of your choice.

Chapter 5

Automatic Differentiation

5.1 Introduction

Computing gradients is a fundamental step in shape optimization. Gradients are needed in sensitivity analysis to measure the variation of performance $\frac{dj}{dz_i} \cdot dz_i$ induced by a small perturbation dz_i of the i control parameter. They are also used in reliability analysis for computing a first-order random model from known stochastic distributions of the data. Finally, we have seen in the previous chapter that gradients are needed in most algorithms used in mathematical programming, and more generally in most multipoint optimization techniques. If the functions to be differentiated are outputs of computer programmes, these gradients can be computed automatically by differentiating each line of these computer programmes. This is called automatic differentiation and is the topic of the present chapter. For complex cost functions used in shape optimization, automatic differentiation will have to be coupled with Lagrangian techniques, and this will be the purpose of the next chapter.

5.2 Computing gradients by finite differences

The easiest way to compute a gradient $\frac{dj(z)}{dz} = \nabla_z j(z)$ on a computer is to approximate its value by finite differences. Using a first-order Taylor expansion, the directional derivative at point z of the function j under study is simply identifed with the difference between two of its values

$$(\nabla j(z))_i = \frac{\partial j(z)}{\partial z_i} \approx \frac{j(z + \varepsilon e_i) - j(z)}{\varepsilon} \quad \text{for} \quad i = 1 \text{ to } n.$$

Above, ε is a conveniently chosen small parameter and $(e_i)_{i=1,n}$ denotes the canonical basis of $\mathbb{R}^n$.

This technique is very general: it can be applied to all types of shape optimization problems, and does not require specific developments. The numerical tools which are needed to compute gradients by finite differences in shape optimization are just a solver to compute the solution W of the state equation for

E. Laporte et al., *Numerical Methods in Sensitivity Analysis and Shape Optimization*

a given configuration z, and a subroutine to compute the cost function j for a given configuration z and state variable W.

Nevertheless, this technique has severe limitations. First, the result is rather sensitive to the choice of parameter ε. The value of ε must be optimized, and different values should in fact be used in different directions because the directional derivatives of j may have completely different characteristics (see the exercise at the end of the chapter).

Moreover, this technique requires evaluation at high accuracy of one or several values of the cost function (and thus to solve one occurence of the state equations) per direction. Computing a gradient in $\mathbb{R}^n$ by finite differences requires thus n different solutions of the state variables and n different evaluations of the cost functions. This therefore limits the use of finite difference techniques to situations where the number n of parameters is small.

Better techniques are thus needed, but this requires first a better understanding of the structure of a computer code.

5.3 Schematic structure of a computer code

Let us consider a given function

$$\begin{aligned} F: \quad & \mathbb{R}^n \to \mathbb{R}^m, \\ & x \to y = F(x). \end{aligned}$$

A basic computer code calculating the function F can be viewed as a sequence of simple instructions:

- affectations

$$\begin{aligned} v_{-n+1} &= x_1, \\ v_{-n+2} &= x_2, \\ &\cdot \\ &\cdot \\ v_0 &= x_n, \end{aligned}$$

- instructions

$$\begin{aligned} v_1 &= \varphi_1(\{v_j, j \in J_1, j < 1\}), \\ v_2 &= \varphi_2(\{v_j, j \in J_2, j < 2\}), \\ v_3 &= \varphi_3(\{v_j, j \in J_3, j < 3\}), \\ &\cdot \\ &\cdot \\ &\cdot \\ v_l &= \varphi_l(\{v_j, j \in J_l, j < l\}). \end{aligned}$$

- results

$$
\begin{aligned}
y_1 &= v_{l-m+1}, \\
y_2 &= v_{l-m+2}, \\
&\cdot \\
&\cdot \\
y_m &= v_l.
\end{aligned}
$$

There might be a very large number of instructions, up to several millions or more. But each instruction φ_i is very simple and only involves a small number of active arguments $\{v_j\} \in J_i$. Typical instructions are additions, multiplications, divisions, evaluation of simple trigonometric functions and so on. In any case, J_i is the set of the indices of variables which are effectively used during the instruction. It rarely contains more than four elements. For the instruction to be able to proceed, the value of these input variables $\{v_j\} \in J_i$ must be known when processing the instruction φ_i, which means that we have in general $j < i, \forall j \in J_i$.

For example, the computer code calculating the function

$$
F(x_1, x_2, x_3) = \begin{pmatrix} x_1^2 + x_2^2 \\ \cos x_3 + x_1 \end{pmatrix}
$$

formally corresponds to the sequence of instructions

$$
\begin{aligned}
v_{-2} &= x_1 \\
v_{-1} &= x_2 \\
v_0 &= x_3 \\
v_1 &= v_{-2} * v_{-2} \\
v_2 &= v_{-1} * v_{-1} \\
v_3 &= \cos v_0 \\
v_4 &= v_1 + v_2 \\
v_5 &= v_3 + v_{-2} \\
y_1 &= v_4 \\
y_2 &= v_5
\end{aligned}
$$

The above functional representation of the computer programme gives then a direct way of computing the derivative of F along the direction $\dot{x}$, simply by differentiating each individual instruction. This very simple differential calculus produces then the sequence of new instructions:

- affectations:

$$
\begin{aligned}
\dot{v}_{i-n} &= \dot{x}_i, 1 \leq i \leq n, \\
v_{i-n} &= x_i, 1 \leq i \leq n;
\end{aligned}
$$

- instructions:

$$\begin{aligned}&\text{(loop for } i = 1 \text{ to } l)\\ &\dot{v}_i = \sum_{j \in J_i} \frac{\partial \varphi_i}{\partial v_j}((v_k)_{k \in J_i}) \cdot \dot{v}_j,\\ &v_i = \varphi_i((v_k)_{k \in J_i}).\end{aligned}$$

- result:

$$\begin{aligned}&\frac{\partial y_i}{\partial x} \cdot \dot{x} = \dot{y}_i = \dot{v}_{l-m+i}, 1 \leq i \leq m,\\ &y_i = v_{l-m+i}, 1 \leq i \leq m.\end{aligned}$$

This updated programme now produces both the values $(y_i)_{i=1,m}$ of the function to be computed **and** the values of the directional derivatives $\dfrac{\partial y_i}{\partial x} \cdot \dot{x} = \dot{y}_i$.

5.4 Automatic differentiation

Modelization of a Fortran software

Generalizing the above simple model, we now detail the steps leading to the automatic production of a computer code calculating directional derivatives. For this purpose, we consider the particular case of a subroutine written in Fortran using N variables $v_1, v_2, \ldots, v_N$. These variables are numbered so that the n first variables ($n \leq N$) are the *independent input variables* with respect to which the differentiation is performed, and the m last variables $v_{N-m+1}, \ldots, v_N$ are the *output variables* which are the functions to be computed and differentiated. Variables which are both input and output variables will be numbered twice. A given variable v_i is said to become active when one assigns to it a value which depends on the input values of the independent variables. Furthermore, for any vector $v \in \mathbb{R}^N$ and for any subset A of $\{1, \ldots, N\}$, the vector v_A will denote the vector of $\mathbb{R}^{|A|}$ with components v_i with $i \in A$. Under this notation, the considered subroutine will consist of a sequence of K successive instructions

$$v_{\mu_k} := \varphi_k\left(v_{D_k}\right), \quad k = 1, \ldots, K. \tag{5.1}$$

At each instruction k, the variable v_{μ_k} is modified and receives the value taken by the function φ_k calculated with input data v_i, $i \in D_k$, with D_k a subset of $\{1, \ldots, N\}$. The different operations φ_k which are used in the subroutine belongs to a global set $\mathcal{F}$ of feasible instructions. They may represent elementary algebraic operations (+, -, *, /), intrinsic Fortran fuctions, group of such functions or even full subroutines. The class of instructions which can be used in Fortran will be denoted by $\mathcal{F}_F$ and is given by

$$\begin{aligned}\mathcal{F}_F = \{&+, -, *, /, \texttt{sqrt}, \texttt{exp}, \texttt{log}, \texttt{log10}, \texttt{sin},\\ &\texttt{cos}, \texttt{tan}, \texttt{asin}, \texttt{acos}, \texttt{atan}, \texttt{sinh}, \texttt{cosh}, \texttt{tanh}\}.\end{aligned}$$

Altogether, denoting by x the input variables $x = v_1, v_2, \ldots, v_n$ and by f the output variables, the subroutine (5.1) can finally be viewed as a function

$$f : x \in \mathbb{R}^n \mapsto f(x) = (v_{N-m+1}, \ldots, v_N) \in \mathbb{R}^m,$$

whose value at a given point x is obtained by running the successive intructions of the subroutine (5.1).

To illustrate the notation, let us consider the example (5.2) of the subroutine given by

$$\begin{aligned} &\texttt{v3} = \cos(\texttt{v1}) * \texttt{v2} * *3 \\ &\texttt{v4} = \texttt{v1} * *1.5 + 2 * \texttt{v3} \\ &\texttt{v4} = \texttt{v4}/\texttt{v3} \\ &\texttt{v5} = \log(1 + \texttt{v4} * *2) \end{aligned} \tag{5.2}$$

We have here $N = 5$ variables, $n = 2$ input data, $m = 1$ output data, and $K = 4$ instructions. The set of input variables is therefore $x = (\texttt{v1}, \texttt{v2})$, and the output variables is $f = \texttt{v5}$. The interpretation (5.1) of the different instructions is detailed in the following table:

instruction number k	instruction	number of the modified variable	function φ_k	set of local input data
1	`v3=cos(v1)*v2**3`	$\mu_1 = 3$	$\varphi_1(a,b) = cos(a)b^3$	$D_1 = \{1,2\}$
2	`v4=v1**1.5+2*v3`	$\mu_2 = 4$	$\varphi_2(a,b) = a^{1.5} + 2b$	$D_2 = \{1,3\}$
3	`v4=v4/v3`	$\mu_3 = 4$	$\varphi_3(a,b) = \frac{a}{b}$	$D_3 = \{3,4\}$
4	`v5=log(1+v4**2)`	$\mu_4 = 5$	$\varphi_4(a) = \log(1 + a^2)$	$D_4 = \{4\}$

Automatic differentiation in direct mode

Let us consider the subroutine of the previous section, viewed as a function $f : x \in \mathbb{R}^n \mapsto f(x)$ and let $d \in \mathbb{R}^n$ be a given direction of differentiation. Our objective in direct differentiation is to compute the directional derivative $Df(x) \cdot d$ of the function f at point x along the direction d. We will assume that each operation φ_k appearing in the subroutine is differentiable with respect to its arguments v_{D_k}. The trick is to observe that each active variable v_i appearing in such instructions can also be viewed as a function of x and that therefore its directional derivatives $Dv_i(x) \cdot d$ can be computed. A direct differentiation of the instruction k in the subroutine (5.1) then gives by directional differentiation of composed functions

$$Dv_{\mu_k}(x) \cdot d = \sum_{i \in D_k} \frac{\partial \varphi_k}{\partial v_i}(v_{D_k}) \, Dv_i(x) \cdot d.$$

This formula yields the value of the directional derivative of v_{μ_k} as a function of the directional derivatives of its arguments. Using this formula, and introducing

the notation

$$\dot{v}_i = Dv_i(x) \cdot d,$$

we can therefore propagate the value of the directional derivatives starting from the directional derivatives of the input arguments

$$\dot{v}_i = Dv_i(x) \cdot d := d_i, \quad \text{for } i = 1, \ldots, n,$$

running the intermediate instructions to compute both the intermediate directional derivatives and variables

$$\left.\begin{aligned} \dot{v}_{\mu_k} = Dv_{\mu_k}(x) \cdot d := \sum_{i \in D_k} \frac{\partial \varphi_k}{\partial v_i}(v_{D_k})\, \dot{v}_i \\ v_{\mu_k} := \varphi_k(v_{D_k}) \end{aligned}\right\}, \quad k = 1, \ldots, K,$$

and arriving at the final directional derivatives and values of the output variables

$$Df(x) \cdot d := (\dot{v}_{N-m+1}, \ldots, \dot{v}_N), \quad f(x) := (v_{N-m+1}, \ldots, v_N).$$

In the above procedure, the derivatives $\dot{v}_{\mu_k}$ are computed before the functions v_{μ_k}. This guarantees that the calculation of the directional derivatives uses the old value of v_{μ_k} in situations where the function φ_k depends on a previously computed value of the variable v_{μ_k}.

This type of calculation is called automatic differentiation in *direct mode* and the resulting code is the *linear tangent code* of (5.1). It can be summarized by the following sequence of instructions:

$$\begin{aligned} &\text{for } i = 1, \ldots, n, \\ &\quad \dot{v}_i := d_i\,; \\ &\text{for } k = 1, \ldots, K, \\ &\quad \dot{v}_{\mu_k} := \sum_{i \in D_k} \frac{\partial \varphi_k}{\partial v_i}(v_{D_k})\, \dot{v}_i\,; \\ &\quad v_{\mu_k} := \varphi_k(v_{D_k})\,; \\ &Df(x) \cdot d := (\dot{v}_{N-m+1}, \ldots, \dot{v}_N)\,; \\ &f(x) := (v_{N-m+1}, \ldots, v_N). \end{aligned} \tag{5.3}$$

This code is produced automatically by code differentiators such as O∂yssée or ADIFOR. It computes the value of f and of one directional derivative. It associates to each code variable v_i its directional derivative $\dot{v}_i$. The requirement in memory space is thus twice the one of the initial programme when all intermediate variables are active. As for the CPU time $T(f(x), Df(x) \cdot d)$ used for running the instruction (5.3), it can be proved using reasonable estimates on the running times associated to the operations $\mathcal{F}_F$ and to their derivatives that one has (see [17])

$$\frac{T(f(x), Df(x) \cdot d)}{T(f(x))} \leq 5, \tag{5.4}$$

where $T(f(x))$ denotes the CPU time used for evaluating $f(x)$. The total CPU time of the linear tangent code is thus at most four times the CPU times of the original code.

To illustrate automatic differentiation in direct mode, we give below the result obtained by applying the differentiator O∂yssée (version 1.6) to the subroutine (5.2):

```
sr01sttl = 0.
sr01sttl = -v1ttl*sin(v1)
sr01s = cos(v1)
v3ttl = 0.
v3ttl = (3*v2ttl*v2**2)*sr01s+v2**3*sr01sttl
v3 = sr01s*v2**3
v4ttl = 0.
v4ttl = 1.5*v1ttl*v1**(1.5-1.)+2*v3ttl
v4 = v1**1.5+2*v3
v4ttl = (v4ttl*v3-v4*v3ttl)/v3**2
v4 = v4/v3
sr01sttl = 2*v4ttl*v4
sr01s = 1+v4**2
v5ttl = 0.
v5ttl = sr01sttl/sr01s
v5 = log(sr01s)
```

It can be observed that:

- the suffix `ttl` is used to denote the directional derivatives $\dot{v}_i$ of the code variables v_i;
- a new intermediate variable `sr01s` has been automatically introduced to avoid redundant calculations;
- the initialization steps $\dot{v}_i := d_i$ required in (5.3) are suppressed because the directional derivatives `v1ttl` of `v2ttl` of the input variables are directly identified with the input parameters d_1 and d_2 giving the direction of differentiation;
- to compute several (p) directional derivatives in parallel, one just has to replace the scalar variables `vittl` by arrays `vittl(1 :p)`.

Automatic differentiation in inverse mode

As it will be seen later, directional derivatives obtained by multiplying the Jacobian matrix $\frac{Df}{dx} \in \mathbb{R}^{m \times n}$ by the direction of differentiation $d \in \mathbb{R}^n$ are less useful in shape optimization algorithms than the cotangent (adjoint) directional derivative obtained by multiplying the adjoint Jacobian matrix $\left(\frac{Df}{dx}\right)^t \in \mathbb{R}^{n \times m}$ by an adjoint direction $d \in R^m$. The interest of these adjoint derivatives

can be easily understood in situations where one wishes to compute the full Jacobian matrix of a function $f : \mathbb{R}^n \to \mathbb{R}^m$. This Jacobian can be obtained either by computing n directional derivatives along n independent directions d, or m cotangent directional derivatives along m independent adjoint directions d. The second choice is obviously much cheaper when one has m much smaller than n, which is the case in particular of the cost function for which we have $m = 1$.

This cotangent directional derivative $(Df(x))^t \cdot d$ with d given in $\mathbb{R}^m$ can also be obtained by automatic differentation. Following the point of view presented in [13], the source code (5.1) can be viewed as the composition of K applications, each independent application (instruction) k acting on the whole set of variables $v_1, \ldots, v_N$ but only modifying the variable v_{μ_k}. Each instruction can therefore be viewed as the application $\Phi_k : \mathbb{R}^N \to \mathbb{R}^N$ given by

$$(\Phi_k(v))_i = \begin{cases} v_i & \text{if } i \neq \mu_k, \\ \varphi_k(v_{D_k}) & \text{if } i = \mu_k. \end{cases} \tag{5.5}$$

The full code (5.1) is now viewed as the composition of K functions:

$$(\Phi_K \circ \cdots \circ \Phi_1)(v).$$

By differentiation, we therefore get

$$D(\Phi_K \circ \cdots \circ \Phi_1)(v) = \begin{pmatrix} * & * \\ Df(x) & * \end{pmatrix}.$$

After left multiplication by $\begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}^t$ with 0_p denoting the null vector in $\mathbb{R}^p$, we get

$$\begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}^t D(\Phi_K \circ \cdots \circ \Phi_1)(v) = (d \cdot Df(x), *).$$

Introducing the unit matrix I_p of order p and the null matrix $0_{p\times q}$ in $\mathbb{R}^{p\times q}$, we obtain after right multiplication

$$d \cdot Df(x) = \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}^t D(\Phi_K \circ \cdots \circ \Phi_1)(v) \begin{pmatrix} I_n \\ 0_{(N-n)\times n} \end{pmatrix},$$

which yields by transposition

$$(Df(x))^t \cdot d = \begin{pmatrix} I_n & 0_{n\times(N-n)} \end{pmatrix} (D(\Phi_K \circ \cdots \circ \Phi_1)(v))^t \cdot \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}.$$

Let us now denote by

$$v^k = (\Phi_k \circ \cdots \circ \Phi_1)(v),$$

the value of the code variables after the k first instructions. After development,

the above expression finally reduces to

$$\begin{aligned}&(Df(x))^t \cdot d\\&= \begin{pmatrix} I_n & 0_{n\times(N-n)} \end{pmatrix} \left(D\Phi_1\left(v^0\right)\right)^t \left(D\Phi_2\left(v^1\right)\right)^t \cdots \left(D\Phi_K\left(v^{K-1}\right)\right)^t \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}.\end{aligned} \tag{5.6}$$

The most efficient way to perform this sequence of matrix multiplications is to proceed from right to left, which reduces the above expression to a succession of matrix vector products (with auxiliary vector dual variables $\bar{v} = (\bar{v}_i)_{i=1,\ldots,N}$)

$$\begin{aligned}\bar{v} &\longleftarrow \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix},\\ \bar{v} &\longleftarrow \left(D\Phi_K\left(v^{K-1}\right)\right)^t \cdot \bar{v},\\ \bar{v} &\longleftarrow \left(D\Phi_k\left(v^{k-1}\right)\right)^t \cdot \bar{v}, \qquad (5.7)\\ &\vdots\\ \bar{v} &\longleftarrow \left(D\Phi_1\left(v^0\right)\right)^t \cdot \bar{v}, \qquad (5.8)\\ &\vdots\end{aligned}$$

ending by

$$(Df(x))^t \cdot d = \begin{pmatrix} I_n & 0_{n\times(N-n)} \end{pmatrix} \bar{v}.$$

Each intermediate product (5.7) can be further detailed: from (5.5), each intermediate matrix takes the form

$$\left(D\Phi_k\left(v^{k-1}\right)\right)^t = \begin{pmatrix} 1 & & \frac{\partial\varphi_k}{\partial v_1}\left(v^{k-1}\right) & & \\ & \ddots & \vdots & & \\ & 1 & \vdots & & \\ & & \frac{\partial\varphi_k}{\partial v_{\mu_k}}\left(v^{k-1}\right) & & \\ & & \vdots & 1 & \\ & & \vdots & & \ddots \\ & & \frac{\partial\varphi_k}{\partial v_N}\left(v^{k-1}\right) & & 1 \end{pmatrix},$$

and thus (5.7) corresponds to the detailed instructions

$$\begin{cases} \bar{v}_i \longleftarrow \bar{v}_i & \text{if } i \notin D_k \bigcup \{\mu_k\}, \\ \bar{v}_i \longleftarrow \bar{v}_i + \frac{\partial\varphi_k}{\partial v_i}\left(v^{k-1}\right)\bar{v}_{\mu_k} & \text{if } i \in D_k - \{\mu_k\}, \\ \bar{v}_{\mu_k} \longleftarrow \frac{\partial\varphi_k}{\partial v_{\mu_k}}\left(v^{k-1}\right)\bar{v}_{\mu_k}. & \end{cases}$$

Here the variables $\bar{v}_i$ for $i \neq \mu_k$ must be updated before $\bar{v}_{\mu_k}$, because we obviously need to use the old value of $\bar{v}_{\mu_k}$ when computing $\bar{v}_i$ ($i \in D_k - \{\mu_k\}$).

The above mode of automatic differentiation is the *adjoint mode* or *inverse mode* and the computer code obtained as output is the *cotangent linear code* of (5.1). It corresponds to the sequence of instructions:

$$\begin{aligned} &\text{for } i = 1, \ldots, N-m, \\ &\quad \bar{v}_i := 0\,; \end{aligned} \tag{5.9}$$

$$\begin{aligned} &\text{for } i = N-m+1, \ldots, N, \\ &\quad \bar{v}_i := d_i\,; \\ &\text{for } k = K, \ldots, 1 \\ &\quad \text{for } i \in D_k - \{\mu_k\}\,, \\ &\qquad \bar{v}_i \longleftarrow \bar{v}_i + \frac{\partial \varphi_k}{\partial v_i}\left(v^{k-1}\right)\bar{v}_{\mu_k}\,; \\ &\quad \bar{v}_{\mu_k} \longleftarrow \frac{\partial \varphi_k}{\partial v_{\mu_k}}\left(v^{k-1}\right)\bar{v}_{\mu_k}\,; \\ &Df(x)^t \cdot d := (\bar{v}_1, \ldots, \bar{v}_n)\,. \end{aligned}$$

To illustrate this mode, we give below the result obtained by O∂yssée when differentiating the code (5.2) in inverse mode:

```
C
C Initializations of local variables
C

      v4ccl = 0.
      v3ccl = 0.
      sr01sccl = 0.
C
C Trajectory
C

      save1 = v5
      save2 = sr01s
      sr01s = cos(v1)
      save3 = v3
      v3 = sr01s*v2**3
      save4 = v4
      v4 = v1**1.5+2*v3
      save5 = v4
      v4 = v4/v3
      save6 = sr01s
      sr01s = 1+v4**2
      save7 = v5
      v5 = log(sr01s)
C
C Transposed linear forms
```

C

```
      v5 = save7
      sr01sccl = sr01sccl+v5ccl*(1./sr01s)
      v5ccl = 0.
      v5ccl = 0.
      sr01s = save6
      v4ccl = v4ccl+sr01sccl*(2*v4)
      sr01sccl = 0.
      v4 = save5
      v3ccl = v3ccl-v4ccl*((1./v3**2)*v4)
      v4ccl = v4ccl*((1./v3**2)*v3)
      v4 = save4
      v1ccl = v1ccl+v4ccl*(1.5*v1**(1.5-1.))
      v3ccl = v3ccl+2*v4ccl
      v4ccl = 0.
      v4ccl = 0.
      v3 = save3
      v2ccl = v2ccl+v3ccl*((3*v2**2)*sr01s)
      sr01sccl = sr01sccl+v3ccl*v2**3
      v3ccl = 0.
      v3ccl = 0.
      sr01s = save2
      v1ccl = v1ccl-sr01sccl*sin(v1)
      sr01sccl = 0.
      sr01sccl = 0.
      v5 = save1
```

One can observe that

- the suffix `ccl` is used to denote the dual variables $\bar{v}_i$;
- `v1ccl` and `v2ccl` are not initialized to zero. In fact, the subroutine computes $d+(Df(x))^t \cdot d$, and it can therefore be used inside an interpocedural mode with differentiating subroutines calling themselves other subroutines;
- the subroutine is split into two main steps. The first one computes the value of the intermediate variables v_i and the second one runs the iterations (5.9). Indeed, running the instruction (5.9) requires the value of v^{K-1}, then of v^{K-2}, and so on in reverse order down to v^0. Therefore, the variables v_i^k must be precomputed and their values before their updates in φ_k must be stored. The other choice of recomputing the full variable v^k at each step of (5.9) would require a CPU time in $(T(f(x)))^2$.

The operation cost of this differentiation in adjoint mode is again given by

$$OPS\left(\frac{DF^t}{Dx}\cdot w\right) = \sum_l OPS(D\varphi_i^t) \approx 4\sum_l OPS(\varphi_i) \approx 4\,OPS(F),$$

independently of the size n of the argument x. For a scalar function j, one therefore obtains the full Jacobian in $4 \times OPS$ operations, whatever the number n of input variables is. This is in sharp contrast with finite difference approximations.

The total CPU time of the differentiation in reverse mode can also be estimated by

$$\frac{T\left(f(x), (Df(x))^t \cdot d\right)}{T(f(x))} \leq C, \tag{5.10}$$

with C a function of the computer hardware characteristics and in particular a function of the memory access time.

The inverse mode has nevertheless a severe limitation in memory requirement. By opposition with the direct mode where both F and DF were computed in parallel, the adjoint mode proceeds in reverse mode, and therefore stores the intermediate results

$$(v_i)_{i=1,l}$$

which requires a tremendous amount ($\approx OPS(F)$ bytes) of memory. This can be improved by storing only partial results and by recomputing the other ones when needed, and by carefully predecomposing the codes in smaller subroutines to be differentiated in interprocedural mode (only storing argument values).

5.5 Examples of automatic differentiation by O∂yssée

The automatic differentiator O∂yssée processes Fortran subroutines computing $f(x)$ and produces a subroutine computing either the directional derivative $\langle Df(x), y\rangle$ (linear tangent mode), or a subroutine computing the whole Jacobian $Df(x)$ using n independent directional derivatives (direct mode), or a subroutine computing the adjoint directional derivative $\langle Df^t(x), y\rangle$ (reverse mode). For illustration purposes, let us consider the following subroutine:

```
      subroutine f(x,y)
c
      real x(2),y(2)
c
      y(1)=2*x(1)+x(2)
      y(2)=3*x(1)+4*x(2)
c
      return
      end
```

The first result produced by O∂yssée in *direct mode* computes $dydx = \frac{dy}{dx}(x) \cdot A$ with $A = dxdx$ a given square matrix of order n:

```
      SUBROUTINE fd (x, y, dxdx, dydx)
```

```
REAL x(2), y(2)
INTEGER n2
INTEGER n1
REAL dxdx(2,2)
REAL dydx(2,2)

DO n1 = 1, 2
  DO n2 = 1, 2
    dydx(n1,n2) = 0.
  END DO
END DO
DO n1 = 1, 2
  dydx(1,n1) = 2*dxdx(1,n1)+dxdx(2,n1)
END DO
y(1) = 2*x(1)+x(2)
DO n1 = 1, 2
  dydx(2,n1) = 3*dxdx(1,n1)+4*dxdx(2,n1)
END DO
y(2) = 3*x(1)+4*x(2)
RETURN
END
```

The second result is obtained in *linear tangent mode* and computes the directional derivative $yttl = \left\langle \frac{dy}{dx}(x), xttl \right\rangle$ with $xttl$ a given input vector of order n:

```
SUBROUTINE ftl (x, y, xttl, yttl)

REAL x(2), y(2)
DIMENSION xttl(2)
DIMENSION yttl(2)

DO n1 = 1, 2
  yttl(n1) = 0.
END DO
yttl(1) = 2*xttl(1)+xttl(2)
y(1) = 2*x(1)+x(2)
yttl(2) = 3*xttl(1)+4*xttl(2)
y(2) = 3*x(1)+4*x(2)
RETURN
END
```

The last result is obtained in *inverse mode* and computes the adjoint directional derivative $xccl = \left\langle \frac{dy}{dx}^t(x), yccl \right\rangle$ with $yccl$ a given input vector of order m:

```
SUBROUTINE fcl (x, y, xccl, yccl)

REAL x(2), y(2)
```

```
      DIMENSION yccl(2)
      DIMENSION xccl(2)
      DIMENSION save1(1:2)

C
C Trajectory
C

      DO n1 = 1, 2
        save1(n1) = y(n1)
      END DO
      save2 = y(1)
      y(1) = 2*x(1)+x(2)
      save3 = y(2)
      y(2) = 3*x(1)+4*x(2)
C
C Transposed linear forms
C

      y(2) = save3
      xccl(1) = xccl(1)+3*yccl(2)
      xccl(2) = xccl(2)+4*yccl(2)
      yccl(2) = 0.
      y(1) = save2
      xccl(1) = xccl(1)+2*yccl(1)
      xccl(2) = xccl(2)+yccl(1)
      yccl(1) = 0.
      DO n1 = 2, 1, -1
        yccl(n1) = 0.
      END DO
      DO n1 = 2, 1, -1
        y(n1) = save1(n1)
      END DO
      RETURN
      END
```

5.6 Exercises

Finite differences

Let us consider the following function:

$$\begin{aligned} f(x) &= \left(x_1^2 \cos(x_2) + 4x_3,\ x_1 e^{x_2}/x_3,\ 5x_3,\ 2x_1 + \log(1+x_2)\right) \quad (5.11) \\ &= \left(f^i(x)\right)_{i=1,\ldots,4} \end{aligned}$$

with $x = (x_1, x_2, x_3) \in \mathbb{R}^3$. Write a computer programme computing $f^i(x+\epsilon e_j)$ for given i and j (with $e_j = (\delta_{kj})_{k=1,\ldots,3}$ the j unit vector of $\mathbb{R}^3$), and for a

given logarithmic sequence of $\epsilon \in [10^{-10}, 10^{-2}]$ ($\epsilon = a \times b^n$, n to be chosen by the user). The results $(\epsilon, f^i(x + \epsilon e_j))$ will be saved on a file to be displayed by your favorite plotter. Use these results for computing different estimates of $\frac{\partial f^i}{\partial x_j}(x)$ and observe the dependence on ϵ.

Automatic Differentiation

Obtain by O∂yssée the subroutines computing

$$\langle Df(x), u \rangle \quad \text{and} \quad \langle Df^t(x), u \rangle$$

with f given by (5.11). Write a programme computing the gradient

$$\frac{\partial f^i}{\partial x_j}(x)$$

for each x, i and j using either

- O∂yssée direct mode,
- O∂yssée linear tangent mode,
- O∂yssée reverse mode,
- finite differences with a good choice of ϵ (see previous exercise).

Compare the different results in cost and accuracy.

Chapter 6

Computing Gradients by Adjoint States

6.1 Structure of the cost function

We have already observed that the key point in sensitivity analysis or in most optimization algorithms is the accurate calculation of the gradient of the cost function j. In order to detail this calculation, it is useful to first recall the structure of a cost function in shape optimization. In such problems, the cost function is defined by

$$\begin{cases} j : Z_{adm} \subset \mathbb{R}^n \to \mathbb{R}, \\ z \mapsto j(z) = J(\underline{X}(z), W(z)), \end{cases}$$

the state variable $W = W(z)$ being the solution of a (direct) system of state equations

$$f_h(\underline{X}(z), W) = 0 \quad \text{in} \quad \mathbb{R}^p, \tag{6.1}$$

which are themselves functions of the present position $\underline{X}(z)$ of the computational grid.

The form (6.1) of the cost functional j expresses a double dependency: by construction, this function depends both on the present value of the control z and of the value of the state variable W depending implicitly on the control z through the equation of state. Using the chain rule, the differentiation of j with respect to z then gives

$$(\nabla j)_i(z) = \left\langle \frac{\partial J(\underline{X})}{\partial \underline{X}}, \frac{\partial \underline{X}}{\partial z_i} \right\rangle + \left\langle \frac{\partial J(\underline{X}, W)}{\partial W}, \frac{\partial W(z)}{\partial z_i} \right\rangle \tag{6.2}$$

where $\langle ., . \rangle$ is an alternate notation for the dot product in $\mathbb{R}^n$ or in $\mathbb{R}^p$.

The main problem in this expression is the calculation of the gradient $\dfrac{\partial W(z)}{\partial z}$ of the state variable with respect to the control z. First, W is only known

E. Laporte et al., *Numerical Methods in Sensitivity Analysis and Shape Optimization*

implicitly as a function of z, and second, W usually belongs to a space of very large dimension. A good strategy to overcome this difficulty is then to proceed by duality and to compute directly the product $\left(\frac{\partial W(z)}{\partial z}\right)^t \frac{\partial J(\underline{X}, W)}{\partial W}$ by introducing an adequate adjoint state. This is precisely the purpose of the section to come.

6.2 A Lagrangian approach

The idea behind the Lagrangian calculation of gradients is to consider the state equation $f_h(\underline{X}, W) = 0$ as an additional constraint on the problem under study and to explicitly introduce its corresponding Lagrange multipliers. This amounts to the introduction of the *Lagrangian functional*

$$L : \begin{cases} \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}, \\ (z, W, P) \mapsto L(z, W, P) \end{cases}$$

given by

$$L(z, W, P) = J(\underline{X}(z), W) + \langle P, f_h(\underline{X}(z), W) \rangle. \tag{6.3}$$

In this form, the state variable W is supposed to be independent of the control variable z and the vector $P \in \mathbb{R}^p$ represents the vector of Lagrange multipliers of the constraint $f_h(\underline{X}, W) = 0$ satisfied by the pair (z, W).

In this Lagrangian framework, from the identity

$$\frac{\partial L(z, W, P)}{\partial P} = f_h(\underline{X}, W),$$

we first have the equivalence

$$\frac{\partial L(z, W, P)}{\partial P}|_{(z,W,P)} = 0 \text{ if and only if } W = W(z). \tag{6.4}$$

Thus, calculating the state variable $W(z)$ amounts to cancelling the derivative of L with respect to P.

We introduce then the adjoint state through

Definition 6.1 *We define the adjoint state (or adjoint variable) $P(z)$ associated to the state variable $W(z)$ to be the vector P in $\mathbb{R}^p$ obtained by solving the linear adjoint state equation*

$$\frac{\partial L(z, W(z), P)}{\partial W} = 0 \in \mathbb{R}^p. \tag{6.5}$$

The adjoint system (6.5) characterizes therefore the adjoint state P as the solution of the linear system obtained by cancelling the derivative of L with respect to W, which after development is written

$$\left(\frac{\partial f_h(\underline{X}, W)}{\partial W}\right)^t|_{(z,W)} \cdot P = -\frac{\partial J(\underline{X}, W)}{\partial W}|_{(z,W)}. \tag{6.6}$$

With this notation, the gradient of the cost function can then be obtained by a very classical formula.

Theorem 6.1 *The gradient of the cost functional j at point z is given by*

$$\begin{aligned}\nabla j(z) &= \frac{\partial L(z,W,P)}{\partial z}|_{(z,W(z),P(z))} \\ &= \left[\frac{\partial J(\underline{X},W(z))}{\partial \underline{X}} + \left(\frac{\partial f_h(\underline{X},W(z))}{\partial \underline{X}}\right)^t \cdot P(z)\right] \cdot \frac{\partial \underline{X}}{\partial z}. \end{aligned} \quad (6.7)$$

Proof By direct differentiation of the state equation and by considering the grid $\underline{X}$ as an explicit function of the control z, we first get, for all $z \in Z_{adm}$,

$$\frac{\partial f_h}{\partial z}(\underline{X}(z),W)\cdot dz + \frac{\partial f_h}{\partial W}(\underline{X}(z),W)\cdot \frac{dW}{dz}(z)\cdot dz = 0.$$

A dot product of this equation with the adjoint state vector $P(z)$ yields after transposition

$$\begin{aligned}\langle P(z), \frac{\partial f_h}{\partial z}(\underline{X}(z),W)\cdot dz\rangle &= -\left\langle P(z), \frac{\partial f_h}{\partial W}(\underline{X}(z),W)\cdot \frac{dW}{dz}(z)\cdot dz\right\rangle \\ &= -\left\langle \frac{\partial f_h}{\partial W}(\underline{X}(z),W)^t \cdot P(z), \frac{dW}{dz}(z)\cdot dz\right\rangle.\end{aligned}$$

By construction of the adjoint state, this identity reduces to

$$\begin{aligned}\langle P(z), \frac{\partial f_h}{\partial z}(\underline{X}(z),W)\cdot dz\rangle &= \left\langle \frac{\partial J}{\partial W}(\underline{X}(z),W), \frac{dW}{dz}(z)\cdot dz\right\rangle \\ &\quad \forall z \in Z_{adm}, \forall dz \in \mathbb{R}^n,\end{aligned}$$

from which we finally get

$$\begin{aligned}\langle \nabla j(z), dz\rangle &= \left\langle \frac{\partial J}{\partial z}(\underline{X}(z),W), dz\right\rangle + \left\langle \frac{\partial J}{\partial W}(\underline{X}(z),W), \frac{dW(z)}{dz}\cdot dz\right\rangle \\ &= \left\langle \frac{\partial J}{\partial z}(\underline{X}(z),W), dz\right\rangle + \left\langle P(z), \frac{\partial f_h}{\partial z}(\underline{X}(z),W)\cdot dz\right\rangle \\ &= \left\langle \frac{\partial \mathcal{L}}{\partial z}(z,W(z),P(z)), dz\right\rangle, \forall dz \in \mathbb{R}^n.\end{aligned}$$

□

Remark 6.1 Introducing the adjoint state uses at best the fact that the expression of the gradient of the cost function $j(z)$ does not use the whole Jacobian matrix

$$\frac{DW}{Dz} \in \mathbb{R}^p \times \mathbb{R}^n$$

but only the left product of this Jacobian with the gradient $\frac{\partial J}{\partial W} \in \mathbb{R}^p$.

In fact, formula (6.7) can be obtained by a direct application of the implicit function theorem as follows:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial z} &= \frac{\partial J}{\partial z} + P \cdot \frac{\partial f_h}{\partial z} \\ &= \frac{\partial J}{\partial z} - \left[\left(\frac{\partial f_h}{\partial W}\right)^{-t} \cdot \frac{\partial J}{\partial W}\right] \cdot \frac{\partial f_h}{\partial z} \\ &= \frac{\partial J}{\partial z} - \frac{\partial J}{\partial W} \cdot \left(\frac{\partial f_h}{\partial W}\right)^{-1} \cdot \frac{\partial f_h}{\partial z} \\ &= \frac{\partial J}{\partial z} + \frac{\partial J}{\partial W} \cdot \frac{\partial W}{\partial z} \\ &= \frac{dj}{dz}.\end{aligned}$$

6.3 Application to automatic differentiation in adjoint mode

The automatic procedure of the previous chapter computing $(\frac{DF}{Dx})^t \cdot w$ by automatic differentiation in adjoint mode can be recovered by the above Lagrangian framework. We will illustrate this point in the simple case where we have $\mu(i) = i$. In this situation, we introduce the cost function

$$j(x) = w \cdot y,$$

with $w \in \mathbb{R}^m$ given and $y(x) \in \mathbb{R}^m$ the solution of the system of state equations obtained by writing the different instructions of the subroutine under study:

$$\begin{aligned}&v_{i-n} = x_i, \quad 1 \leq i \leq n, \\ &v_i = \varphi_i(v_{D_i}), 1 \leq i \leq l, \\ &y_i = v_{l-m+i}, 1 \leq i \leq m.\end{aligned}$$

The corresponding Lagrangian is now

$$\begin{aligned}\mathcal{L}(x,(y,v),(\bar{w},\bar{v})) &= w \cdot y - \sum_{i=1}^{m} \bar{w}_i (y_i - v_{l-m+i}) \\ &+ \sum_{k=1}^{l} \bar{v}_k (\varphi_k - v_k) + \sum_{r=1}^{n} \bar{v}_{r-n}(x_r - v_{r-n}).\end{aligned}$$

Solving the state equation first yields the value of v and y as a function of w. The adjoint state equation then takes the form

$$\frac{\partial \mathcal{L}}{\partial (y,v)} = 0,$$

that is

$$\begin{aligned}&\bar{w} = w, \\ &\sum_{i=1}^{m} \bar{w}_i \delta_{j,l-m+i} + \sum_{k=1}^{l} \bar{v}_k \left(\frac{\partial \varphi_k}{\partial v_j} - \delta_{jk}\right) - \sum_{r=1}^{n} \bar{v}_{r-n} \delta_{j,r-n} = 0, \forall -n < j \leq l.\end{aligned}$$

After elimination of the adjoint variables $\bar{w}$ and treating first the case $1 \leq j \leq l$, these adjoint state equations reduce first to

$$\bar{v}_j = \sum_{i=1}^{m} w_i \delta_{j,l-m+i} + \sum_{j \in D_k} \left[\frac{\partial \varphi_k}{\partial v_j}(v_{D_k}) \right]^t \cdot \bar{v}_k, l \geq j \geq -n,$$

which is the loop (5.9) introduced in the automatic differentiation in reverse mode. A direct application of (6.7) then yields the final result

$$\left(\frac{dj}{dx}\right)_r^t = \left[\left(\frac{dF}{dx}\right)^t \cdot w\right]_r = \left(\frac{\partial \mathcal{L}}{\partial x}\right)_r^t = \bar{v}_{r-n}, 1 \leq r \leq n.$$

6.4 Numerical calculation of the gradient of the cost function

The Lagrangian approach leads to a very general numerical algorithm for computing rapidly the value $j(z)$ and the gradient $\nabla j(z)$ of the cost function. The algorithm is independent of the practical optimization strategy which is employed and can be used in all generality in any sensitivity analysis. As summarized below, it only requires two specific numerical solvers: one for solving the state equation, one for computing the adjoint state. It is organized as follows:

1. Calculation of the grid deformation $\underline{X} = \underline{X}(z)$.

2. Call of the numerical solver of the state equation $f_h(\underline{X}, W) = 0$ for calculating the state vector $W(z)$.

3. Call of the computer programme computing the cost function $j(z) = J(\underline{X}, W(z))$.

4. Calculation of the gradients

$$\frac{\partial J}{\partial \underline{X}} \text{ and } \frac{\partial J}{\partial W},$$

using automatic differentiation in adjoint mode of the programme calculating the cost function $j(z) = J(\underline{X}, W)$.

5. Solution of the linear adjoint state equation in P (see the next chapter for more details)

$$\left(\frac{\partial f_h(\underline{X}, W)}{\partial W}\right)^t |_{(\underline{X},W)} \cdot P(z) = -\frac{\partial J(\underline{X}, W)}{\partial W} |_{(\underline{X},W)}.$$

6. Calculation of the adjoint directional derivative

$$\left(\frac{\partial f_h(\underline{X}, W(z))}{\partial \underline{X}}\right)^t \cdot P(z),$$

by automatic differentiation in adjoint mode of the discrete state equation $f_h(\underline{X}, W) = 0$.

7. Calculation of the grid gradient by (6.7)

$$\frac{dJ}{d\underline{X}} = \frac{\partial J(\underline{X}, W(z))}{\partial \underline{X}} + \left(\frac{\partial f_h(\underline{X}, W(z))}{\partial \underline{X}}\right)^t \cdot P(z). \tag{6.8}$$

8. Final calculation of the gradient of the cost function by computing the adjoint directional derivative

$$\frac{dj}{dz} = \left(\frac{\partial \underline{X}}{\partial z}\right)^t \cdot \frac{dJ}{d\underline{X}},$$

using automatic differentiation in adjoint mode of the programme calculating the grid deformation $\underline{X} = \underline{X}(z)$.

This algorithm may have different implementations depending on the type of solver (explicit or implicit) used for the direct solver. In fact, as illustrated in the next chapter, the various applications of the above algorithm differ through the type of discretization scheme which is used, the structure of the state equation's solver and the strategy of solution used for the adjoint state equation.

Remark 6.2 We have presented above a technique for computing the gradient of the cost function. It can be easily adapted to the calculation of the gradient of a design constraint of the type $g(z, W(z)) \leq 0$. The only update which is needed is to introduce in the above algorithm one additional adjoint state by scalar constraint.

Remark 6.3 Very often, the components of the grid gradients $\frac{dJ}{d\underline{X}_i}$ with respect to the position of the i internal grid point are very small, compared to what is observed at the boundary nodes, meaning that the cost function is not influenced by the grid distribution inside the domain (cf. for example Figure 7.14). In such situations, we can replace the gradient of j by the simpler approximation

$$\frac{dj}{dz} = \sum_{i \in \gamma} \frac{dJ}{d\underline{X}_i} \frac{\partial \underline{X}_i}{\partial z}.$$

6.5 Calculation of the Hessian

The second derivative of the cost function j may be needed when using a full Newton's algorithm or when designing experience plans. The technique for

computing higher order derivatives of the function j up to order p is to use first a direct mode to compute the derivatives of the grid and of the state variable with respect to the control z up to the order $p-1$ (by solving in fact the linearized equations of mesh deformation and of state up to the order $p-1$), and then to report these derivatives in an adjoint expression of the higher order derivatives.

To see how to apply this technique to the calculation of the Hessian of j, we first differentiate the cost function and the equation of state with respect to the control z_i, yielding

$$\frac{\partial j}{\partial z_i}(z) = \left\langle \frac{\partial J}{\partial \underline{X}}, \frac{\partial \underline{X}}{\partial z_i} \right\rangle + \left\langle \frac{\partial J}{\partial W}, \frac{\partial W}{\partial z_i} \right\rangle, \tag{6.9}$$

$$0 = \frac{\partial f_h}{\partial W} \cdot \frac{\partial W}{\partial z_i} + \frac{\partial f_h}{\partial \underline{X}} \cdot \frac{\partial \underline{X}}{\partial z_i}. \tag{6.10}$$

The derivatives $\dfrac{\partial \underline{X}}{\partial z_i}$ are to be computed by automatic differentiation of the grid deformation map $\underline{X} = \underline{X}(z)$ in direct mode, and the derivatives $\dfrac{\partial W}{\partial z_i}$ are obtained by solving n occurences of the linearized state equation (6.10). Each linearized equation has the complexity of one adjoint state equation, and might be expensive to solve.

But once these derivatives are computed, the second-order derivatives of j can be directly obtained by simple differentiation of the first-order developments (6.9)–(6.10) which gives (assuming here for simplicity that $\underline{X}(z)$ is a linear map in z)

$$\begin{aligned}\frac{\partial^2 j}{\partial z_j \partial z_i}(z) &= \langle \frac{\partial J}{\partial W}, \frac{\partial^2 W}{\partial z_j \partial z_i} \rangle \\ &+ \left\langle \frac{\partial}{\partial(\underline{X}, W)} \left[\langle \frac{\partial J}{\partial \underline{X}}, \frac{\partial \underline{X}}{\partial z_i} \rangle + \langle \frac{\partial J}{\partial W}, \frac{\partial W}{\partial z_i} \rangle \right]_{|d\underline{X}_{,i}, dW_{,i}}, \left(\frac{\partial \underline{X}}{\partial z_j}, \frac{\partial W}{\partial z_j} \right) \right\rangle,\end{aligned} \tag{6.11}$$

$$\begin{aligned}0 &= \frac{\partial f_h}{\partial W} \cdot \frac{\partial^2 W}{\partial z_j \partial z_i} \\ &+ \frac{\partial}{\partial(\underline{X}, W)} \left[\frac{\partial f_h}{\partial W} \cdot \frac{\partial W}{\partial z_i} + \frac{\partial f_h}{\partial \underline{X}} \cdot \frac{\partial \underline{X}}{\partial z_i} \right]_{|d\underline{X}_{,i}, dW_{,i}} \cdot \left(\frac{\partial \underline{X}}{\partial z_j}, \frac{\partial W}{\partial z_j} \right).\end{aligned} \tag{6.12}$$

In the above formula, for any given $\mathcal{F}\left(\underline{X}, W, \dfrac{\partial \underline{X}}{\partial z_i}, \dfrac{\partial W}{\partial z_i}\right)$ depending on $\underline{X}$, W, $\dfrac{\partial \underline{X}}{\partial z_i}$ and $\dfrac{\partial W}{\partial z_i}$, the notation $\dfrac{\partial}{\partial(\underline{X}, W)} \left[\mathcal{F}(\underline{X}, W, \dfrac{\partial \underline{X}}{\partial z_i}, \dfrac{\partial W}{\partial z_i}) \right]_{|d\underline{X}_{,i}, dW_{,i}}$ denotes the partial derivative of the functional $\mathcal{F}$ with respect to the primal variables $\underline{X}$ and W, the values of the gradients $\frac{\partial \underline{X}}{\partial z_i}$ and $\frac{\partial W}{\partial z_i}$ being fixed. We can now

eliminate the second-order derivative $\dfrac{\partial^2 W}{\partial z_j \partial z_i}$ by using Equation (6.12) to obtain

$$\begin{aligned}
&\langle \frac{\partial J}{\partial W}(\underline{X}, W), \frac{\partial^2 W}{\partial z_j \partial z_i} \rangle \\
&= -\left\langle \left(\frac{\partial f_h}{\partial W}\right)^{-t} \frac{\partial J}{\partial W}, \frac{\partial}{\partial(\underline{X}, W)} \left[\frac{\partial f_h}{\partial W} \cdot \frac{\partial W}{\partial z_i} + \frac{\partial f}{\partial \underline{X}} \cdot \frac{\partial \underline{X}}{\partial z_i}\right]_{|d\underline{X}_{,i}, dW_{,i}} \cdot \left(\frac{\partial \underline{X}}{\partial z_j}, \frac{\partial W}{\partial z_j}\right) \right\rangle .
\end{aligned}$$

Using the definition of the adjoint state P and reporting the result in (6.11) yields finally

$$\begin{aligned}
\frac{\partial^2 j}{\partial z_j \partial z_i}(z) \;=\; &\left\langle \frac{\partial}{\partial(\underline{X}, W)} \left[\langle \frac{\partial J}{\partial \underline{X}}, \frac{\partial \underline{X}}{\partial z_i} \rangle + \langle \frac{\partial J}{\partial W}, \frac{\partial W}{\partial z_i} \rangle \right.\right. \\
&\left.\left. + \langle P, \frac{\partial f_h}{\partial W} \cdot \frac{\partial W}{\partial z_i} + \frac{\partial f_h}{\partial \underline{X}} \cdot \frac{\partial \underline{X}}{\partial z_i} \rangle \right]_{|d\underline{X}_{,i}, dW_{,i}, P}, \left(\frac{\partial \underline{X}}{\partial z_j}, \frac{\partial W}{\partial z_j}\right) \right\rangle ,
\end{aligned}$$

that is

$$\begin{aligned}
&\frac{\partial^2 j}{\partial z_j \partial z_i}(z) \\
&= \left\langle \frac{\partial}{\partial(\underline{X}, W)} \left[\langle \frac{\partial L}{\partial(\underline{X}, W)}(\underline{X}, W, P), \left(\frac{\partial \underline{X}}{\partial z_i}, \frac{\partial W}{\partial z_i}\right) \rangle \right]_{|d\underline{X}_{,i}, dW_{,i}, P}, \left(\frac{\partial \underline{X}}{\partial z_j}, \frac{\partial W}{\partial z_j}\right) \right\rangle .
\end{aligned} \tag{6.13}$$

Based on this formula, the algorithm for computing the Hessian of the cost function j, knowing the computational grid $\underline{X}$, the state W and the adjoint state P, becomes

1. Compute the derivatives $\dfrac{\partial \underline{X}}{\partial z_i}$ by automatic differentiation in direct mode of the grid deformation map $\underline{X} = \underline{X}(z)$ (n directional derivatives in z).
2. Compute the derivatives $\dfrac{\partial W}{\partial z_i}$ by solving the n linearized equations of state
$$\frac{\partial f_h}{\partial W} \frac{\partial W}{\partial z_i} = -\frac{\partial f_h}{\partial \underline{X}} \frac{\partial \underline{X}}{\partial z_i}.$$
3. Obtain a subroutine computing the directional derivative
$$\begin{aligned}
dL(\underline{X}, W, d\underline{X}, dW, P) = &\left\langle \frac{\partial J}{\partial \underline{X}}(\underline{X}, W), d\underline{X} \right\rangle + \left\langle \frac{\partial J}{\partial W}(\underline{X}, W), dW \right\rangle \\
&+ \left\langle P, \frac{\partial f_h}{\partial W}(\underline{X}, W) \cdot dW + \frac{\partial f_h}{\partial \underline{X}}(\underline{X}, W) \cdot d\underline{X} \right\rangle ,
\end{aligned}$$
by direct differentiation in direct mode and with respect to the variable $(\underline{X}, W)$ of the programme computing the Lagrangian
$$L(\underline{X}, W, P) = J(\underline{X}, W) + \langle P, f_h(\underline{X}, W) \rangle .$$

4. Obtain a subroutine computing directional derivatives

$$d^2L(\underline{X}, W, d\underline{X}, dW, d\underline{X}', dW', P)$$
$$= \frac{\partial}{\partial(\underline{X}, W)} dL(\underline{X}, W, d\underline{X}, dW, P) \cdot (d\underline{X}', dW')$$

by automatic differentiation in direct mode and with respect to the variable $(\underline{X}, W)$ of the previous programme computing the gradient

$$dL(\underline{X}, W, d\underline{X}, dW, P).$$

Use this programme $n(n+1)/2$ times to compute the Hessian

$$\frac{\partial^2 j}{\partial z_j \partial z_i}(z) = d^2L\left(\underline{X}, W, \frac{\partial \underline{X}}{\partial z_i}, \frac{\partial W}{\partial z_i}, \frac{\partial \underline{X}}{\partial z_j}, \frac{\partial W}{\partial z_j}, P\right).$$

Remark 6.4 In fact, the expression (6.13) of the Hessian of j can also be obtained by direct differentiation of the adjoint formula (6.8) with respect to z, simply by checking the algebraic identity

$$\left\langle \frac{dP}{dz}, \frac{\partial f_h}{\partial \underline{X}} \frac{d\underline{X}}{dz'} \right\rangle = \left\langle \frac{d}{dz}\left(\frac{\partial J}{\partial W}\right), \frac{dW}{dz'} \right\rangle + \left\langle P, \frac{d}{dz}\left(\frac{\partial f_h}{\partial W}\right) \cdot \frac{dW}{dz'} \right\rangle.$$

Chapter 7

Applications

7.1 Introduction

We have now a complete set of numerical tools for handling realistic shape optimization problems: space discretization reducing the problem to a finite-dimensional constrained optimization problem, grid deformation methodologies, curve parametrization by splines, constrained optimization algorithms using interior point algorithms on the space $\mathbb{R}^n$ of design parameters, gradient calculations combining, as in the previous chapter, adjoint state techniques and automatic differentiation of computer codes.

We will now illustrate the complete methodology on problems of increasing complexity: inverse problem in diffusion, drag reduction in inviscid aerodynamics, stability optimization in aeroelasticity. These two-dimensional problems will involve not only gradient evaluations, as required in any basic sensitivity analysis, but also full optimization runs.

7.2 Inverse problem in diffusion

Isotropic diffusion problems governed by the Poisson equation are quite easy to solve numerically and are no longer of great practical interest. As detailed in [34], their study in shape optimization is nevertheless quite interesting from an academic point of view because they give a good framework for the theoretical and the numerical validation of the different tools introduced up to now. They also are good benchmarks to assess the numerical performances and complexity of the different algorithms proposed in shape optimization.

Problem's data

We consider a bounded domain D in $\mathbb{R}^d$ and a given right-hand side and target solutions $f, u_0 \in H^1_{loc}\left(\mathbb{R}^d\right)$. The objective is to find a connected domain Ω_γ with boundary γ which is strictly included in D and which minimizes the cost

E. Laporte et al., *Numerical Methods in Sensitivity Analysis and Shape Optimization*

function

$$j(\gamma) = J(\gamma, u_\gamma) = c \int_{\Omega_\gamma} (u_\gamma - u_0)^2 \, dx,$$

where u_γ is the unique solution of the isotropic diffusion equation

$$\begin{cases} -\Delta u_\gamma := -\sum_{i=1}^{d} \frac{\partial^2 u_\gamma}{\partial x_i^2} = f & \text{in } \Omega_\gamma, \\ u_\gamma = 0 & \text{on } \partial\Omega = \gamma. \end{cases} \tag{7.1}$$

The minimization is to be implemented under the two design constraints

- the volume of Ω_γ is bounded from below $vol(\Omega_\gamma) \geq vol_{min}$;
- the domain is inside a box $\Omega_{min} \subset \Omega_\gamma \subset \Omega_{max}$.

The external contour γ of the unknown domain Ω_γ (see Figure 7.1) is parametrized by splines as described in Chapter 1, §1.7. For a given vector of control variables $z = (z_i)_{i=1,\ldots,n}$, $n \in \mathbb{N}$, the curve $\gamma(z,t)$, $t \in [0,2]$ is thus defined by

$$\gamma(z,t) = \hat{\gamma}(t) + \left(\sum_{i=1}^{n-1} \sum_{j=1}^{4} c_{j,i}(z) f_{j,i}(t) \right) \underline{\hat{n}}(t). \tag{7.2}$$

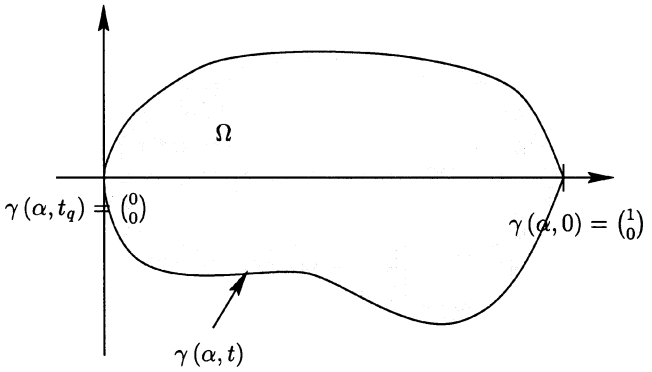


Figure 7.1. A typical domain Ω_γ

In the following examples, the reference curve corresponds to an NACA wing profile

$$\hat{\gamma}(t) = \begin{pmatrix} (t-1)^2 \\ 0.13t\,(t-1)\,(t-2) \end{pmatrix}, \quad t \in [0,2[,$$

interpolated with $n = 16$ control points (see Figure 7.1) defined as indicated in *Tables 7.1, 7.2.* In this case, we have $\lim_{t \to 2} \hat{\gamma}(t) = \hat{\gamma}(0)$, and the normal vector is given by

$$\underline{\hat{n}}(t) = \frac{d\hat{\gamma}}{dt}(t) = \begin{pmatrix} 0.13\,(3t^2 - 6t + 2) \\ -2\,(t-1) \end{pmatrix}, \forall t \in]0,2[.$$

Shapes associated to different values of the parameters z are represented in Figure 7.2. The other data are $c = 0,4354.10^{-6}$,

$$u_0(x,y) = y^2 - 0.0169x(x-1)^2,$$

and

$$f(x,y) = 1.014 \times x - 2.676.$$

For this set of data, the reference curve $\gamma = \hat{\gamma}$ is a solution of the optimization problem since the target state is a solution of the boundary value problem

$$-\Delta u_0 = f \text{ on } \Omega_{\hat{\gamma}},$$
$$u_0 = 0 \text{ on } \hat{\gamma}.$$

The corresponding value of the control parameters is thus $z = 0$ yielding $j = 0$, the trivial degenerate solution $\Omega_\gamma = \emptyset$ being avoided because of the volume constraint imposed on Ω_γ.

i	1	2	3	4
t_i	0	$5.83493.10^{-2}$	0.2020366	0.2866862
i	5	6	7	8
t_i	0.3994241	0.5404287	0.6815434	0.8967789
i	9	10	11	12
t_i	1	1.14522	1.37125	1.56399
i	13	14	15	16
t_i	1.67775	1.88577	1.9701	2

Table 7.1. Values of the control abscissa t_i

As seen in *Table 7.2*, the parameter number 8 must lie between two identical bounds, meaning that the position of the leading edge should not change during the optimization process. This double inequality constraint will be treated as a shape equality constraint. Altogether, the design constraints acting on the shape are

- one inequality constraint on the volume;
- 15 lower box constraints on the parameters $z_i, i \neq 8$;
- 15 upper box constraints on the parameters $z_i, i \neq 8$;
- one equality constraint on the eighth parameter.

Discrete Solvers and cost evaluations

A finite element grid $\underline{X}$ of the domain Ω_γ is built or is updated at each step of the algorithm, that is for each value of the parameters z. Thus, the state equation (7.1) can be discretized in space by a standard finite element method

Parameter number	1	2	3
Initial shape	$1.15196.10^{-2}$	$3.62869.10^{-2}$	$3.31756.10^{-2}$
Ω_{min}	$-2.77876.10^{-3}$	$-2.2.10^{-2}$	-3.10^{-2}
Ω_{max}	$1.91522.10^{-2}$	$5.86892.10^{-2}$	$7.11341.10^{-2}$
Parameter number	4	5	6
Initial shape	$-1.43061.10^{-2}$	$1.42697.10^{-3}$	$1.74912.10^{-2}$
Ω_{min}	$-3.6.10^{-2}$	$-3.5.10^{-2}$	$-2.7.10^{-2}$
Ω_{max}	$7.17145.10^{-2}$	$5.69501.10^{-2}$	$3.61135.10^{-2}$
Parameter number	7	8	9
Initial shape	$6.84191.10^{-3}$	0	$1.04588.10^{-2}$
Ω_{min}	$-0.9.10^{-2}$	0	-10^{-2}
Ω_{max}	$8.41302.10^{-3}$	0	$1.4787.10^{-2}$
Parameter number	10	11	12
Initial shape	$3.25779.10^{-2}$	$-2.6695.10^{-2}$	$-2.27616.10^{-5}$
Ω_{min}	$-2.5.10^{-2}$	$-3.7.10^{-2}$	-4.10^{-2}
Ω_{max}	$4.56504.10^{-2}$	$6.63305.10^{-2}$	$6.42832.10^{-2}$
Parameter number	13	14	15
Initial shape	$1.22867.10^{-2}$	$-1.71519.10^{-3}$	0.1527829
Ω_{min}	-3.10^{-2}	$-1.25673.10^{-2}$	$-1.56241.10^{-3}$
Ω_{max}	$2.94982.10^{-2}$	$8.07421.10^{-3}$	3.258273
Parameter number	16		
Initial shape	0.1215625		
Ω_{min}	0.4966996		
Ω_{max}	-0.32089		

Table 7.2. Initial values of the control parameters z

using first-order P_1-Lagrange finite elements. Denoting by $K(\underline{X})$ the associated stiffness matrix

$$K(\underline{X})_{ij} = \int_\Omega \frac{\partial \varphi_i}{\partial x} \cdot \frac{\partial \varphi_j}{\partial x},$$

the state equation reduces to the linear system of order p

$$(\mathcal{P}_h) \quad K(\underline{X}) \cdot W = b(\underline{X}),$$

which we write as

$$f(\underline{X}, W) = 0,$$

under the notation

$$f(\underline{X}, W) = K(\underline{X}) \cdot W - b(\underline{X}).$$

This matrix system (P_h) could be solved by any available sparse direct solver. In our tests, we have chosen to solve it by a preconditioned conjugate gradient algorithm.

Using the classical formula (6.7) based on the adjoint state, the gradient of the cost function j with respect to the mesh (computational grid) $\underline{X}$ is given

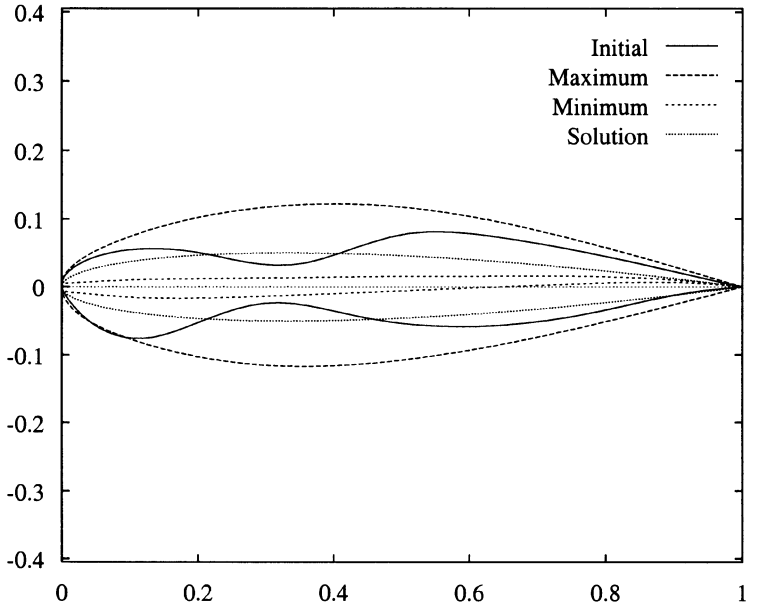


Figure 7.2. Initial, maximal, minimal and solution shapes

by

$$\frac{dj}{d\underline{X}}(\underline{X}) = \frac{\partial J}{\partial \underline{X}}(\underline{X}, W) + \left(\frac{\partial f}{\partial \underline{X}}(\underline{X}, W)\right)^t \cdot P, \tag{7.3}$$

where the adjoint state P satisfies the adjoint state equation

$$\left(\frac{\partial f}{\partial W}(\underline{X}, W)\right)^t \cdot P = -\frac{\partial J}{\partial W}(\underline{X}, W).$$

From the symmetry of the stiffness matrix $K(\underline{X})$, this adjoint equation reduces to

$$(\mathcal{P}_h^*) \quad K(\underline{X}) \cdot P = -\frac{\partial J}{\partial W}(\underline{X}, W). \tag{7.4}$$

This adjoint system $(\mathcal{P}_h^*)$ is solved using the same preconditioned conjugate gradient algorithm as used for the direct system $(\mathcal{P}_h)$. Finally, the subroutines computing $\frac{\partial J}{\partial \underline{X}}(\underline{X}, W)$, $\frac{\partial J}{\partial W}(\underline{X}, W)$ and $\left(\frac{\partial f}{\partial \underline{X}}(\underline{X}, W)\right)^t$ are obtained by automatic differentiation using O∂yssée in adjoint mode (see Chapter 5). The global procedure for computing the gradient of the cost function is then as described at the end of Chapter 6.

The last numerical tool to be specified is the mesh deformation algorithm. In this example, the mesh deformations will be rather large, and therefore we have used the sophisticated (and expensive) formula (1.8). On the other hand,

the values of the grid gradient $\frac{dJ}{d\underline{X}}$ at the internal nodes are very small (the cost function is almost insensitive to the position of the internal nodes), and therefore in the formula

$$\frac{dj}{dz} = \left(\frac{\partial \underline{X}}{\partial z}\right)^t \cdot \frac{dJ}{d\underline{X}},$$

the exact values of the grid derivatives $\frac{\partial \underline{X}}{\partial z}$ at the internal nodes can be replaced by simpler, cheaper approximations. Three techniques will in fact be compared. The simpler, denoted $\mathcal{R}_1$, ignores all internal derivatives, which means that it does not differentiate the cost function with respect to the position of the internal grid points. This amounts to setting

$$\frac{\partial \underline{X}}{\partial z}(\underline{x}_i) = 0, \text{ for all internal nodes } i.$$

The most sophisticated, $\mathcal{R}_2$, does compute the exact grid derivatives $\frac{\partial \underline{X}}{\partial z}$ associated to (1.8) and the corresponding exact value of the gradient of j. The last one, $\mathcal{R}_3$, replaces the grid derivatives $\frac{\partial \underline{X}}{\partial z}$ at the internal nodes by a simplified expression inspired from the techniques described in Section 1.5.2 (see [31, 32] for more details).

We will also compare in our tests three different strategies for defining the mesh of the different configurations:

1. The first strategy is the one described in Chapter 1. It constructs the initial mesh of the initial reference configuration $\hat{\Omega}$ and then deforms this mesh by the sophisticated formula (1.8). This reduces the shape optimization problem to a true finite dimensional constrained minimization problem, but the result is sensitive to the choice of the first computational grid, and the algorithm may fail to produce admissible grids when the shape deformation is too large;

2. The second strategy constructs a new mesh after each update of the parameter z, that is after each update of the contour geometry γ. This authorizes very large shape deformations. On the other hand, the strategy is still grid sensitive, and the definition of the discrete cost function $j(z)$ changes slightly at each iteration since the map $\underline{X}(z)$ is now of the form $\underline{X} = \varphi_h(z, \underline{X}^k)$;

3. The last strategy uses two meshes at each iteration. A first mesh is used to compute estimates of the direct and of the adjoint state. A mesh adaption algorithm is then used, based on these first estimates of *both the state and the adjoint state.* Updated values of the state and adjoint state vectors, of the cost function j, and of its gradients $\frac{dj}{d\underline{X}}$ are computed on this adapted mesh. This last strategy authorizes very large shape deformations, and is not grid sensitive. On the other hand, it is more expensive, and the definition of the discrete cost function $j(z)$ again changes slightly at each iteration. Nevertheless, these changes should be of lower order since the discrete cost function is supposed to be a more accurate approximation of a well-defined continuous cost function $J(\gamma(z))$.

By combining the different approximations of the grid derivatives, and the different strategies of grid construction, we get nine variants for any test problem. These variants are numbered as indicated in *Table 7.3.*

		Lifting		
		$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{R}_3$
Grid	(i)	1	2	3
Construction	(ii)	4	5	6
Algorithm	(iii)	7	8	9

Table 7.3. Numbers of the different variants

Numerical results

For simplicity, we will not present below the results obtained by the original algorithm (i), corresponding to the variants 1,2 and 3. Indeed, in this particular case, because of the large shape deformations of the body between the initial and the final configuration, the grid deformation algorithm fails to produce an admissible grid sooner or later in the updating process. Then, the state equation can no longer be solved, and the computer programme must be stopped.

The first results of *Table 7.4* compare the averaged CPU times, as observed on an HP-UX B.10.20 9000/800 work station. As expected, algorithm (iii) computing two direct and two adjoint states at each iteration is twice as expensive as algorithm (ii). Also, in this simple situation, the cost of using an exact evaluation of the grid derivative (variant $\mathcal{R}_2$) is significant. It will not be so for more complex state equations, where the cost of $\mathcal{R}_2$ becomes negligible compared to the solution of the state and of the adjoint state equation.

		Lifting		
		$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{R}_3$
Grid	(ii)	0.44	1.32	0.43
Construction	(iii)	1.16	3.06	1.19

Table 7.4. Averaged CPU time by iteration in the different variants

We now turn to the performance evaluation of the interior point algorithm. In all six cases, the algorithm has stopped very close to the exact minimum, by failure of the line search algorithm (no significant decrease of the cost function after ten iterations of line search). As indicated in Figures 7.3, 7.4, the performance of the algorithm is quite insensitive to the discretization variant. Grid adaptation gives nevertheless a lower value of the cost function. This is not a surprise. Next to the exact solution, the gradient is very small and a more accurate strategy gives a better approximation of this gradient, and thus can go closer to the exact minimum. The correct heuristics is therefore to use an approximate gradient far from convergence and to use more accurate gradients close to convergence.

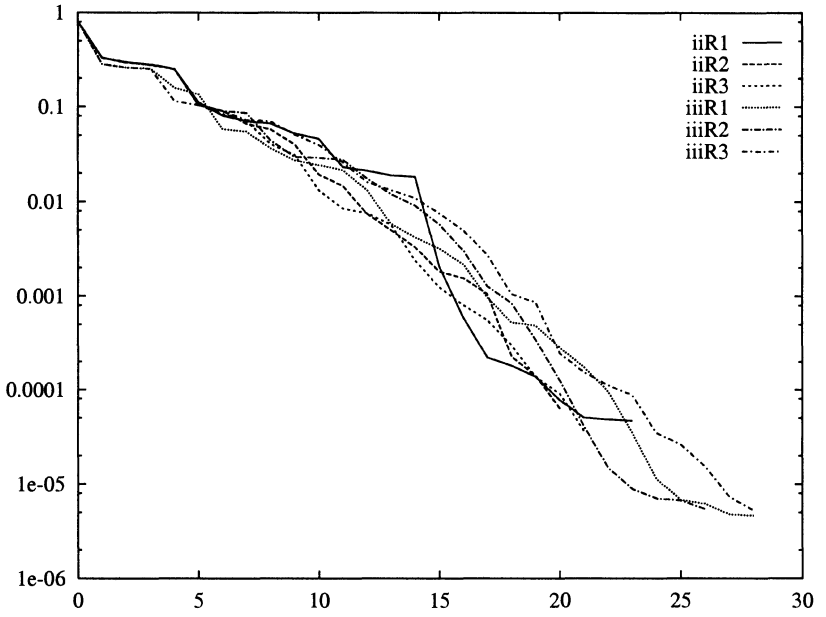


Figure 7.3. Evaluation of the cost function during the iterations of the interior point algorithm

Although all variants lead to similar results, the best results (at least in terms of Lagrangian norms) seem to be obtained in variant 8 using the exact gradient derivatives $\mathcal{R}_2$ and the mesh adaption strategy (iii). The corresponding values of the parameters at convergence are given in *Table 7.5.* The position parameters $z_i, i \neq 15$ and $i \neq 16$ are very close to their exact values. The parameters z_{15} and z_{16} are further away. This means that the value of the trailing edge angle has very little effect on the cost function. In other words, the optimization problem is poorly conditioned with respect to these two last parameters.

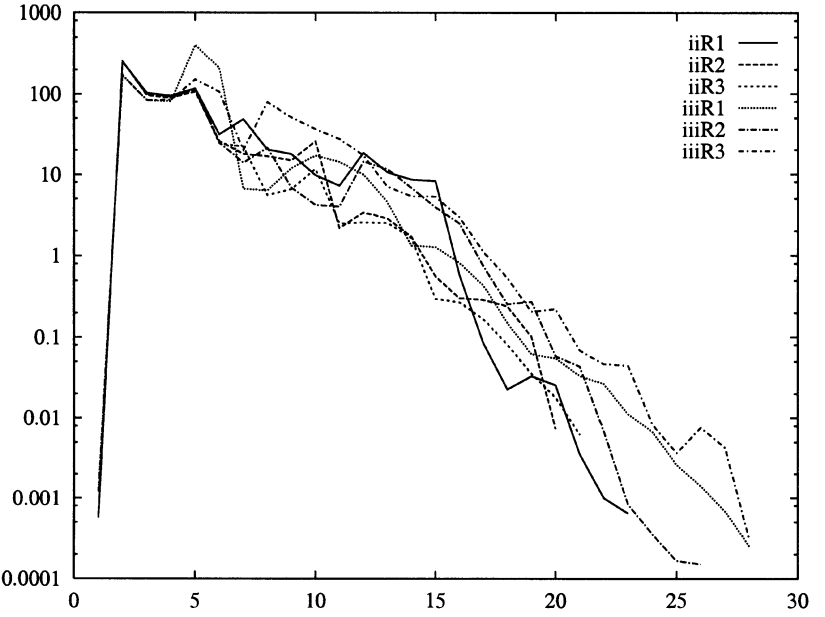


Figure 7.4. Evaluation of the norm squared of the Lagrangian during the iterations of the interior point algorithm

parameter number	1	2	3
Result 8	$1.58114.10^{-3}$	$6.10807.10^{-5}$	$-5.24360.10^{-5}$
parameter number	4	5	6
Result 8	$-6.99330.10^{-6}$	$-2.57035.10^{-5}$	$4.53073.10^{-5}$
parameter number	7	8	9
Result 8	$9.99388.10^{-6}$	$5.27650.10^{-19}$	$2.83577.10^{-6}$
parameter number	10	11	12
Result 8	$9.20594.10^{-6}$	$-2.38088.10^{-5}$	$-3.45927.10^{-5}$
parameter number	13	14	15
Result 8	$-8.15454.10^{-5}$	$-8.07493.10^{-4}$	0.20988
parameter number	16		
Result 8	0.12533		

Table 7.5. Values of the control parameters at convergence for the variant 8

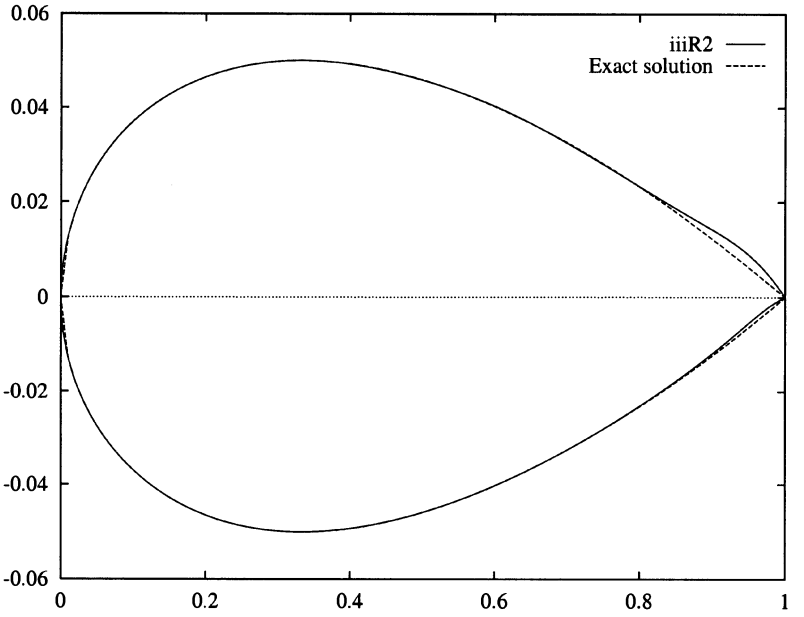


Figure 7.5. Comparing the result of test case 8 with the exact solution

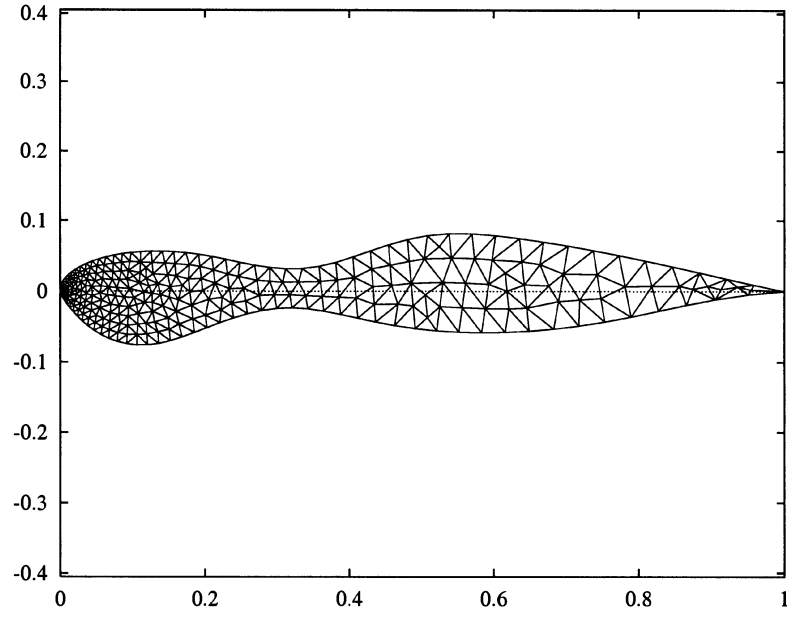


Figure 7.6. First mesh inside the initial shape

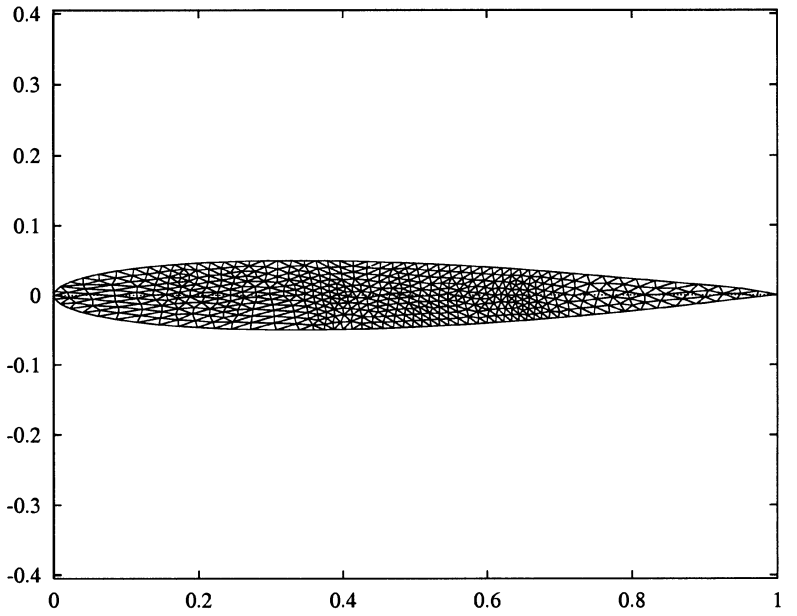


Figure 7.7. Adapted mesh at convergence in variant 8

7.3 Shape optimization in aerodynamics

7.3.1 Navier–Stokes equations

The Navier–Stokes equations governing the three-dimensional flow of a homogeneous perfect compressible Newtonian fluid relate its density ρ, its velocity $\underline{u} = \left(u^1, u^2, u^3\right)$, its temperature T, its total energy $E = T + \frac{\|\underline{u}\|^2}{2}$ and its pressure $p = (\gamma - 1)\rho T$ by (see [7, 43]):

- the mass conservation equation

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \underline{u}) = 0; \tag{7.5}$$

- the momentum conservation equation

$$\frac{\partial \rho \underline{u}}{\partial t} + \nabla \cdot (\rho \underline{u} \otimes \underline{u}) + \nabla p = \underline{f} + \operatorname{div}\left[\eta \left(\nabla \underline{u} + \nabla^t \underline{u}\right) + \left(\xi - \frac{2}{3}\eta\right) \nabla \cdot \underline{u}\, \mathbf{I}\right]; \tag{7.6}$$

- the energy conservation equation

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot ((\rho E + p)\, u) = \underline{f} \cdot \underline{u} + \nabla \cdot (\kappa \nabla T)$$
$$+ \left[\eta \left(\nabla \underline{u} + \nabla^t \underline{u} \right) + \left(\xi - \frac{2}{3} \eta \right) \nabla \cdot \underline{u}\, \mathbf{I} \right] : \nabla \underline{u}. \tag{7.7}$$

In these equations, η and ξ denote the first and the second viscosity of the fluid (the latter being negligible in general), $\underline{f}$ represents the external body forces exerted on the fluid, $\kappa \nabla T$ is the heat flux vector and $\mathbf{I}$ denotes the unit tensor in $\mathbb{R}^3$.

Introducing the vector of conservative variables

$$W = \begin{pmatrix} \rho \\ \rho u^1 \\ \rho u^2 \\ \rho u^3 \\ \rho E \end{pmatrix},$$

the inviscid flux vector

$$F(W) = \begin{pmatrix} F_1(W) \\ F_2(W) \\ F_3(W) \end{pmatrix}$$

with

$$F_i(W) = \begin{pmatrix} \rho u^i \\ \rho u^1 u^i + \delta_{1i} p \\ \rho u^2 u^i + \delta_{2i} p \\ \rho u^3 u^i + \delta_{3i} p \\ u^i (E + p) \end{pmatrix} \quad \text{for } i = 1, \ldots, 3,$$

and a similar notation for the viscous part $K(W, \nabla W)$ of the flux, the Navier–Stokes system constructed with equations (7.5), (7.6) and (7.7) takes the abstract divergence form

$$\frac{\partial W}{\partial t} + \nabla \cdot F(W) - \nabla \cdot K(W, \nabla W) = S(W). \tag{7.8}$$

In component form, we have

$$\frac{\partial W^i}{\partial t} + \sum_{j=1}^{3} \frac{\partial F_j^i(W)}{\partial x_j} - \sum_{j=1}^{3} \frac{\partial K_j^i(W, \nabla W)}{\partial x_j} = S^i(W), \forall i = 1, 5.$$

7.3.2 Euler's equations

If we neglect all viscous effects, that is the thermic diffusion and the viscous stresses by setting the conductivity κ and the fluid viscosities η and ξ to zero, the Navier–Stokes system of equations (7.5), (7.6) and (7.7) reduce to the Euler equations

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \underline{u}) = 0, \tag{7.9}$$

$$\frac{\partial \rho \underline{u}}{\partial t} + \nabla \cdot (\rho \underline{u} \otimes \underline{u}) + \nabla p = \underline{f}, \tag{7.10}$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot ((\rho E + p)\,\underline{u}) = \underline{f} \cdot \underline{u}. \tag{7.11}$$

In abstract form and under the above notation, the Euler equations take the simple divergence form

$$\frac{\partial W}{\partial t} + \nabla \cdot F(W) = S(W).$$

7.3.3 NSC2KE solver

This section describes the numerical solver NSC2KE developed among others by B. Mohammadi S. Lantéri (see [36]) for the solution of the bidimensional compressible Navier–Stokes or Euler's equations.

Let Ω_h be the computational domain around the considered profile, endowed with the unstructured triangular grid $\Omega_h = \bigcup_j T_j$. With this grid, we also construct a partition of the computational domain in node-centered cells $\Omega_h = \bigcup_j C_j$, the cell C_j centered at node j being obtained by cutting each edge originating from j in two, and by attaching to the cell the part of the domain containing the node j. This construction is detailed in Figure 7.8 which is well known by NSC2KE users.

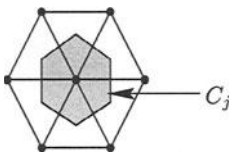


Figure 7.8. Construction of the computational cell C_j associated to node j

To construct the space discretization of the divergence equation (7.8), we first integrate this equation on the cell C_j, which yields a system of the form

$$\int_{C_j} \frac{\partial W}{\partial t} + \int_{\partial C_j} F(W) \cdot \underline{n} = *, \forall j.$$

We next introduce two different approximations W_h and W_h' of the vector field W of conservative variables, both characterized by the set of nodal values $(W_j)_j$ of W. The first approximation defines W_h on each triangle by a standard linear interpolation. It is used to reconstruct the gradients of W and to compute the viscous fluxes $\int_{\partial C_j} K(W, \nabla W) \cdot \underline{n}$ by a centered scheme. The second one W_h' takes constant values on each cell C_j,

$$W_h' \,|_{C_j} = W_j.$$

It is used to reconstruct the inviscid flux by a formula of the type

$$\int_{\partial C_j} F(W_h') \cdot \underline{n} \approx \sum_{i \neq j} \Phi\left(W_h'|_{C_j}, W_h'|_{C_i}, \underline{n}\right) \int_{\partial C_j \cap C_i} \underline{n}. \tag{7.12}$$

Above, $\Phi(W_1, W_2, \underline{n})$ can be any reasonable numerical flux function of the type

$$\Phi(W_1, W_2, \underline{n}) = \frac{1}{2}\left(F(W_1) + F(W_2)\right) - d(W_1, W_2, \underline{n}),$$

which satisfies the consistency condition

$$\Phi(W, W, \underline{n}) = F(W).$$

The numerical diffusion operator d which appears in the numerical flux function must be adapted to the physical characteristics of the problem under study. In the software **NSC2KE**, the user defines this numerical diffusion operator by choosing between a Roe scheme, an Osher scheme or an HUS kinetic scheme (see [36, 38]).

The above formulation is only first-order in space. Second-order accuracy can be achieved by replacing in the flux functions of (7.12) the piecewise constants $W_h'|_{C_j}$ by a more sophisticated MUSCL interpolation. In this construction, if ∇W_i is a given approximation of the gradient of W at node i obtained by averaging the local gradients of W_h on all triangles containing i, external and internal values of the vector W at the middle of edge $\underline{ij}$ are given by the weighted upwind interpolations

$$W_{ij} = W_i + 0,5\,\mathrm{Lim}\left(\beta\left(\nabla W\right)_i \cdot \underline{ij}, (1-\beta)\left(W_i - W_j\right)\right)$$

and

$$W_{ji} = W_j + 0,5\,\mathrm{Lim}\left(\beta\left(\nabla W\right)_j \cdot \underline{ji}, (1-\beta)\left(W_j - W_i\right)\right).$$

Above, Lim is a slope limiter such as proposed in Van Albada [16],

$$\mathrm{Lim}(a,b) = 0,5\left(1 + \mathrm{sgn}(ab)\right)\frac{\left(a^2+\alpha\right)b + \left(b^2+\alpha\right)a}{a^2+b^2+2\alpha},$$

with $0 < \alpha \ll 1$ a given regularization coefficient. The coefficient β defines the amount of upwinding to be introduced at each node. Second-order accuracy in smooth regions is then simply obtained by replacing in the flux function of (7.12) the piecewise constants W_i' and W_j' by the interpolated values W_{ij} and W_{ji}, leading to the new system of equations

$$\int_{C_j} \frac{\partial W_{h,j}}{\partial t} + \sum_i \int_{\partial C_j \cap \partial C_i} \Phi\left(W_{ij}, W_{ji}, \underline{n}\right) \cdot \underline{n} = *, \tag{7.13}$$

the viscous fluxes appearing in $*$ being computed by a standard centered scheme.

The space discretized equation (7.13) takes finally the abstract form

$$\frac{\partial W_h}{\partial t} = \mathcal{F}\left(W_h(t)\right). \tag{7.14}$$

This evolution equation is then integrated in time by an explicit four stages Runge–Kutta scheme [30]:

$$\begin{cases} W_h^{n,0} = W_h^n, \\ W_h^{n,k} = W_h^{n,0} + \alpha_k \Delta t \mathcal{F}\left(W_h^{n,k-1}\right), k = 1,4, \\ W_h^{n+1} = W_h^{n,4}, \end{cases} \tag{7.15}$$

whose optimal coefficients α_k are given by

$$\alpha_1 = 0.11,\ \alpha_2 = 0.2766,\ \alpha_3 = 0.5,\ \alpha_4 = 1.0.$$

The above scheme is stable under the local CFL condition

$$\Delta t_i \leq \frac{\Delta x_i}{|u| + c} \tag{7.16}$$

in the inviscid case and

$$\Delta t_i \leq \min\left(\frac{\Delta x_i}{|u| + c}, \frac{\rho Pr \Delta x_i^2}{2\left(\mu + \mu_t\right)}\right) \tag{7.17}$$

in the viscous case. Above, Δx_i is the minimal height among all triangles containing the node i. When computing unsteady flows, the time step must be identical for all nodes i. In such cases, we choose the minimum of all the nodal values obtained in (7.16) or in (7.17)

$$\Delta t = \min_i \Delta t_i.$$

When computing steady flows as the limit in time of pseudo-unsteady flows, the time step can take different local values. In such cases, we choose at each node the maximal value allowed by (7.16) or (7.17). In both cases, after time discretization of the space discrete problem (7.14), the numerical integration of the Navier–Stokes equations reduces to the explicit induction formula

$$W_{h,j}^{n+1} = W_{h,j}^n + \frac{\Delta t_j}{|C_j|} F_{h,j}\left(W_h^n\right), \forall j. \tag{7.18}$$

For unsteady problems, this scheme is integrated on a given time interval $(0, T_{max})$. For computing steady flows solutions of the stationary equations

$$F_{h,j}\left(W_h^n\right) = 0, \forall j, \tag{7.19}$$

equation (7.18) is integrated up to a point where the relative residual $\frac{\|F_h(W_h^n)\|}{\|F_h(W_h^0)\|}$ has sufficiently decreased.

More details on this time integration of the compressible Navier–Stokes equations, including the calculation of axisymmetric flows can be found in [36].

7.3.4 Input-Output data files

The sofware NSC2KE uses two data files: the first one, named MESH contains the grid description using the *amdba* format introduced in Chapter 1. The second data file, named DATA contains all physical and numerical parameters to be used in the calculation, as described in the following example. The lines between two data are comments which must be deleted in an actual run.

```
0          --> =0 2d, =1 axisymmetric
0          --> =0 Euler, =1 Navier-Stokes
1.e5       --> Reynolds by meter (the mesh is given in meter)
0.         --> inverse of Froude number (=0 no gravity)
0.85       --> inflow Mach number
1.         --> ratio pout/pin
1          --> wall =1 newmann b.c. on the temp.(adiabatic wall),
                   =2 Dirichlet.(isothermal wall)
300.       --> inflow temperature (in Kelvin) for Sutherland laws
288.       --> if isothermal walls , wall temperature (in Kelvin)
0.0        --> angle of attack
1          --> Euler fluxes =1 roe, =2 osher,=3 kinetic
3          --> nordre =1 first order scheme, =2 second order,
                     =3 limited second order
0          --> =0 global time steping (unsteady), =1 local Euler,
               =2 local N.S.
1.5        --> cfl
40000      --> number of time step
200        --> frequence for the  solution to be saved
1.e10      --> maximum physical time for run
               (for unsteady problems)
1.e-11     --> final value of the residual (for steady problems)
0          --> =0 start with uniform solution,
               =1 restart from INIT_NS
```

When dealing with turbulent flows, the DATA file must be complemented by an empty line and by the following data:

```
0          --> =0 no turbulence model, =1 k-epsilon model
0          --> =0 two-layer technique, =1 wall laws
1.e-2      --> delta in wall laws or limit of the one-eq. model.
               (in meter)
0          --> =0 start from uniform solution for k-epsilon,
               =1 from INIT_KE
-1.e10 1.e10 -1.e10 1.e10      --> xtmin,xtmax,ytmin,ytmax
                                   (BOX for k-epsilon r.h.s)
```

The output files contain a stored solution which can be used in a restart and the following curves or isolines which can be displayed by GNUPLOT [1].

[1] GNUPLOT is a graphic tool developed by Thomas Williams and Collin Kelley. For more information write to info-gnuplot@dartmouth.edu

- `GNU.MACH`: iso-Mach;
- `GNU.MESH`: mesh;
- `GNU.PRES`: isobars;
- `GNU.TEMP`: iso-temperatures;
- `GNU.VECT`: velocity fields;
- `RESIDUAL`: evolution of the residual in funtion of the time step;
- `WALL.DATA`: pressure coefficient on the body.

A typical cost function

A typical cost function in aerodynamics is the pressure drag C_D obtained by integrating the axial component of the action of the pressure on the body:

$$C_D = \frac{2}{\rho_\infty u_\infty} \int_\gamma (p - p_\infty)\, \underline{n}.\underline{e}_\infty \, d\gamma.$$

Here

- ρ_∞, u_∞ et p_∞ are the density, velocity norm and pressure of the fluid at farfield,
- $\underline{n}$ is the inwards unit normal vector to the contour γ,
- $\underline{e}_\infty$ is the unit vector along the flow direction at infinity. Its angle is in fact the angle of attack α specified by the user in the input `DATA` file,
- p is the local value of the pressure when the flow has reached its steady regime (7.19).

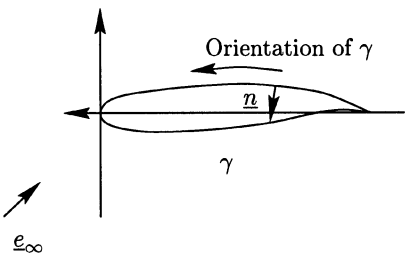


Figure 7.9. Flow configuration

Since the pressure used in `NSC2` is the normalized pressure

$$\bar{p} = \frac{p}{\rho_\infty u_\infty},$$

and since we have the standard identity

$$\int_\gamma \underline{n}\, d\gamma = \underline{0},$$

the pressure drag can be written under the equivalent form

$$C_D = 2 \int_\gamma \bar{p}\underline{n} \cdot \vec{e}_\infty \, d\gamma.$$

It can be split into horizontal and vertical components

$$C_T = 2 \int_\gamma \bar{p}\underline{n} \cdot \underline{e}_x \, d\gamma, \qquad C_N = 2 \int_\gamma \bar{p}\underline{n} \cdot \underline{e}_y \, d\gamma.$$

Furthermore, since the contour γ has been replaced in all calculations by the straight edges γ_i of the boundary triangles, the first integral can also be written

$$2 \int_{\gamma_i} \bar{p}\underline{n} \cdot \underline{e}_x \, d\gamma = 2 \left(\int_{\gamma_i} \bar{p} \, d\gamma \right) \underline{n}_i \cdot \underline{e}_x$$

with $\underline{n}_i$ denoting the inwards unit normal vector to the edge γ_i. For piecewise linear pressure fields $\bar{p}$ obtained from the approximate solution field W_h, this integral reduces to

$$2 \int_{\gamma_i} \bar{p}\underline{n} \cdot \underline{e}_x \, d\gamma = 2 \, |\gamma_i| \times 0.5 \, (\bar{p}_{i1} + \bar{p}_{i2}) \, \underline{n}_i \cdot \underline{e}_x = (\bar{p}_{i1} + \bar{p}_{i2}) \times |\gamma_i| \, \underline{n}_i \cdot \underline{e}_x$$

with $\bar{p}_{i1}$ and $\bar{p}_{i2}$ denoting the nodal values of the pressure $\bar{p}$ at the two extremities of the edge γ_i. Using the orientation described in Figure 7.9, the coordinates of the unit normal vector $\underline{n}_i$ take the form (Figure 7.10)

$$\underline{n}_i = \frac{1}{|\gamma_i|} \begin{pmatrix} y_{i1} - y_{i2} \\ x_{i2} - x_{i1} \end{pmatrix},$$

from which we deduce the final form of the first integral in the pressure drag

$$C_T = \sum_i (\bar{p}_{i1} + \bar{p}_{i2}) \times (y_{i1} - y_{i2}) \, .$$

The expression of the second integral is completely similar. An example of a subroutine computing the above pressure drag is given in Appendix D.1 and in the enclosed CD-ROM.

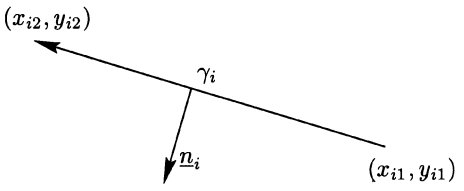


Figure 7.10. Geometric configuration and orientation of a boundary edge

Computing the adjoint state

We can now proceed to the calculation of the gradient of this cost function by following step by step the detailed procedure introduced at the end of Chapter 6. The direct solver has just been described. The problem here is to compute the adjoint state. In a steady regime, this direct state W_h is the solution of the steady equation (7.19) that we get by integrating in time the pseudo-unsteady solution $W_h(t)$ of the weighted evolution problem

$$|C_j| \frac{\partial W_{h,j}(t)}{\partial t} - F_{h,j}\left(W_h(t)\right) = 0, \forall j.$$

By construction, the adjoint state P_h is then a solution of the linear adjoint state equation

$$\frac{\partial F_{h,j}\left(W_h\right)^t}{\partial W_h} \cdot P_h + \frac{\partial J(\underline{X}, W_h)}{\partial W_{h,j}} = 0, \forall j, \tag{7.20}$$

and corresponds to the limit for large pseudo-times of the unsteady solution $P_h(\tau)$ of the adjoint evolution problem

$$-|C_j| \frac{\partial P_{h,j}(\tau)}{\partial \tau} + \frac{\partial F_{h,j}\left(W_h\right)^t}{\partial W_h} \cdot P_h + \frac{\partial J(\underline{X}, W_h)}{\partial W_{h,j}} = 0, \forall j.$$

The pseudo-time τ here is an auxiliary variable which will go backward with respect to the real physical time ($\tau = -t$, see the adjoint state calculation introduced in (7.4).

We can therefore obtain the adjoint state P_h by an explicit time integration of the above evolution problem. The iterative scheme for computing P_h corresponds then to the induction formula

$$P_{h,j}^{n+1} = P_{h,j}^n + \frac{\Delta t_j}{|C_j|}\left(\frac{\partial F_{h,j}\left(W_h\right)^t}{\partial W_h} \cdot P_h^n + \frac{\partial J(\underline{X}, W_h)}{\partial W_{h,j}}\right),$$

which is stopped when the adjoint state P_h^n has reached a steady state. In this induction, the term

$$\frac{\partial F_{h,j}\left(W_h\right)^t}{\partial W_h} \cdot P$$

can be obtained by automatic differentiation in adjoint mode of the software computing the numerical fluxes F_h. The application of this strategy to the solution of the adjoint state problem is further detailed in Appendix D.2.

We show in Figures 7.11, 7.12 (with a vertical log scale) convergence examples of adjoint problems obtained on the mesh of Figure 7.13, for viscous or inviscid flows, with various Mach numbers. In all cases, the angle of attack is zero, and a Roe-type scheme was used. In the viscous case, the Reynolds number was 0.85, and a k-ε model and a wall law were used (cf. [36]). For each case, the convergence (norm of the residual with respect to the iteration number) of the state equation and two adjoint state equation (one for the drag and one for the lift) are plotted.

The figures show that the speeds of convergence of the state equation and of the adjoint state equation are of the same order. Nevertheless, remember that as a matter of CPU time, one iteration for the adjoint state equations is more expensive than one for the state equation (cf. (5.4)).

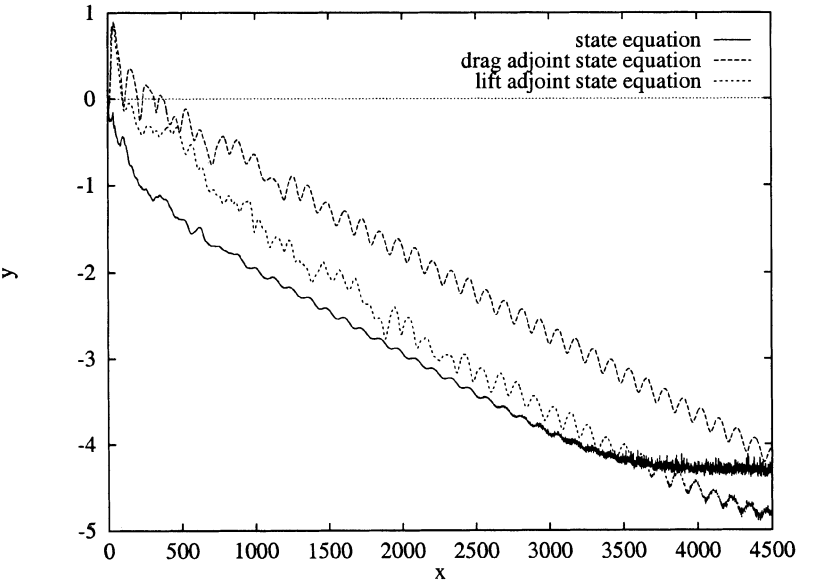


Figure 7.11. Euler, Mach 0.7

This adjoint state integration strategy is simple, but converges rather slowly. In many applications, a symmetric Gauss–Seidel procedure directly applied to the solution of the steady adjoint linear system (7.20) seems to be more efficient.

In any case, once the adjoint state has been computed, the grid gradient vector is simply given by the usual formula

$$\left(\frac{\partial j}{\partial x_i^j}\right)_{\substack{i=1,\dots,ns \\ j=1,2}} = \left[\frac{\partial J}{\partial x_i^j} + \langle P, \frac{\partial F_h}{\partial x_i^j}\rangle\right]_{\substack{i=1,\dots,ns \\ j=1,2}}$$

which has two components (one for each direction) for each node $i = 1, \dots, ns$ of the computational grid. Here, ns is the total number of nodes in the grid. A typical output obtained on the mesh of Figure 7.13 is displayed in Figure 7.14 (gradient of the drag, inviscid computation, Mach 0.85, angle of attack equal to zero).

Observe that again, the gradient $\left(\frac{\partial j}{\partial x_i^j}\right)$ is very much concentrated on the body's surface, which means that we would get a good approximation of this gradient by ignoring the derivatives of the cost function with respect to the

position of the internal grid points. In fact, in the present example and in some similar aerodynamics applications [37], a simpler and very crude approximation of the gradient can be used. Indeed, the contribution $P(z) \cdot \frac{\partial f(\underline{X}, W(z))}{\partial \underline{X}}$ to the gradient of the change of state variables W induced by the change of geometry is often smaller than the direct effect $\frac{\partial J(\underline{X}, W(z))}{\partial \underline{X}}$ of this change of geometry. This leads to the following simple approximation of the gradient

$$\nabla j(z) \approx \frac{\partial J(\underline{X}, W(z))}{\partial \underline{X}} \cdot \frac{d\underline{X}}{dz}.$$

This last approximation, which may be used in optimization or in control applications, does not involve any adjoint state and is therefore very cheap to obtain.

A full numerical example

The results below were obtained while minimizing the pressure drag of a wing flying inside an inviscid flow at Mach number $M_\infty = 0,85$ with zero angle of attack. The design constraints include a minimal volume constraint (the final volume must be at least 95% of the initial volume), lower and upper box constraints on the parameters, the usual equality constraint imposing that the leading edge remains fixed during the optimization, and a minimal lift constraint defined by

$$g_1(\gamma) = 0,95 \times C_L(\gamma) - C_L(\hat{\gamma}) \leq 0.$$

Above, $C_L(\gamma)$ denotes the lift of the present configuration, and $C_L(\hat{\gamma})$ the lift of the initial reference configuration. In this example, the initial and reference configuration is an RAE profile as described in Figure 7.13 and with parameters as defined in *Table 7.7*. The shape deformation is performed along the normal to the contour $\hat{\gamma}$ as indicated in Chapter 1 (see section 1.4.3). The parametrization uses $n = 10$ control nodes, and the values or constraints on the control parameters z are detailed in *Tables 7.6, 7.7*. The different meshes are obtained by deforming the initial mesh by the sophisticated technique of 1.5.3.

i	1	2	3	4	5	6	7	8	9	10
t_i	0	0.25	0.5	0.75	1	1.2	1.4	1.6	1.8	2

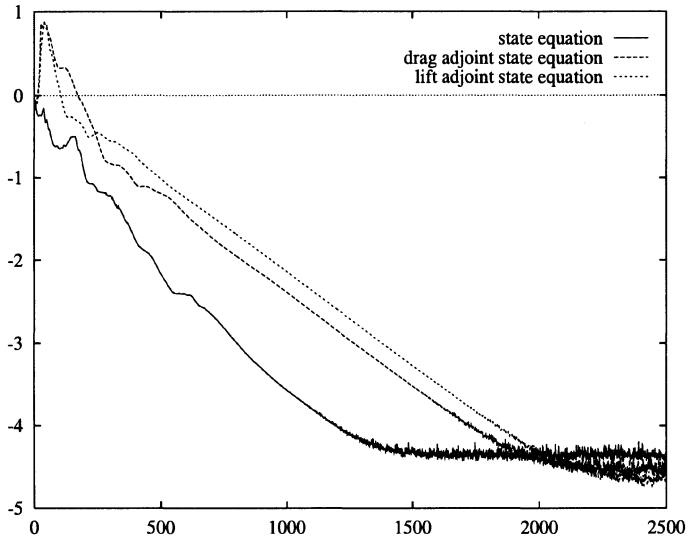
Table 7.6. Abscissa of the control points t_i for the RAE profile

The contour shapes before and after optimization are represented in Figure 7.15 and the associated pressure coefficients maps are represented in Figure 7.16. They confirm that the drag has indeed been reduced during the optimization. To preserve a minimal lift, this drag reduction has been achieved mainly by reducing the intensity of the shock located below the wing (intrados shock). The evolution of the drag and of the lift during the optimization process is displayed in Figure 7.17. We observe a smooth decrease of the drag while the lift

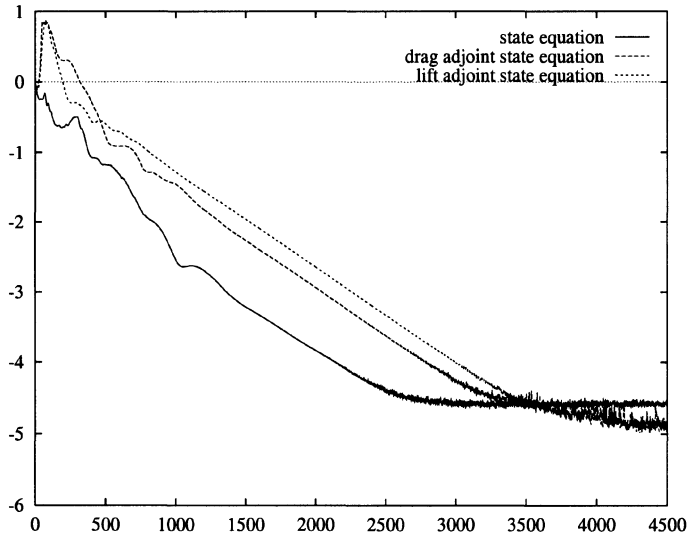
Parameter number	1	2	3	4
Initial values of z	0	0	0	0
Ω_{min}	$-3.5.10^{-2}$	$-5.5.10^{-2}$	-4.10^{-2}	0
Ω_{max}	$3.5.10^{-2}$	$5.5.10^{-2}$	4.10^{-2}	0
Parameter number	5	6	7	8
Initial values of z	0	0	0	0
Ω_{min}	$-3.5.10^{-2}$	-4.10^{-2}	$-3.5.10^{-2}$	-8.10^{-3}
Ω_{max}	$3.5.10^{-2}$	4.10^{-2}	$3.5.10^{-2}$	$1.8.10^{-2}$
Parameter number	9	10		
Initial values of z	0	0		
Ω_{min}	-5.10^{-2}	5.10^{-2}		
Ω_{max}	0.325827	-0.320890		

Table 7.7. Initial values and bounds on the control parameters for the RAE profile

always stays above its lower limit. The results of this section are therefore quite convincing. They are nevertheless quite time consuming. At each iteration, one must solve a full direct and adjoint problem. It turns out that by far the most time consuming part is the calculation of the adjoint state which is roughly 5,5 times more expensive here than the calculation of the direct state. It might be useful here to use approximate adjoint states, but this is a subject of ongoing research.

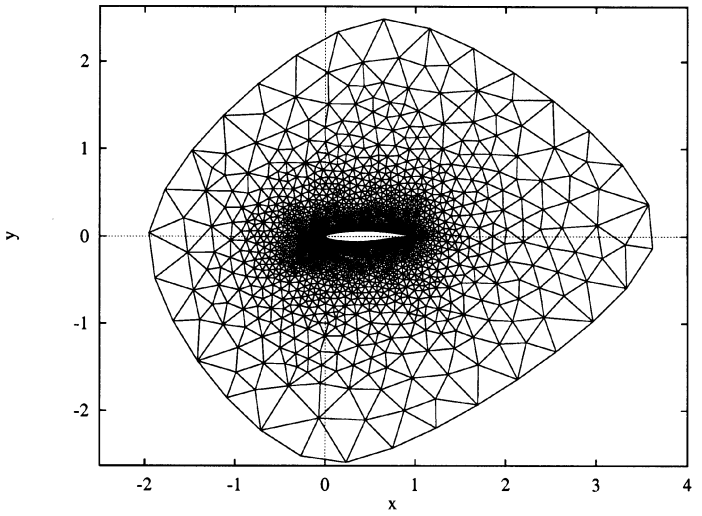


(a) Euler

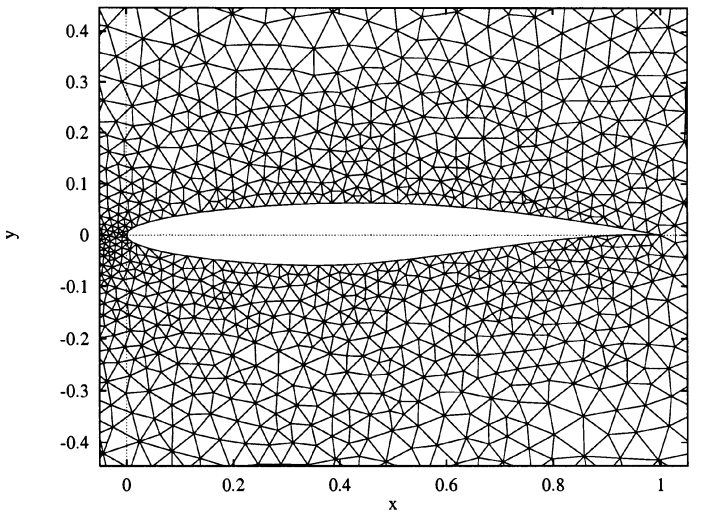


(b) Navier–Stokes

Figure 7.12. Mach 0.85



(a) General view of the computational domain



(b) Zoom

Figure 7.13. The initial computational grid on the RAE profile (2001 vertices, 3886 elements)

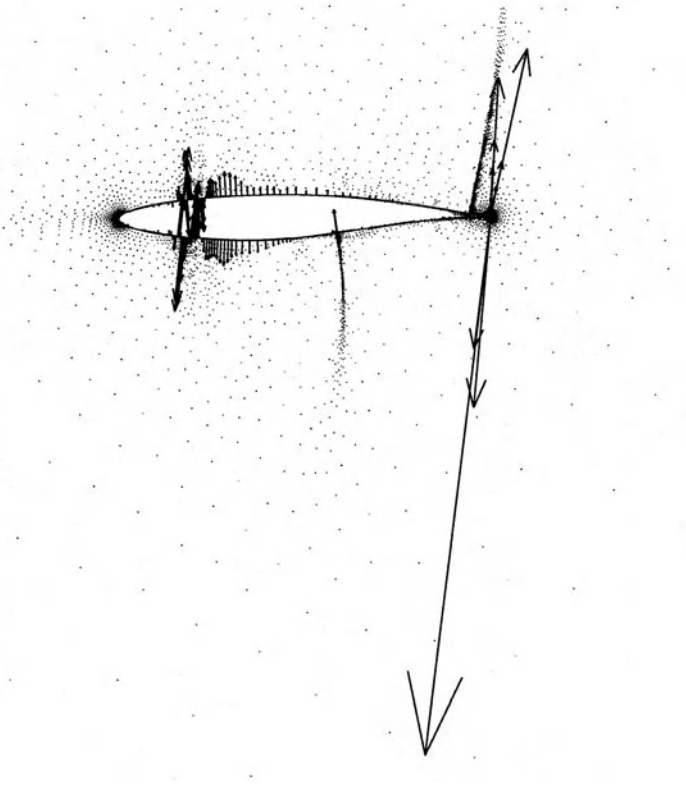


Figure 7.14. A typical gradient field

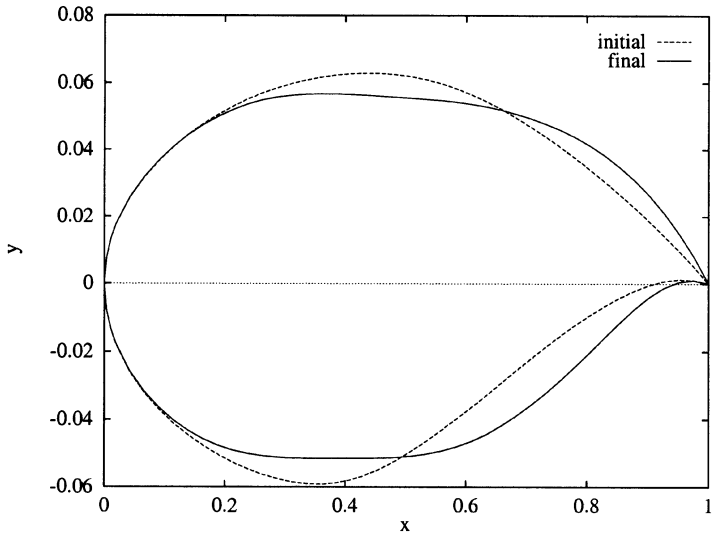


Figure 7.15. Contours before and after optimization (RAE problem, Mach 0.85, Euler)

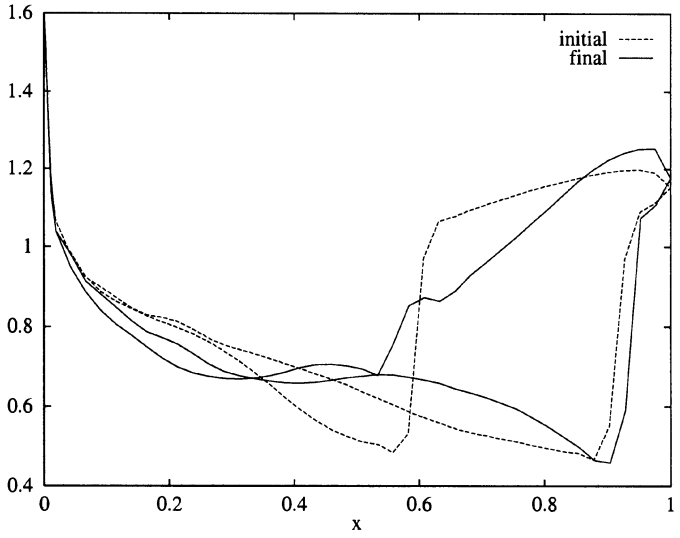


Figure 7.16. Pressure coefficients on the body before and after optimization (RAE problem, Mach 0.85, Euler)

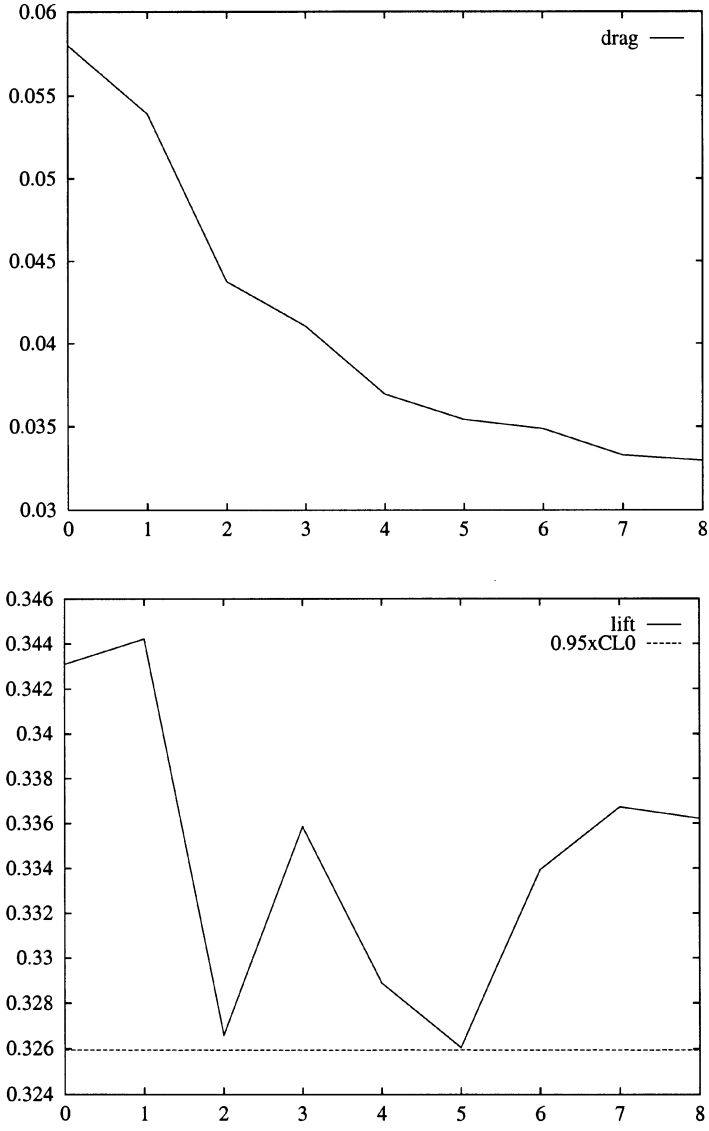


Figure 7.17. Evolution of the drag and of the lift during the iterations of the constrained optimization algorithm. (RAE problem, Mach 0.85, Euler)

7.4 Computing the gradient in aeroelasticity

We now go back to our original aeroelastic model problem, where we wish to optimize the stability of the blade profile by minimizing the energy cost function

$$j(z) = \frac{1}{T_1}\int_0^{T_1}\int_{\partial\Omega_{\text{ext}}} p(W_h(t))\,\underline{n}\cdot\dot{\underline{\xi}}(t)\,da\,dt.$$

Here $T_1 = 2\pi/\omega_1$ and $\dot{\underline{\xi}}$ are respectively the period and the velocity field of the first structural eigenmode of the blade. They are given by solving the eigensystem (1.1)–(1.2). Moreover, p represents the fluid pressure that we obtain by integrating the fluid equations (1.3)–(1.5) over a full period, and $\underline{n}$ denotes the inwards unit normal vector to the blade contour $\gamma = \partial\Omega$.

The structural problem (1.1)–(1.2) and the fluid problem (1.3)–(1.5) are solved on two distinct domains, denoted Ω_{int} and Ω_{ext} respectively (see Figure 7.18). Therefore, the numerical solution of the full problem requires two computational grids $\underline{X}_{\text{int}}$ et $\underline{X}_{\text{ext}}$ associated to the domains Ω_{int} et Ω_{ext}, respectively.

To do this, we just have to construct two initial grids $\underline{X}^o_{\text{int}}$ and $\underline{X}^o_{\text{ext}}$ on the reference internal and external domains, and to define two deformation maps associating to each value z of the control parameters the updated grids

$$\underline{X}_{\text{int}}(z) = \varphi_h(z, \underline{X}^o_{\text{int}}) \quad \text{and} \quad \underline{X}_{\text{ext}}(z) = \varphi_h(z, \underline{X}^o_{\text{ext}}).$$

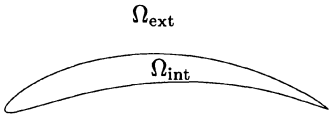


Figure 7.18. The internal and external computational domains Ω_{int} and Ω_{ext}

The construction of the computational grids makes it possible to solve the equations of state. In our aeroelastic problem, we must begin by computing the (finite element) scaled eigenmode $\underline{\xi}(\underline{x}, t) = \underline{\xi}_1(\underline{x})\cos(\omega_1 t) = \sum_i (U_1)_i \underline{\varphi}_i \cos(\omega_1 t)$ of the structure. By construction, the pulsation ω_1 and the vector U_1 containing the nodal values $((U_1)_i)_{i=1,2ns}$ of the corresponding modal deformation are obtained by solving the *implicit* finite element eigenproblem

$$K(\underline{X}_{\text{int}})\cdot U_1 = \omega_1^2 M(\underline{X}_{\text{int}})\cdot U_1, \tag{7.21}$$

$$\langle U_1, M(\underline{X}_{\text{int}})\cdot U_1\rangle = 1. \tag{7.22}$$

This eigenproblem is solved by a rather complex matrix solver whose automatic differentiation is neither needed nor advised.

Knowing the structural mode, one obtains the associated time period

$$T_1 = \frac{2\pi}{\omega_1} \tag{7.23}$$

and the imposed motion of the interface $\partial\Omega_{\text{ext}}$ as a function of time, given by

$$\underline{\xi}(t) = \cos(\omega_1 t) \left(\sum_{i \in \gamma} (U_1)_i \underline{\varphi}_i \right)_{|\gamma} \tag{7.24}$$

and velocity

$$\dot{\underline{\xi}}(t) = \frac{d\underline{\xi}}{dt}(t) = -\omega_1 \sin(\omega_1 t) \left(\sum_{i \in \gamma} (U_1)_i \underline{\varphi}_i \right)_{|\gamma} . \tag{7.25}$$

Observe that here it is much simpler to consider the time period T_1 and the interface motion $(\underline{\xi}, \dot{\underline{\xi}})$ as independent state variables, and to introduce equations (7.23), (7.24) and (7.25) as additional state equations.

Because of the time deformation $\underline{\xi}$ of the blade, the fluid domain $\Omega_{\text{ext}}(t)$ varies in time. Nevertheless, in a simple stability analysis, this vibration is supposed to be of sufficiently small amplitude so that we can assume that, for a given configuration γ, the corresponding change of geometry is negligible. Then, one can compute the flow outside the moving profile by integrating the equations of fluid on the fixed geometry $\Omega_{\text{ext}}(0)$ as indicated in the previous paragraph. This will be usually done by using an explicit solver calculating the vector $W_h(t)$ of conservative variables inside the fluid for a given grid $\underline{X}_{\text{ext}}$, given time period T_1, and given imposed blade velocity $\dot{\underline{\xi}}$ of the interface γ solving the time periodic equations

$$\begin{cases} W_h(0) = W_h(T_1), \\ C_h(\underline{X}_{\text{ext}}) \frac{\partial W_h}{\partial t}(t, \underline{X}_{\text{ext}}) = F_h\left(\underline{X}_{\text{ext}}, \dot{\underline{\xi}}(t), W_h(t)\right) \text{ on }]0, T_1[, \end{cases} \tag{7.26}$$

by a pseudo-integration in time. Above, $C(X_{\text{ext}})$ denotes the diagonal matrix of geometric coefficients $|C_j|$, F_h the vector obtained by adding the different numerical flux functions (see equation (7.18)). Since the pressure $p(W_h(t))$ is also an explicit function of the values of the state vector W_h on the profile, we then obtain explicitly the value of the cost function by computing

$$j(z) = \frac{1}{T_1} \int_0^{T_1} \int_{\partial\Omega_{\text{ext}}} p(W_h(t)) \underline{n} \cdot \dot{\underline{\xi}}(t) \, da \, dt,$$

which can be written equivalently as

$$j(z) = \frac{1}{T_1} \int_0^{T_1} J\left(\underline{X}_{\text{ext}}(z), \dot{\underline{\xi}}(t), W_h(t)\right) dt. \tag{7.27}$$

The next step is to compute the gradient of this cost function with respect to the control (design) parameters. This is done by computing the gradients of

j with respect to the grids $\underline{X}_{\text{int}}$ and $\underline{X}_{\text{ext}}$ as indicated in the previous chapter, by multiplying these gradients respectively by the maps $\frac{\partial \underline{X}_{\text{int}}}{\partial z}(z)$ and $\frac{\partial \underline{X}_{\text{ext}}}{\partial z}(z)$), and by adding the two contributions.

Because of the relative complexity of the problem, the calculation of the gradients of the cost function j with respect to the grids $\underline{X}_{\text{int}}$ and $\underline{X}_{\text{ext}}$ can only be done here by a Lagrangian approach, taking as independent state variables the set $\left(\omega_1, U_1, T_1, \underline{\xi}, \dot{\underline{\xi}}, W_h\right)$ and introducing the Lagrangian

$$
\begin{aligned}
&\mathcal{L}\left(\underline{X}_{\text{int}}, \underline{X}_{\text{ext}}, \left(\omega_1, U_1, T_1, \underline{\xi}, \dot{\underline{\xi}}, W_h\right), \left(\beta, V, S, \underline{\eta}, \dot{\underline{\eta}}, P_h\right)\right) \\
&\quad = \frac{\beta}{2}\left[1 - \left\langle M\left(\underline{X}_{\text{int}}\right) \cdot U_1, U_1\right\rangle\right] + V \cdot \left[-\omega_1^2 M\left(\underline{X}_{\text{int}}\right) + K\left(\underline{X}_{\text{int}}\right)\right] \cdot U_1 \\
&\qquad + S\left(\omega_1 T_1 - 2\pi\right) + \frac{1}{T_1}\int_0^{T_1} \sum_{i \in \gamma} \eta_i(t)\left(\underline{\xi}_i(t) - \cos\left(\omega_1 t\right)\left(U_1\right)_i\right) dt \\
&\qquad\quad + \frac{1}{T_1}\int_0^{T_1} \sum_{i \in \gamma} \dot{\eta}_i(t)\left(\dot{\underline{\xi}}_i(t) + \omega_1 \sin\left(\omega_1 t\right)\left(U_1\right)_i\right) dt \\
&\qquad + \frac{1}{T_1}\int_0^{T_1} P_h(t) \cdot \left(-C_h\left(\underline{X}_{\text{ext}}\right) \cdot \frac{\partial W_h}{\partial t} + F_h\left(\underline{X}_{\text{ext}}, \underline{\xi}(t), \dot{\underline{\xi}}(t), W_h(t)\right)\right) dt \\
&\qquad\quad + \frac{1}{T_1} P_h^{ref} \cdot C_h\left(\underline{X}_{\text{ext}}\right) \cdot \left(W_h\left(T_1\right) - W_h(0)\right) \\
&\qquad\quad + \frac{1}{T_1}\int_0^{T_1} J\left(\underline{X}_{\text{ext}}, \underline{\xi}(t), \dot{\underline{\xi}}(t), W_h(t)\right) dt.
\end{aligned}
$$

To be general, the above Lagrangian takes into account a possible dependence of the state equation and of the cost function on the deformed geometry $\underline{\xi}$ of the blade, which would exist if the change of geometry in time of the fluid domain is no longer neglected. The notation η_i, $\underline{\xi}_i$, $\dot{\eta}_i$ and $\dot{\underline{\xi}}_i$ represents the value of the vector fields $\underline{\eta}$, $\underline{\xi}$, $\dot{\underline{\eta}}$ and $\dot{\underline{\xi}}$ on the i^{th} degree of freedom of the interface γ.

The state equations (7.21)–(7.26) already used for computing the state variable

$$\left(\omega_1, U_1, T_1, \underline{\xi}, \dot{\underline{\xi}}, W_h\right)$$

can be recovered by simple differentiation of this Lagrangian with respect to the adjoint variables $\left(\beta, V, S, \underline{\eta}, \dot{\underline{\eta}}, P_h\right)$. The adjoint state equations are obtained by differentiation with respect to the state variables, inverting the order used for the integration of the state equations.

We therefore begin with the adjoint state equation for the fluid. We get the equation satisfied by the adjoint fluid variable P_h by integrating the Lagrangian by parts in time:

$$\begin{aligned}\mathcal{L}\left(\underline{X}_{\text{int}}, \underline{X}_{\text{ext}}, \left(\omega_1, U_1, T_1, \underline{\xi}, \dot{\underline{\xi}}, W_h\right), (\beta, V, S, \underline{\eta}, \dot{\underline{\eta}}, P_h)\right) = \cdots \\ + \frac{1}{T_1}\int_0^{T_1}\left(\frac{\partial P_h}{\partial t}\cdot C_h\left(\underline{X}_{\text{ext}}\right).W_h(t) + P_h \cdot F_h\left(\underline{X}_{\text{ext}}, \underline{\xi}(t), \dot{\underline{\xi}}(t), W_h(t)\right)\right) dt \\ + \frac{1}{T_1}\left(P_h^{ref} - P_h(T_1)\right)\cdot C_h\left(\underline{X}_{\text{ext}}\right).W_h(T_1) \\ - \frac{1}{T_1}\left(P_h^{ref} - P_h(0)\right)\cdot C_h\left(\underline{X}_{\text{ext}}\right).W_h(0) + \cdots\end{aligned}$$

and by differentiating all terms related to the external flow W_h with respect to the fluid variable W_h. The resulting equation is then the time periodic linear problem with unknown $P_h(t)$ (to be integrated with respect to the auxiliary variable $\tau = -t$ for stability reasons)

$$\begin{cases} P_h(0) = P_h(T_1) = P_h^{\text{ref}}, \\ -C_h\left(\underline{X}_{\text{ext}}\right)\dfrac{\partial P_h}{\partial t}(t) = \left(\dfrac{\partial F_h}{\partial W_h}\left(\underline{X}_{\text{ext}}, \underline{\xi}(t), \dot{\underline{\xi}}(t), W_h\left(t, \underline{X}_{\text{ext}}\right)\right)\right)^t \cdot P_h(t) \\ \qquad\qquad + \dfrac{\partial J}{\partial W_h}\left(\underline{X}_{\text{ext}}, \underline{\xi}(t), \dot{\underline{\xi}}(t), W_h(t)\right) \qquad \text{on }]0, T_1[\,. \end{cases} \tag{7.28}$$

The interface's motion adjoint variables are then obtained by differentiating the Lagrangian with respect to the motion $\underline{\xi}$ and $\dot{\underline{\xi}}$, yielding the explicit formula

$$\underline{\eta}_i(t) + \left(\frac{\partial F_h}{\partial \underline{\xi}_i}\right)^t \cdot P_h(t) + \frac{\partial J}{\partial \underline{\xi}_i} = 0 \quad \text{and} \quad \dot{\underline{\eta}}_i(t) + \left(\frac{\partial F_h}{\partial \dot{\underline{\xi}}_i}\right)^t \cdot P_h(t) + \frac{\partial J}{\partial \dot{\underline{\xi}}_i} = 0, \tag{7.29}$$

that is

$$\underline{\eta}_i(t) = -\left(\frac{\partial F_h}{\partial \underline{\xi}_i}\right)^t \cdot P_h(t) - \frac{\partial J}{\partial \underline{\xi}_i} \quad \text{and} \quad \dot{\underline{\eta}}_i(t) = -\left(\frac{\partial F_h}{\partial \dot{\underline{\xi}}_i}\right)^t \cdot P_h(t) - \frac{\partial J}{\partial \dot{\underline{\xi}}_i}.$$

The adjoint variable S associated to the time period T_1 is obtained by differentiating the Lagrangian with respect to T_1. From the nullity of the term in front of $\frac{1}{T_1}$ and the identity $\omega_1 T_1 = 2\pi$, we get simply

$$\begin{aligned} S = -\frac{1}{2\pi}P_h^{\text{ref}}\cdot C_h\left(\underline{X}_{\text{ext}}\right).\frac{\partial W_h}{\partial t}(T_1) \\ - \frac{1}{2\pi}J\left(\underline{X}_{\text{ext}}, \underline{\xi}(T_1), \dot{\underline{\xi}}(T_1), W_h(T_1)\right) + \frac{1}{2\pi}j\left(\underline{X}_{\text{int}}, \underline{X}_{\text{ext}}\right). \end{aligned} \tag{7.30}$$

The final step is to differentiate the Lagrangian with respect to the eigen-

mode (ω_1, U_1). This yields

$$-2\omega_1 V \cdot M(\underline{X}_{\text{int}}) \cdot U_1 + ST_1 + \frac{1}{T_1}\int_0^{T_1} \sum_{i\in\gamma} \underline{\eta}_i(t) \cdot t\sin(\omega_1 t)\,(U_1)_i\,dt$$
$$+ \frac{1}{T_1}\int_0^{T_1} \sum_{i\in\gamma} \dot{\underline{\eta}}_i(t) \cdot (\sin(\omega_1 t) + \omega_1 \times t\cos(\omega_1 t))\,(U_1)_i\,dt = 0, \quad (7.31)$$

$$-\beta M(\underline{X}_{\text{int}}) \cdot U_1 + \left[-\omega_1^2 M(\underline{X}_{\text{int}}) + K(\underline{X}_{\text{int}})\right] \cdot V$$
$$+ \left(-\frac{1}{T_1}\int_0^{T_1} \underline{\eta}(t)\cos(\omega_1 t)\,dt + \frac{1}{T_1}\int_0^{T_1} \dot{\underline{\eta}}(t)\,\omega_1 \sin(\omega_1 t)\,dt\right)\mathbf{1}_\gamma = 0, \quad (7.32)$$

where $\mathbf{1}_\gamma$ denotes the nodal vector whose i component is equal to 1 (respectively 0) if the degree of freedom i belongs (respectively does not belong) to the interface γ. This can be rewritten as the well-posed matrix system

$$\begin{bmatrix} 0 & -U_1^t \cdot M(\underline{X}_{\text{int}}) \\ -M(\underline{X}_{\text{int}}) \cdot U_1 & -\omega_1^2 M(\underline{X}_{\text{int}}) + K(\underline{X}_{\text{int}}) \end{bmatrix} \begin{pmatrix} \beta \\ V \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \quad (7.33)$$

under the notation

$$A_1 = \frac{1}{2\omega_1}\Bigg\{ -ST_1 - \frac{1}{T_1}\int_0^{T_1} \sum_{i\in\gamma} \underline{\eta}_i(t) \cdot t\sin(\omega_1 t)\,(U_1)_i\,dt$$
$$- \frac{1}{T_1}\int_0^{T_1} \sum_{i\in\gamma} \dot{\underline{\eta}}_i(t) \cdot (\sin(\omega_1 t) + \omega_1 \times t\cos(\omega_1 t))\,(U_1)_i\,dt\Bigg\}, \quad (7.34)$$

$$A_2 = \left(\frac{1}{T_1}\int_0^{T_1} \underline{\eta}(t)\cos(\omega_1 t)\,dt - \frac{1}{T_1}\int_0^{T_1} \dot{\underline{\eta}}(t)\,\omega_1 \sin(\omega_1 t)\,dt\right)\mathbf{1}_\gamma. \quad (7.35)$$

Applying the standard formula (6.7) , we then get

$$\frac{dj}{d(\underline{X}_{\text{int}}, \underline{X}_{\text{ext}})} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{int}}} \\ \frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{ext}}} \end{pmatrix}$$

where the partial derivatives $\mathcal{L}$ with respect to the grids $\underline{X}$ are simply given by

$$\frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{int}}} = -\frac{\beta}{2}\left\langle \frac{dM}{d\underline{X}_{\text{int}}}(\underline{X}_{\text{int}})\,U_1, U_1 \right\rangle$$
$$+ V \cdot \left[-\omega_1^2 \frac{dM}{d\underline{X}_{\text{int}}}(\underline{X}_{\text{int}}) + \frac{dK}{d\underline{X}_{\text{int}}}(\underline{X}_{\text{int}})\right] \cdot U_1, \quad (7.36)$$

and

$$\frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{ext}}} = \frac{1}{T_1}\int_0^{T_1} \frac{\partial J}{\partial \underline{X}_{\text{ext}}}(\underline{X}_{\text{ext}}, W_h(t))\,dt + \frac{1}{T_1}\int_0^{T_1} \frac{\partial P_h}{\partial t} \cdot \frac{dC_h}{d\underline{X}_{\text{ext}}}.W_h(t)\,dt$$
$$+ \frac{1}{T_1}\int_0^{T_1} \left[\frac{\partial F_h}{\partial \underline{X}_{\text{ext}}}(t, \underline{X}_{\text{ext}}, W_h(t))\right]^t \cdot P_h(t)\,dt. \quad (7.37)$$

The gradient of the cost function is then obtained by a direct application of the chain rule, yielding

$$\frac{dj}{dz}(z) = \left\langle \frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{int}}}, \frac{\partial \underline{X}_{\text{int}}}{\partial z}(z) \right\rangle + \left\langle \frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{ext}}}, \frac{\partial \underline{X}_{\text{ext}}}{\partial z}(z) \right\rangle$$

with $\frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{int}}}$ and $\frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{ext}}}$ as computed in (7.36) and (7.37).

Altogether, the procedure for computing the cost function and its gradient in aeroelasticity reduces to the following sequence of operations:

1. z given;
2. calculation of the deformed mesh $\underline{X}_{\text{int}}(z)$ and $\underline{X}_{\text{ext}}(z)$ using the proper mesh deformation subroutine;
3. calculation of the structural eigenmode (ω_1^2, U_1) by solving the finite element matrix eigenproblem (7.21)–(7.22);
4. calculation of the time period T_1, and of the interface's imposed motion $\underline{\xi}$ and $\dot{\underline{\xi}}$ (explicit formula (7.23)–(7.25));
5. calculation of the fluid state vector W_h by solving the time periodic problem (7.26);
6. calculation of the cost function $j(z)$ using formula (7.27);
7. calculation of the fluid adjoint state P_h by solving the time periodic problem (7.28);
8. calculation of the motion adjoint variables and of the corresponding integrals S, A_1, A_2 and $\frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{ext}}}$ by the explicit formula (7.30), (7.34), (7.35) and (7.37);
9. solution of the structural matrix adjoint problem (7.33);
10. calculation of the gradients

$$\frac{\partial}{\partial \underline{X}_{\text{int}}} \langle V, [K(\underline{X}_{\text{int}}) - \omega_1^2 M(\underline{X}_{\text{int}})] U_1 \rangle \quad \text{and} \quad \frac{\partial}{\partial \underline{X}_{\text{int}}} \langle U_1, M(\underline{X}_{\text{int}}) U_1 \rangle$$

by automatic differentiation in adjoint mode of the software calculating the products $\langle V, K(\underline{X}_{\text{int}}) \cdot U_1 \rangle$ and $\langle V, M(\underline{X}_{\text{int}}) \cdot U_1 \rangle$;

11. calculation of the grid gradient $\frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{int}}}$ on all nodes of the internal grid $\underline{X}_{\text{int}}$ using formula (7.36);
12. final calculation of the gradient

$$\frac{dj}{dz}(z) = \left\langle \frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{int}}}, \frac{\partial \underline{X}_{\text{int}}}{\partial z}(z) \right\rangle + \left\langle \frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{ext}}}, \frac{\partial \underline{X}_{\text{ext}}}{\partial z}(z) \right\rangle$$

by automatic differentiation in adjoint mode of the mesh deformation software.

We give below a first example of optimization of the above aeroelastic cost function using on the blade a Young modulus of 217 Mpa, a Poisson coefficient of 0.25, and a mass density of 7800 kg/m^3. These data are typical of an IN 100 turbine blade (ONERA data). The corresponding Lamé coefficients are $\lambda_L = \mu_L = 86.8$ Mpa, and correspond to the constitutive law

$$\underline{\underline{\sigma}}(\underline{\xi}) = \mathbf{E}\underline{\underline{\varepsilon}}(\underline{\xi}) = \lambda_L \text{ Tr } (\underline{\underline{\varepsilon}}(\underline{\xi}))\underline{\underline{Id}} + 2\mu_L\underline{\underline{\varepsilon}}(\underline{\xi}).$$

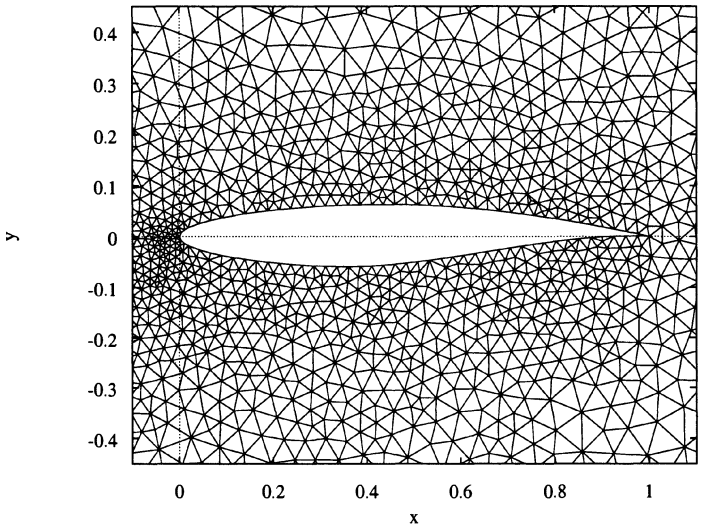
The geometrical design constraints are the same as in the steady aerodynamic case. We also add in one of the two examples presented below a constraint on the mean lift $\frac{1}{T}\int_0^T C_L(\gamma,t)\,dt$, which must remain above 95% of its initial value, where $C_L(\gamma,t)$ is the instantaneous lift of the airfoil γ. Finally, the flow is supposed to be inviscid and the Mach number at inflow is 0.85.

The initial meshes are given by Figure 7.19. Note that the boundary points of the two meshes coincide on the interface γ. During the optimization, all meshes are updated by the explicit sophisticated formula (1.8). The lifting $\mathcal{R}$ used to compute the gradient according to the formula

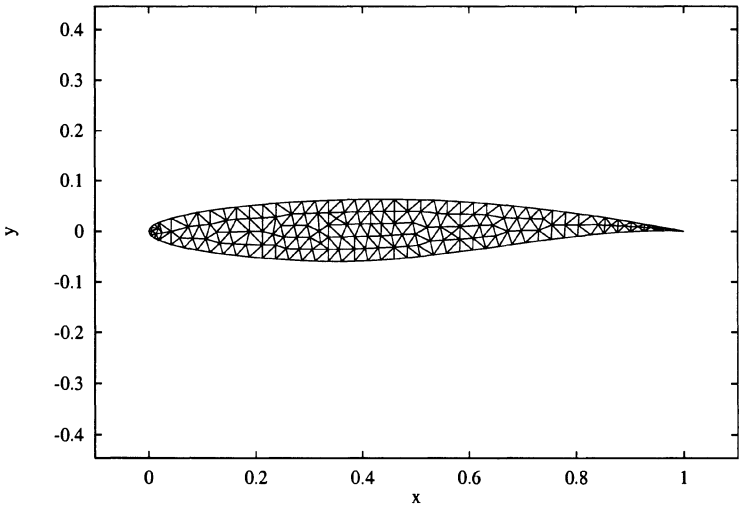
$$\frac{dj}{dz}(z) = \left\langle \frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{int}}}, \frac{\partial \underline{X}_{\text{int}}}{\partial z}(z) \right\rangle + \left\langle \frac{\partial \mathcal{L}}{\partial \underline{X}_{\text{ext}}}, \frac{\partial \underline{X}_{\text{ext}}}{\partial z}(z) \right\rangle$$

replaces the grid derivatives $\frac{\partial \underline{X}}{\partial z}$ at the internal nodes by a simplified expression inspired from the techniques described in Section 1.5.2, the resulting products being obtained by automatic differentiation in adjoint mode of the subroutine iteratively deforming the mesh by this simple averaging technique. The number of shape parameters is $n = 10$. We present the results with only few comments because we did not find any comparison in the literature. Figure 7.20 shows the convergence of the cost for the problem with or without the constraint on the mean lift. We can see the small gain (11% without the constraint and 7% with it). Adding the constraint on the mean lift greatly decreased the speed of convergence of the minimization algorithm. Moreover, Figure 7.21 shows that the result of the optimization without the constraint on the mean lift satisfies nevertheless this constraint. This shows once more the great dependency of the result with respect to the problem and the existence of several local minima.

During the optimization, we see a global decrease of the mean drag. Nevertheless, Figure 7.22 shows that energy and drag are not correlated together. This explains the reason for not reducing the minimization problem to a simple problem on drag. We also see in Figures 7.20, 7.22, 7.24 that there are several main steps in the optimization process, separated by large jumps. We can conjecture that this is due to variations in the pulsation, which would mean that the function which gives the pulsation from the shape has discontinuities, but remains roughly piecewise differentiable. We must also add that both optimizations were made with only 10 shape parameters. A larger number of shape parameters might have given better performances, but on the other hand it might have led to oscillating shapes.



(a) mesh of the flow (2001 nodes)



(b) mesh of the structure (197 nodes)

Figure 7.19. Initial meshes

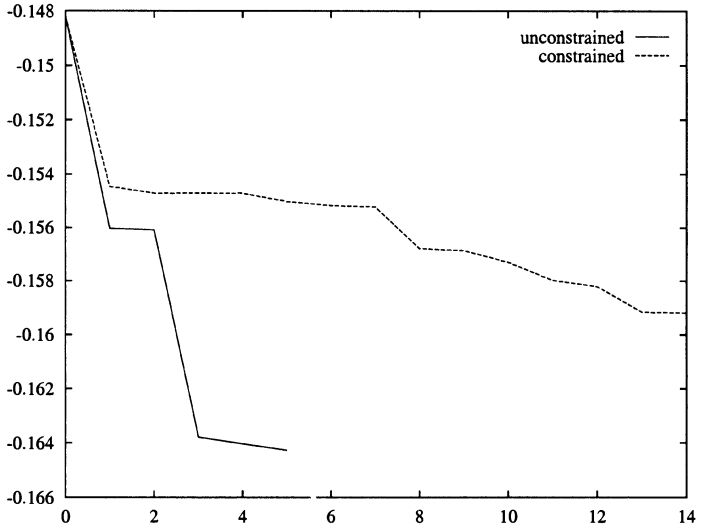


Figure 7.20. Cost convergence for problems with and witout the constraint on the mean lift

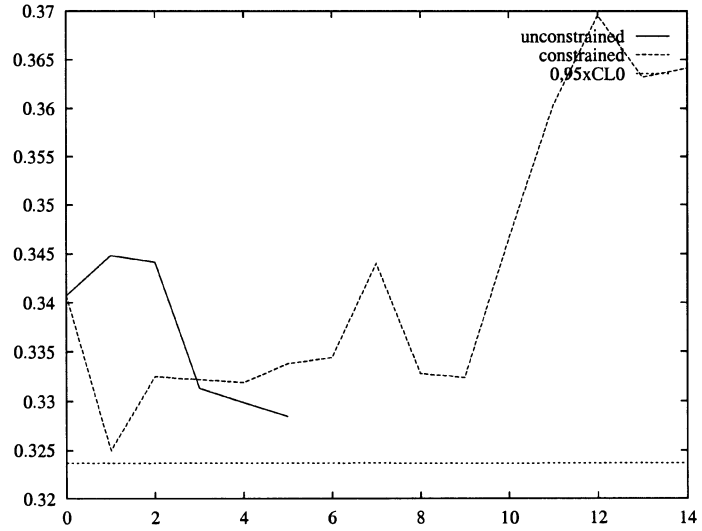


Figure 7.21. Mean lift during optimizations

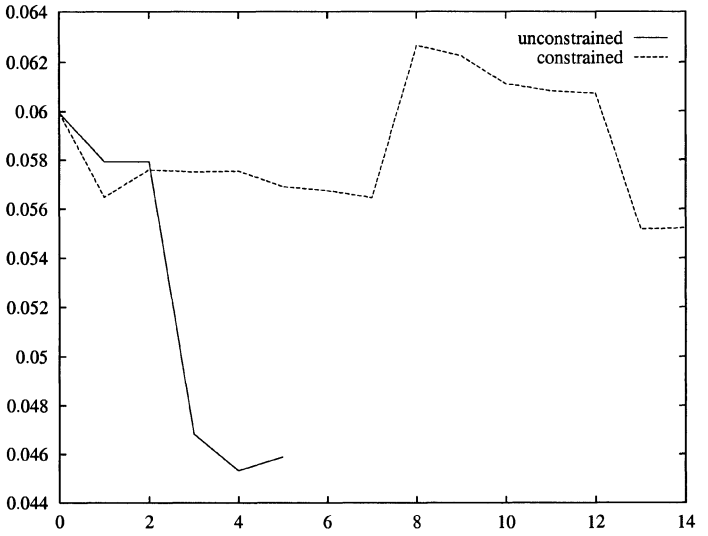


Figure 7.22. Mean drag during optimizations

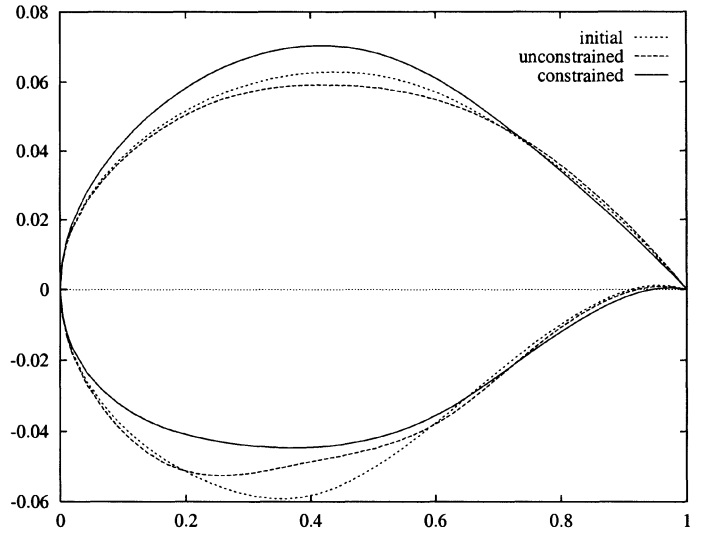
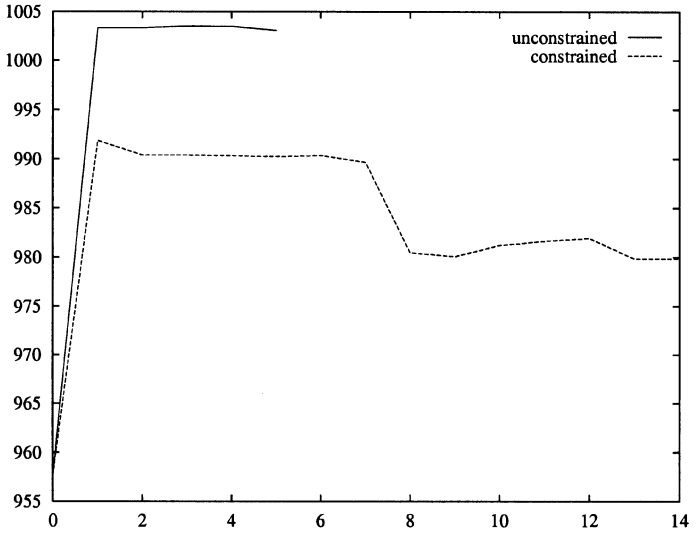
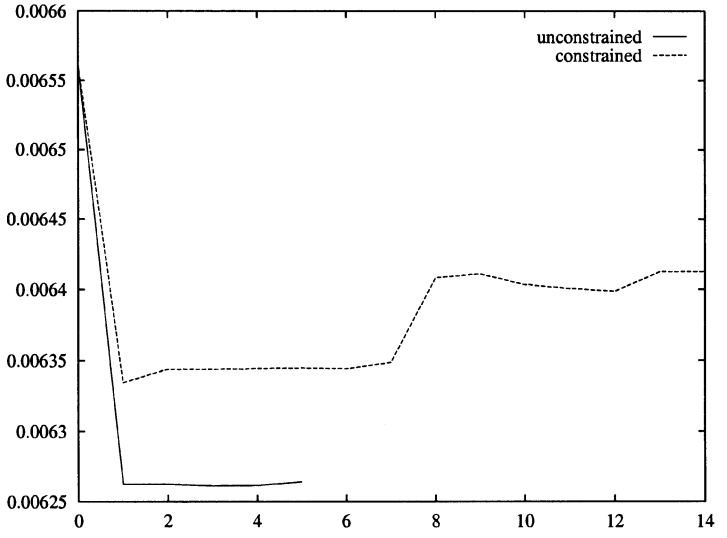


Figure 7.23. Initial and final airfoils for the two problems



(a) pulsations



(b) periods

Figure 7.24. Pulsations and periods during optimizations (physical values)

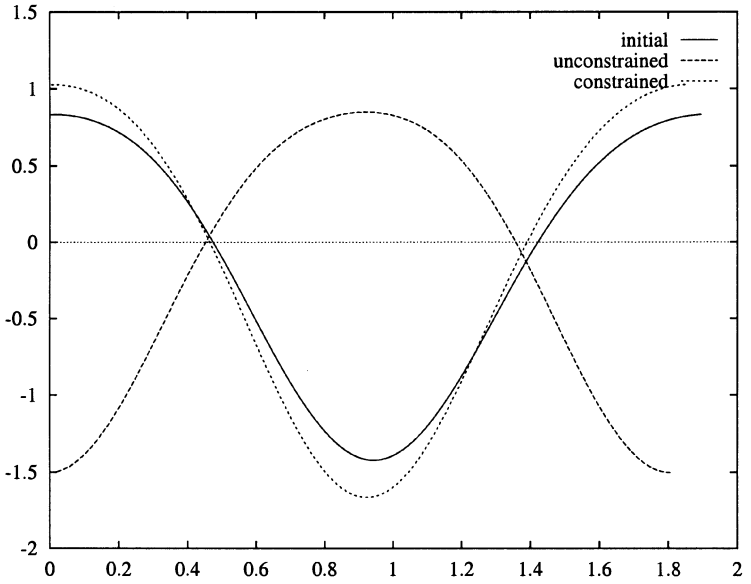


Figure 7.25. Instantaneous energy for initial and final airfoils (with respect to the non-dimensional (reduced) time $\bar{t} = t\sqrt{\frac{E}{\rho}}$)

Chapter 8

One Shot Methods

8.1 Introduction

Many industrial problems of structural design involve a very small number n of design parameters compared with the total number of degrees of freedom needed to correctly handle the state equation. This is why it is interesting to proceed as we have done up to now, that is to eliminate the state variables from the problem and to reduce the optimal shape problem to a constrained cost minimization problem set over the small set $A \subset \mathbb{R}^n$ of admissible designs

$$\min_{\substack{z\in\mathbb{R}^n \\ g_l(z)\leq 0}} J(\underline{X}(z), W_h(z)), \tag{8.1}$$

the computational grid $\underline{X}(z)$ and the state $W_h(z)$ being obtained by solving the state equation

$$(E_h) \begin{cases} f_h(\underline{X}, W_h) = 0 \quad \text{in} \quad \mathbb{R}^p, \\ \underline{X} = \underline{X}(z). \end{cases}$$

Using as before either a Lagrangian approach or applying the implicit function theorem leads then to a rather accessible expression for the gradient

$$\frac{dj}{dz}(z) = \left(\frac{\partial \underline{X}}{\partial z}\right)^t \cdot \left(\frac{\partial J(\underline{X}, W_h(z))}{\partial \underline{X}} + \left(\frac{\partial f(\underline{X}, W_h(z))}{\partial \underline{X}}\right)^t \cdot P(z)\right),$$

with $P(z)$ the adjoint state vector solution of

$$\left(\frac{\partial f(\underline{X}, W_h)}{\partial W}\right)^t|_{(\underline{X},W)} \cdot P = -\frac{\partial J(\underline{X}, W_h)}{\partial W}|_{(\underline{X},W)}.$$

We can also get the gradient $\frac{dg_l}{dz}$ by a similar procedure. These values of j and of its gradient can then be used in a direct sensitivity analysis or in a standard interior point algorithm as described in Chapter 4.

E. Laporte et al., *Numerical Methods in Sensitivity Analysis and Shape Optimization*

The problem in such a procedure is that each evaluation of the cost function or of its gradient requires knowledge of the state vector $W_h \in \mathbb{R}^p$, that is the solution of one occurence of the state equation in $\mathbb{R}^p$. Since p may be very large, and $f_h(\underline{X}, W_h)$ may be very nonlinear, solving all these occurences of the state equation may become prohibitively expensive. To overcome this difficulty, one can think of relaxing the equality constraint $f_h(\underline{X}, W_h) = 0$ and adding it to the optimization programme as an external equality constraint, leading to the new problem

$$\min_{\substack{(z,W_h)\in\mathbb{R}^n\times\mathbb{R}^p \\ g_l(z)\leq 0 \\ f_h(\underline{X}(z),W_h)=0}} J(\underline{X}(z), W_h). \tag{8.2}$$

After introduction of the Lagrange multipliers $\lambda \in \mathbb{R}^m$ and $\mu \in \mathbb{R}^p$ of the inequality constraints $g_l(z) \leq 0$ and of the state equality constraint

$$f_h(\underline{X}(z), W_h) = 0,$$

any local solution (z^*, W_h^*) of this constrained minimization problem satisfies the following system of optimality (Kuhn–Tucker) in $\mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^m$:

$$\frac{\partial J}{\partial z}(\underline{X}(z^*), W_h^*) + \sum_{i=1}^{p} \mu_i^* \frac{\partial f_{hi}}{\partial z}(\underline{X}(z^*), W_h^*) + \sum_{l=1}^{m} \lambda_l^* \frac{\partial g_l}{\partial z}(z^*, W_h^*) = 0, \tag{8.3}$$

$$\frac{\partial J}{\partial W}(\underline{X}(z^*), W_h^*) + \sum_{i=1}^{p} \mu_i^* \frac{\partial f_{hi}}{\partial W}(\underline{X}(z^*), W_h^*) + \sum_{l=1}^{m} \lambda_l^* \frac{\partial g_l}{\partial W}(z^*, W_h^*) = 0, \tag{8.4}$$

$$f_h(\underline{X}(z^*), W_h^*) = 0, \tag{8.5}$$

$$\lambda_l g_l(z^*, W_h^*) = 0, \forall l = 1, m, \tag{8.6}$$

$$\lambda_l \geq 0,\ g_l(z^*, W_h^*) \leq 0, \forall l = 1, m. \tag{8.7}$$

Before describing different algorithms that solve this system, let us recall that the dimension p of the space of state variables is usually much larger than the number n of design parameters or the number m of design constraints. Typical values could be $p = 10000, n = 20, m = 40$. Observe also that the different equations in the above optimality system play different roles:

- the state equation (8.5) characterizes the state variable $W_h(z)$ as an implicit function of the design z;
- the dual optimality equation (8.4) characterizes the multipliers (adjoint variables) μ as an implicit function of the design z, the state variable W_h and the inequality multipliers λ;
- the optimality equations (8.3), (8.6) and (8.7) characterize the optimal design z^* on the boundary of the set C of active constraints as the point where the gradient of the objective function $j(z)$ is orthogonal to this set of active constraints.

One shot methods are then general algorithms designed to solve all the above equations (8.3)–(8.7) (including state equation and conditions of optimality) in one single, but complex, sweep.

8.2 Global algorithm

In fluid mechanics, the above full optimality system is often solved by time marching techniques, or by similar descent methods ([46]). Interior point methods turn out to be also a good method to solve this large dimensional constrained minimization problem (8.2) or its equivalent optimality conditions (8.3)–(8.7) in one single shot. These techniques apply a modified Newton method to construct sequences of variables (z^k, W^k) that satisfy the state equation not at each step, but only at convergence. In other words, the equation of state and the optimality condition are solved simultaneously within the Newton loop. This differs of course from our previous algorithms where one would first solve completely the equation of state and then update z by computing the cost function $j(z)$ and its gradient. The global strategy looks simpler, but this is at the price of solving the complex large nonlinear system (8.3)–(8.7) in one shot.

In order to adapt interior point algorithms to this global problem, we shall employ the variable x to denote the pair (z = design, W_h = state), and introduce the constraint vector $g = (g_l)_{l=1,m} \in \mathbb{R}^m$ and the diagonal matrix $\mathcal{G}$ of order m with diagonal terms g_l. We can now rewrite our problem as

$$\min_{\substack{x \in \mathbb{R}^n \times \mathbb{R}^p \\ f_h(x)=0 \\ g(x) \geq 0}} J(x).$$

Equivalently, we want to find $(x^*, \mu^*, \lambda^*) \in (\mathbb{R}^n \times \mathbb{R}^p) \times \mathbb{R}^p \times \mathbb{R}^m$, a solution of the optimality conditions

$$\frac{\partial \mathcal{L}}{\partial x}(x, \lambda, \mu) := \frac{dJ}{dx}(x) + + \sum_{i=1}^{p} \mu_i \frac{d(f_h)_i}{dx} + \sum_{l=1}^{m} \lambda_l \frac{dg_l}{dx} = 0, \tag{8.8}$$

$$f_h(x) = 0, \tag{8.9}$$

$$\mathcal{G}(x)\lambda = 0, \tag{8.10}$$

$$g(x) \leq 0, \lambda \geq 0, \tag{8.11}$$

associated to the Lagrangian

$$\mathcal{L}(x, \lambda, \mu) = J(x) + \sum_{i=1}^{p} \mu_i (f_h)_i(x) + \sum_{l=1}^{m} \lambda_l g_l(x). \tag{8.12}$$

We are now ready to adapt to this framework the different interior point algorithms introduced in the preceding chapters. Each iteration of such an algorithm will propose a convenient approximation $B \in M_{n+p}(\mathbb{R})$ of the Hessian of the Lagrangian $\mathcal{L}$ with respect to x, construct the corresponding gobal matrix

$H \in M_{n+2p+m}$ of the linearized problem

$$H = \begin{pmatrix} B & \frac{df_h}{dx}^t & \frac{dg}{dx}^t \\ \frac{df_h}{dx} & 0 & 0 \\ \frac{dg}{dx}\lambda & 0 & \mathcal{G} \end{pmatrix} \tag{8.13}$$

and execute the following steps:

1) Newton's prediction: compute the Newton direction by solving the linearized optimality conditions

$$H \cdot \begin{pmatrix} \delta x_o \\ \delta \mu_o \\ \delta \lambda_o \end{pmatrix} = - \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial x}(x, \lambda, \mu) \\ f_h \\ \mathcal{G}\lambda \end{pmatrix}.$$

2) Deflection: to avoid possible unfeasible points as proposed by the Newton direction, introduce a direction of deflection by solving

$$H \begin{pmatrix} \delta x_1 \\ \delta \mu_1 \\ \delta \lambda_1 \end{pmatrix} = - \begin{pmatrix} 0 \\ 0 \\ (\lambda_l \omega_l)_l \end{pmatrix}$$

where ω_l is some given positive number. Once δx_1 is computed, calculate a maximum deflection step ρ such that $\delta x' = \delta x_o + \rho \delta x_1$ is a good direction of descent relatively to δx_o, in the sense

$$\nabla J \cdot \delta x' \leq \frac{1}{2} \nabla J \cdot \delta x_o.$$

3) Line search: compute finally the next value of x (and hence of λ and μ) by minimizing J over the trajectory

$$\{x(t); x(t) = x + t\delta x', t > 0\}.$$

8.3 Reduction of the linear systems

The expensive task in the above algorithm is to solve the global very large linearized systems

$$H \begin{pmatrix} dx \\ d\mu \\ d\lambda \end{pmatrix} = r$$

of order $n + m + 2p$. If we develop the expression of H with respect to the four independent variables z, W, μ and λ, and if we assume for simplicity that the constraints g do not depend explicitly on the values of the state variables W,

each of these global linear systems is of the form

$$\begin{pmatrix} B_{zz} & B_{zW} & \frac{\partial f_h}{\partial z}^t & \frac{dg}{dz}^t \\ B_{Wz} & B_{WW} & \frac{\partial f_h}{\partial W}^t & 0 \\ \frac{\partial f_h}{\partial z} & \frac{\partial f_h}{\partial W} & 0 & 0 \\ \frac{d\mathcal{G}}{dz}\lambda & 0 & 0 & \mathcal{G} \end{pmatrix} \begin{pmatrix} dz \\ dW \\ d\mu \\ d\lambda \end{pmatrix} = \begin{pmatrix} r_z \\ r_W \\ r_\mu \\ r_\lambda \end{pmatrix}. \tag{8.14}$$

When the number n of control parameters is small, and if efficient solvers exist for the linearized state equation

$$\frac{\partial f_h}{\partial W} \cdot dW = r_W,$$

the above global linear system (8.14) can be easily solved by simple elimination of the linear increments dW and $d\mu$ of the state and adjoint state variables W and μ. For this purpose, we use the $p \times n$ matrix $U = (U_1, U_2, \ldots, U_n)$ obtained by solving the n linear systems of matrix $\dfrac{\partial f_h}{\partial W}(z, W)$ defined by

$$\frac{\partial f_h}{\partial W} \cdot U_i = -\frac{\partial f_h}{\partial z_i}, i = 1, n.$$

Introducing this matrix in the third line of the global system (8.14), it is then easy to observe that the solution dW is given as a function of dz by

$$dW = U \cdot dz + \left(\frac{\partial f_h}{\partial W}\right)^{-1} \cdot r_W.$$

If we multiply the second line of the system (8.14) by

$$U^t = -\left(\frac{\partial f_h}{\partial z}\right)^t \cdot \left(\frac{\partial f_h}{\partial W}\right)^{-t},$$

we also get

$$U^t B_{Wz} dz + U^t B_{WW} dW - \left(\frac{\partial f_h}{\partial z}\right)^t \cdot d\mu = U^t r_\mu$$

which yields

$$\frac{\partial f_h}{\partial z}^t \cdot d\mu = (U^t B_{Wz} + U^t B_{WW} U) \cdot dz - U^t r_\mu + U^t B_{WW} \left(\frac{\partial f_h}{\partial W}\right)^{-1} \cdot r_W.$$

We can now eliminate dW and $d\mu$ from the first line of (8.14), and this system finally reduces to the simple linear system given by

$$\begin{pmatrix} B_{zz} + B_{zW}U + U^t B_{Wz} + U^t B_{WW} U & \frac{dg}{dz}^t \\ \frac{d\mathcal{G}}{dz}\lambda & \mathcal{G} \end{pmatrix} \begin{pmatrix} dz \\ d\lambda \end{pmatrix}$$

$$= \begin{pmatrix} r_z + U^t r_\mu - (B_{zW} + U^t B_{WW}) \left(\frac{\partial f_h}{\partial W}\right)^{-1} \cdot r_W \\ r_\lambda \end{pmatrix}. \tag{8.15}$$

This linear system is of very small dimension (it is of order $n + m$) which makes it easy to solve by any type of Krylov residual based method such as GMRES. Such techniques only require us to compute matrix vector products of the form

$$H_z dz_i := \Big(B_{zz} + B_{zW}U + U^t B_{Wz} + U^t B_{WW} U \Big) dz_i, \forall i = 1, \ldots, n,$$

which is easily computed by automatic differentiation, even if one chooses for matrix B the exact Hessian of the problem's Lagrangian.

Remark 8.1 In the absence of design constraints, a variant of the above algorithm consists in applying a standard Newton algorithm to the system

$$\begin{aligned} &\frac{\partial J}{\partial z}(z, W_h) + \sum_{i=1}^{p} \mu_i(z, W_h) \frac{\partial (f_h)_i}{\partial z}(z, W_h) = 0, \\ &f_h(z, W_h) = 0, \end{aligned}$$

where the Lagrange multiplier

$$\mu(z, W_h) = - \left(\frac{\partial f_h}{\partial W}(z, W_h) \right)^{-t} \cdot \frac{\partial J}{\partial W_h}(z, W_h)$$

is treated as an explicit function of z and W_h. This variant has been proposed and studied in Hoyer [26].

8.4 Line search

To be robust, the line search strategy must be restricted to a neighborhood of the solution curve $f_h(\underline{X}(z), W_h) = 0$. For this purpose, the basic interior point method must be adapted. A first important enhancement consists in adding a restoration step before the global Newton step in order to generate sequences of state variables W_h leading to small residuals in the equation of state $f_h(\underline{X}(z), W_h) = 0$. The second enhancement is to add a parabolic correction in the line search procedure in order to take into account the curvature of the solution curve $f_h = 0$. The full enhanced line search procedure proposed by Halard [20] can then be organized as follows:

- compute first a linear update δW_0 of the state variable by solving the linearized state equation with frozen design z:

$$\left(\frac{\partial f_h}{\partial W} \right) \delta W_0 = r_W = -f_h(z, W_h);$$

- compute the update direction δz as predicted by the interior point algorithm when applied to the full optimality system, and compute the associated matrix of influence $U = - \left(\frac{\partial f_h}{\partial W} \right)^{-1} \cdot \frac{\partial f_h}{\partial z}$;

- construct a second order parabolic curve

$$(z(t), W(t)) = (z + t\delta z, W_h + \delta W_0 + tU \cdot \delta z + t^2 \delta W_2)$$

going through the updated point $(z, W_h + \delta W_0)$ and satisfying at best the approximate state equation

$$f_h(z + t\delta z, W_h + \delta W_0 + tU \cdot \delta z + t^2 \delta W_2) = f_h(z, W_h + \delta W_0).$$

By construction of the $p \times n$ matrix U, the residual in the above equation at $t = 0$ evolves like $t||D^2 f_h||||\delta W_0||||\delta z||$, whatever choice is made for δW_2. A good strategy for keeping the residual uniformly small is to impose that the residual be also of third order, that is of order $||\delta W_0||^2||\delta z||$, when $t = 1$. A straighforward Taylor expansion around the point $(z + \delta z, W_h + \delta W_0 + U \cdot \delta z)$, and the first-order approximation of the Jacobian $\dfrac{\partial f_h}{\partial W}(z+\delta z, W_h+\delta W_0+U\cdot\delta z)$ by the already available matrix $\dfrac{\partial f_h}{\partial W}(z, W_h)$ shows that this can be achieved by choosing the correction vector δW_2 as the solution of the linear system

$$\left(\frac{\partial f_h}{\partial W}(z, W_h)\right) \cdot \delta W_2 = -\left(f_h(z+\delta z, W_h+\delta W_0+U\cdot\delta z) - f_h(z, W_h+\delta W_0)\right).$$

By construction of δz, the right-hand side in the above equation and hence the corrector δW_2 is of order $||\delta z||^2 + ||\delta W_0||^2$;

- search for the point $(z(\bar{t}), W(\bar{t})) = (z + \bar{t}\delta z, W_h + \delta W_0 + \bar{t}U \cdot \delta z + \bar{t}^2 \delta W_2)$ on the above parabolic curve with minimal cost function

$$\varphi(z, W) = J(z, W) + \sum |\mu_i||f_{hi}(z, W)|.$$

We use here the usual Han penalty function already introduced in Chapter 3. A bound on the correction δz may be imposed during this line search procedure;

- take as the new solution the vector

$$W = W(\bar{t}) = W_h + \delta W_o + \bar{t}U \cdot \delta z + \bar{t}^2 \delta W_2,$$
$$z = z + \bar{t}\delta z.$$

Remark 8.2 The above line search strategy corresponds to the choice of parameters $ICURV = 2$ and $ICOST = 2$ in the numerical results to come. It will be compared to simpler strategies, characterized by different choices of penalty functions or of line search curves. More precisely,

- $ICURV = 0$ will correspond to a standard linear search along the straight line $(z + t\delta z, W_h + t(\delta W_0 + U \cdot \delta z))$ originating from the current estimate (z, W_h) following the direction of descent $(\delta z, \delta W = \delta W_0 + U\delta z)$ proposed by the interior point algorithm,

- $ICURV = 1$ will correspond to a line search along the straight line $(z + t\delta z, W_h + \delta W_0 + tU \cdot \delta z)$ originating from the point $(z, W_h + \delta W_0)$ obtained after linear restoration, and satisfying the approximate state equation $f_h(z + t\delta z, W_h + \delta W_0 + tU \cdot \delta z + t^2 \delta W_2) = f_h(z, W_h + \delta W_0)$ of first-order in t,
- $ICURV = 2$ will correspond to a line search along the second-order parabolic curve $(z + t\delta z, W_h + \delta W_0 + tU \cdot \delta z + t^2 \delta W_2)$ satisfying the approximate state equation $f_h(z + t\delta z, W_h + \delta W_0 + tU \cdot \delta z + t^2 \delta W_2) = f_h(z, W_h + \delta W_0)$ at second-order,
- $ICOST = 0$ will correspond to a basic cost function $\varphi(z, W) = J(z, W)$ ignoring the state equation,
- $ICOST = 1$ or 2 will correspond to the usual penalized cost function given by $\varphi(z, W) = J(z, W) + \sigma \sum_i |f_{hi}(z, W)|$.

8.5 Detailed algorithm

The remaining part of the chapter describes the numerical implementation proposed in Matthieu Halard's doctoral work [20], and the numerical results which he has obtained. We refer to his thesis for additional details. The underlying interior point algorithm has been proposed and studied by Panier, Tits and Herskovits [41]. It is assumed for simplicity that the design constraints g_l do not depend on the state variable W. When applied to one shot optimal design problems, this algorithm takes the following form.

Parameters

- $0 < \alpha < 1/2$ and $0 < \beta < 1$: parameters controlling the Armijo line search strategy;
- $0 < \theta \leq 1$: parameter controlling the deflection step;
- $0 < E_{Max}$: upper bound on the admissible residual in the state equation;
- $0 < \lambda_{Max}$ upper bound on the Lagrange multipliers;
- $0 < \|\lambda_{ref}\|$: parameter controlling the update strategy for the Lagrange multipliers.

Initialization

Choose admissible initial values $\lambda^{(0)}$, $z^{(0)}$ and $W^{(0,0)}$ for the Lagrange multipliers, design parameters and state variables, satisfying in particular

$$\lambda_l^{(0)} > 0, \forall 1 \leq l \leq m, \quad g_l(z^{(0)}) < 0, \forall 1 \leq l \leq m.$$

Equilibrium recovery

Correct the initial value of the state variables by a few Newton steps, $p = 0, \ldots,$ in order to obtain a sufficiently small residual $\|f_h(z^{(0)}, W^{(0,p)})\| < E_{Max}$:

1. compute the residual $f_h(z^{(0)}, W^{(0,p)})$ and the stiffness matrix
$$K = \frac{\partial f_h}{\partial W}(z^{(0)}, W^{(0,p)});$$
2. solve the linear system $K\delta W = -f_h(z^{(0)}, W^{(0,p)})$;
3. set $W^{(0,p+1)} = W^{(0,p)} + \delta W$.

Use then the converged value of the state variable $W^{(0,p+1)}$ as the new initial guess: $W^{(0)} = W^{(0,p+1)}$.

Optimization loop

Within this loop, the current admissible values $\lambda^{(k)}$, $z^{(k)}$ and $W^{(k)}$ of the Lagrange multipliers, design parameters $z^{(k)}$ and state variables are updated together by the following sequences of operations.

Preliminary calculations

1. Compute the present value of the cost function and of its derivatives
$$J^{(k)} = J(z^{(k)}, W^{(k)}),\ J_z^{(k)} = \frac{\partial J}{\partial z}(z^{(k)}, W^{(k)}) \text{ and } J_W^{(k)} = \frac{\partial J}{\partial W}(z^{(k)}, W^{(k)}).$$
2. Compute the present value of the state equation $f_h(z^{(k)}, W^{(k)})$, of its derivatives $f_{h,z}^{(k)} = \frac{\partial f_h}{\partial z}(z^{(k)}, W^{(k)})$ and stiffness matrix
$$K^{(k)} = \frac{\partial f_h}{\partial W}(z^{(k)}, W^{(k)}).$$
3. Compute the present value of the design constraints and of their derivatives
$$g_l^{(k)} = g_l(z^{(k)}) \text{ and } g_{l,z}^{(k)} = \frac{\partial g_l}{\partial z}(z^{(k)}).$$
4. Compute the Newton correction on the state variable by solving the linear system
$$K^{(k)}\delta W_0 = -f_h(z^{(k)}, W^{(k)}).$$
5. *If the residual $f_h(z^{(k)}, W^{(k)})$ or the correction δW_0 are too large, perform a few Newton steps on the state equation in order to decrease the residual to an admissible value smaller than E_{Max}.*

6. Compute the Lagrange multiplier $\mu^{(k)}$ by solving the linear system

$$(K^{(k)})^t \mu^{(k)} = -J_W^{(k)}.$$

7. Compute the matrix of influence $U^{(k)}$ by solving

$$K^{(k)} U^{(k)} = -f_{h,z}^{(k)}.$$

This amounts to solving n linear systems with fixed matrix $K^{(k)}$.

Newton's prediction

1. Compute the reduced right-hand side

$$b_z^{(k)} = \mathcal{L}_z^{(k)} + (\mathcal{L}_{z,W}^{(k)} + U^t \mathcal{L}_{W,W}^{(k)}) \delta W_0$$

in δz either by automatic differentiation in direct mode, or by finite differences.

2. Compute the first Krylov vector $r_0^{(k)} = \dfrac{b_z^{(k)}}{\|b_z^{(k)}\|}$.

3. Compute the reduced Hessian matrix $\mathcal{H}^{(k)}$ in the Krylov basis $(r_i)_{i=1,\ldots}^{(k)}$ by iteratively computing the products $H_z^{(k)} r_i^{(k)}$, that is

   (a) computing the associate update of state variable $\Delta W_i^{(k)} = U^{(k)} r_i^{(k)}$,

   (b) computing the directional derivatives

$$\mathcal{L}_z = \mathcal{L}_{z,z}^{(k)} r_i^{(k)} + \mathcal{L}_{z,W}^{(k)} \Delta W_i^{(k)} + \sum_{l=1}^{m} \frac{\lambda_l^{(k)}}{-g_l^{(k)}} \left(g_{l,z}^{(k)} \cdot r_i^{(k)} \right) g_{l,z}^{(k)},$$

$$\mathcal{L}_W = \mathcal{L}_{W,z}^{(k)} r_i^{(k)} + \mathcal{L}_{W,W}^{(k)} \Delta W_i^{(k)},$$

   (c) computing the product $H_z^{(k)} r_i^{(k)} = \mathcal{L}_z + (U^{(k)})^t \mathcal{L}_W$,

   (d) computing the dot products $\mathcal{H}_{j,i}^{(k)} = \langle H_z^{(k)} r_i^{(k)}, r_j^{(k)} \rangle$, $\forall j = 1, i$,

   (e) computing the new Krylov vector $\tilde{r}_i = H_z^{(k)} r_i^{(k)} - \sum_{j \le i} \mathcal{H}_{j,i}^{(k)} r_j^{(k)}$,

   (f) computing the number $\mathcal{H}_{i+1,i}^{(k)} = \|\tilde{r}_i\|$,

   (g) computint the normalized Krylov vector $r_{i+1}^{(k)} = \tilde{r}_i / \|\tilde{r}_i\|$.

4. Make a QR factorization of the Hessenberg matrix $\mathcal{H}^{(k)}$.

5. Solve the linear system $\mathcal{H}^{(k)} y = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$.

6. Get the solution $\delta z_0^{(k)}$ of the system $H_z^{(k)}\delta z_0^{(k)} = b_z^{(k)}$ as

$$\delta z_0^{(k)} = \sum_i y_i r_i^{(k)}.$$

Deflection

1. Compute the deflection right-hand side $b_1^{(k)} = -\sum_{l=1}^{m} \frac{\lambda_l^{(k)}}{-g_l^{(k)}} g_{l,z}^{(k)}$.

2. Compute the coordinates $x_i = \langle b_1^{(k)}, r_i^{(k)} \rangle$ of the right-hand side $b_1^{(k)}$ in the Krylov basis $(r_i^{(k)})_i$.

3. Solve the linear system $\mathcal{H}^{(k)}\tilde{y} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$.

4. Get the solution $\delta z_1^{(k)}$ of the system $H_z^{(k)}\delta z_1^{(k)} = b_1^{(k)}$ as

$$\delta z_1^{(k)} = \sum_i \tilde{y}_i r_i^{(k)}.$$

5. Compute the maximum coefficient $\rho_1^{(k)}$ in the linear combination of $\delta z_0^{(k)}$ and $\delta z_1^{(k)}$ in order to get the maximum deflection while preserving an efficient decrease of the cost function, that is compute

$$\begin{aligned} \rho_1^{(k)} &= 1 \quad \text{if } \langle b_z^{(k)}, \delta z_1^{(k)} \rangle \leq \theta \langle b_z^{(k)}, \delta z_0^{(k)} \rangle, \\ &= (1-\theta)\frac{\langle b_z^{(k)}, \delta z_0^{(k)} \rangle}{\langle b_z^{(k)}, \delta z_0^{(k)} - \delta z_1^{(k)} \rangle} \quad \text{if not.} \end{aligned}$$

6. Compute the deflected direction of update

$$\delta z^{(k)} = (1-\rho_1^{(k)})\delta z_0^{(k)} + \rho_1^{(k)}\delta z_1^{(k)}.$$

Armijo's Line search

1. Compute the parabolic correction on the state equation by solving the second-order correction equation

$$K^{(k)}\delta W_2^{(k)} = -\Big(f_h(z^{(k)} + \delta z^{(k)}, W^{(k)} + \delta W_0 + U^{(k)} \cdot \delta z^{(k)}) - f_h(z^{(k)}, W^{(k)} + \delta W_0)\Big).$$

2. Defining $z(t)$ by $z(t) = z^{(k)} + t\delta z^{(k)}$, $W(t)$ by

$$W(t) = W^{(k)} + \delta W_0 + t\,U^{(k)}\delta z^{(k)} + t^2\delta W_2^{(k)},$$

the cost function $\phi(t)$ by

$$\phi(t) = J(z(t), W(t)) + \sigma\|f_h(z(t), W(t))\|,$$

and defining an active constraint as a constraint such that

$$g_l(z^{(k)}) \leq \frac{\lambda_l^{(k)}}{-g_l^{(k)}} g_{l,z}^{(k)} \cdot \delta z^{(k)},$$

compute the first element $\bar{t}$ in the sequence $(1, \beta, \beta^2, \ldots)$ such that

$$\begin{aligned}
&\phi(t) < \phi(0) + \alpha t\phi'(t),\\
&g_l(z(t)) < 0, \forall l = 1, m,\\
&g_l(z(t)) < g_l(z(0)) \text{ for inactive constraints .}
\end{aligned}$$

Solution update

$$\begin{aligned}
\text{Set}\quad & z^{(k+1)} = z^{(k)} + \bar{t}\delta z^{(k)},\\
& W^{(k+1)} = W^{(k)} + \delta W_0 + \bar{t}U^{(k)}\delta z^{(k)} + \bar{t}^2\delta W_2^{(k)},\\
& \lambda_l^{(k+1)} = \max\left(\|\lambda_{ref}\|\|\delta z\|, \frac{\lambda_l^{(k)}}{-g_l^{(k)}} g_{l,z}^{(k)} \cdot \delta z^{(k)}\right).
\end{aligned}$$

End of Optimization loop

Remark 8.3 The complexity of the above algorithm is rather easy to estimate when the linear systems involving the stiffness matrix are solved by a direct solver. Let N_E be the number of finite elements used in the discretization and $L \approx N_E^{2/3}$ the bandwidth of the stiffness matrix. Under this notation, and when using finite elements of order 2, a typical iteration in the optimization loop involves

1. $\approx 21000 \times N_E$ floating point operations for computing the functions, stiffness matrix, and first-order derivatives in the preliminary step,

2. $\approx N_E^{7/3}$ floating point operations for factorizing the stiffness matrix $K^{(k)}$ in the preliminary step,

3. $(5+n) \times N_E^{5/3}$ floating point operations for solving the (5+n) linear systems with matrix $K^{(k)}$ needed to compute the first Newton update and influence matrix $U^{(k)}$ in the preliminary step,

4. $\approx (2+n) \times 100N_E + n^3/6$ floating point operations for computing and factoring the Krylov matrix $\mathcal{H}^{(k)}$,

5. $\approx 1000N_E$ floating point operations for the deflection and line search step.

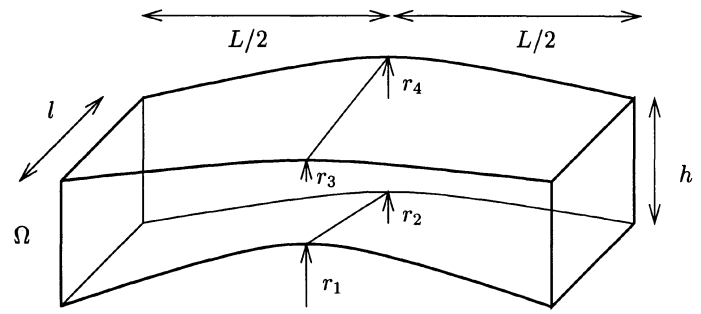
For typical values of $n \approx 10$ and $N_E \in (10^3, 10^4)$, we observe that the main cost in the iteration is the factorization cost. This means that the cost of a full optimization step in our global algorithm is similar to the cost of a single Newton step.

8.6 Numerical examples

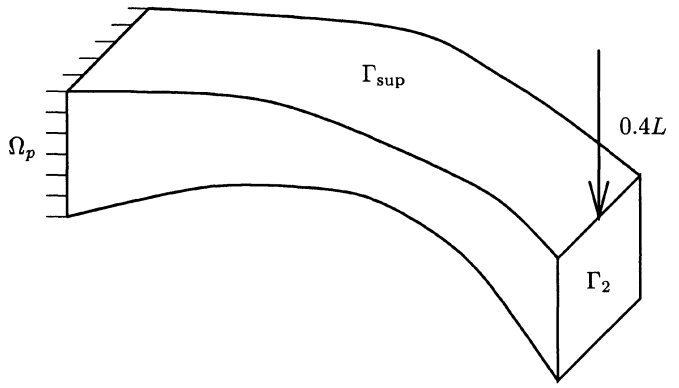
The efficiency of the above one shot method has been validated in [20] on several test cases in which the state equations were the equilibrium equations of a nonlinear structure made of a compressible hyperelastic material in large displacements. The structure under consideration was a thick beam of length $L = 1m$, width $l = 0.3m$ and height $h = 0.20m$, clamped on one end, and subjected at the other end to an imposed displacement. Within the structure, the constitutive law is of Saint-Venant–Kirchhoff type, associated to a free energy density $\psi(E)$ given as a function of the Green–Lagrange strain tensor $\underline{\underline{E}}$ by

$$\psi(\underline{\underline{E}}) = \frac{\lambda_E}{2}(Tr\underline{\underline{E}})^2 + \mu_E Tr(\underline{\underline{E}}^2),$$

with Lamé coefficients $\lambda_E = 0.35\,Gpa$ and $\mu_E = 0.7\,Gpa$. The control parameters are the vertical positions of the four vertices of the mid cross section (see Figure 8.1), measured as differences from the positions of the corresponding vertices at the end sections. In other words, the choice $z = (r_1, r_2, r_3, r_4) = (0, 0, 0, 0)$ corresponds to a straight beam. In this simple case, the coordinates $\underline{X}(z)$ of the finite element nodes are given explicitly in terms of the design parameters z by parabolic interpolation between the end and mid sections. The calculations are performed on regular structured meshes, with a number NE of second-order hexaedral finite elements ranging from $NE = 48$ to $NE = 960$. Correspondingly, the number p of state variables (three times the number of finite element nodes) ranges from $p = 1047$ to $p = 14367$.



Design parameters: vertical displacements of the mid section vertices



Imposed displacement on the end section Γ_2

Figure 8.1. The model mechanical problem: optimal design of a clamped beam in large deformations. Taken from [20]

8.6.1 An example without inequality constraints

In the first case, the right end is subjected to a large vertical displacement $\xi_z = 0.4L$, combined with zero axial displacement and shear forces. For the design

$$z = (r_1, r_2, r_3, r_4) = (0.01m, 0.01m, 0.02m, 0.02m),$$

the equilibrium solution can be computed in 16 Newton iterations starting from the initial guess $W^0 = 0$, and is described in Figure 8.2

Denoting by ξ_z^0 the corresponding vertical displacement, the optimal design problem now consists in approximating at best a given displacement field $\xi_z^0 + 0.05m$ on the top surface $\Gamma_{sup}(z)$ of the beam. For this purpose, we define the

cost function by

$$
\begin{aligned}
&z = (r_1, r_2, r_3, r_4),\\
&W = (\xi_x, \xi_y, \xi_z)(M_i), i = 1, NDOF,\\
&J(z, W) = \int_{\Gamma_{sup}(z)} (\xi_z - \xi_z^0 - 0.05)^2 da.
\end{aligned}
$$

The problem has been solved by the above interior point algorithm with different options. Figure 8.3 presents a typical convergence curve, obtained when starting with $W = 0$, $z = (0.00, 0.00, 0.05, 0.01)$ with the choice $ICOST = 0$ (standard cost function) and $ICURV = 1$ (linear search), and with an initial equilibrium recovery step using three Newton iterations. The optimal solution corresponds to $r_1 = r_2 = -0.25m$ and $r_3 = r_4 = -0.16m$. In this case, the convergence is not very sensitive to the particular line search strategy, but is more sensitive to the tolerance $E_{\max}$ used for the state equation and to a possible upper bound in the update of the control (design) parameters z. Observe that the convergence of the full optimization loop is very similar to the convergence of the Newton method when applied to the solution of the state equation. Full convergence is achieved in less than 20 iterations. By observing in addition that the CPU cost of one iteration of the complete optimization loop is only 10 percent higher than the CPU cost of a single Newton iteration (the main cost is here the factorization of the stiffness matrix), we can infer that in this case, finding the optimal shape is only marginally more expensive than simply computing the equilibrium solution.

8.6.2 Constrained torsion

In this case, the right end is subjected to a given rotation of 15 degrees. The cost function is the distance after deformation of the top face to the horizontal plane. In other words, we have

$$
\begin{aligned}
&z = (r_1, r_2, r_3, r_4),\\
&W = (\xi_x, \xi_y, \xi_z)(M_i), i = 1, NDOF,\\
&J(z, W) = \int_{\Gamma_{sup}(z)} (z + \xi_z)^2 da.
\end{aligned}
$$

In this case, we impose different geometric constraints on the design in order to keep a reasonable shape. The selected constraints are:

1. positive thickness $h + r_3 - r_1 \geq \varepsilon, \quad h + r_4 - r_2 \geq \varepsilon$;
2. shape convexity $r_1 + r_2 \geq 0$;
3. bounded aspect ratios in the mid section. Introducing the angles θ_1 and θ_2 as indicated in Figure 8.4, we bound these angles by the constraints $\theta_{\min} \leq \theta_1 \leq \theta_{\max}$ and $\theta_{\min} \leq \theta_2 \leq \theta_{\max}$, with $\theta_{\min} = 10$ degrees, and $\theta_{\max} = 50$ degrees.

The active constraints at solution are $r_1 + r_2 = 0$, $\theta_1 = \theta_2 = \theta_{\max}$, the solution being given by $z = (-0.032m, 0.032m, 0.066m, 0.033m)$.

The following curves present the convergence histories (cost function, residual in the state equation, residual in the optimality condition) of the different numerical implementations, used with the initial value

$$z = (0.01m, 0.01m, 0.05m, 0.01m).$$

The various implementations are characterized by the line search strategy ($ICOST$, $ICURV$) and by the presence or not of the deflection step ($IFLEX = 0$ if no deflection, $IFLEX = 1$ if deflection). We first can observe on the evolution of the cost function (Figure 8.5) that the standard line search $ICURV = 0$ is rather inefficient. We can also observe that the deflection step brings no spectacular improvement in this particular case. This is probably due to the presence at convergence of conflicting active constraints.

8.7 Conclusion

Altogether, three levels of algorithm are involved in the one shot methodology presented above. The higher level is the optimization strategy of the full problem in (z, W_h), for which we have applied an adapted interior point algorithm. At each iteration of this algorithm, several linear systems are solved, all involving the same Hessian matrix on different right-hand sides. The intermediate level is therefore an algebraic solver that computes and solves efficiently such Hessian systems (8.14) for optimal design problems. At last, to fully describe our method for optimum design as applied to nonlinear structures, gradient computations constitute the third and lowest level and they are performed as indicated in the preceeding chapters.

These one shot methods are also of quite general use, they can be very efficient when properly used, but they are often less robust than the algorithms which operate directly in the design space $\mathbb{R}^n$.

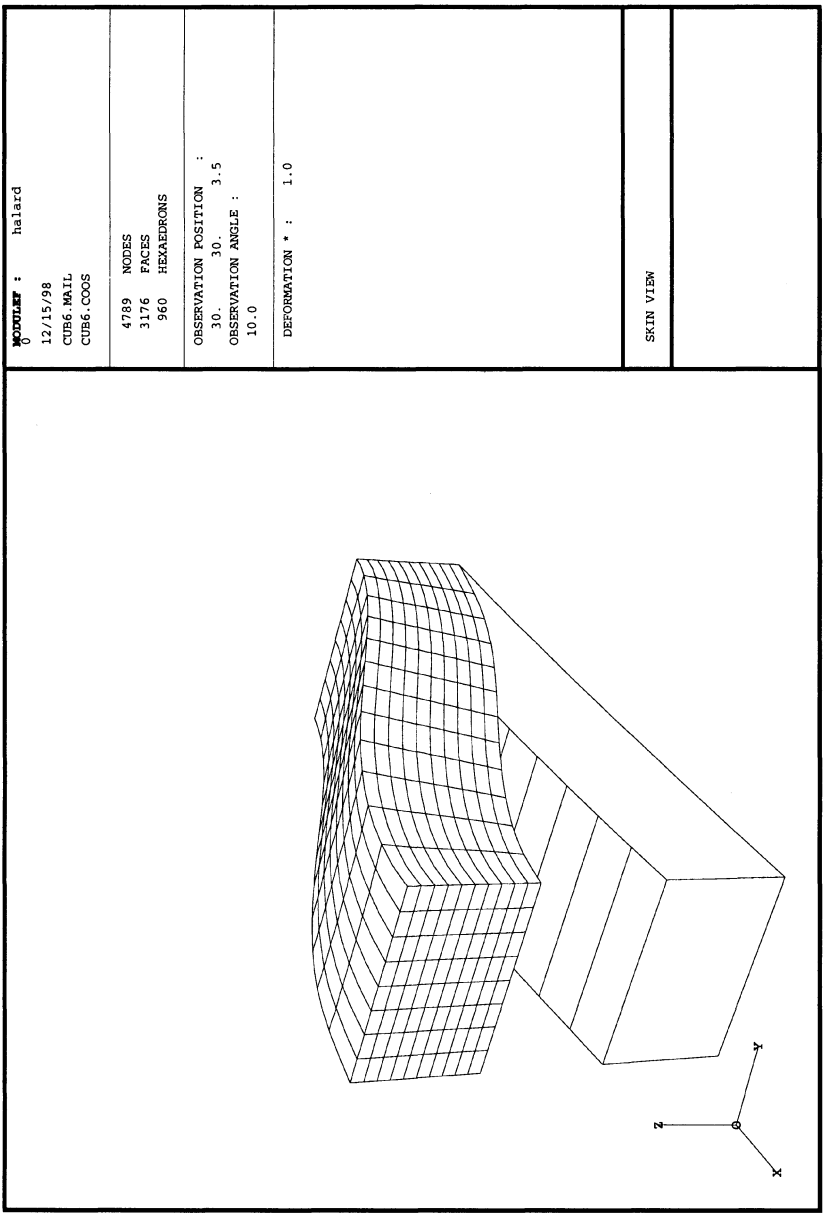


Figure 8.2. Initial and final shape of the reference beam. Calculation using the fine mesh with $NE = 960, p = 14367$. Taken from [20]

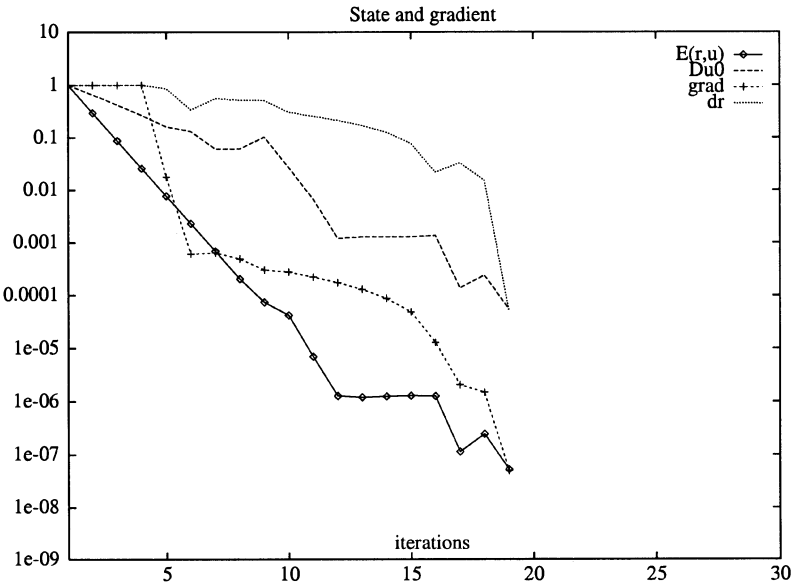


Figure 8.3. Convergence history of the one shot optimization loop. Unconstrained case used with $ICOST = 0$ and $ICURV = 1$. Taken from [20]

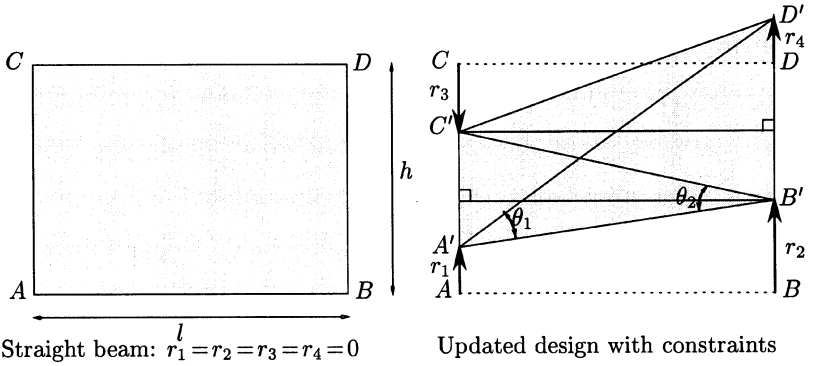


Figure 8.4. Geometric constraints on the aspect ratios within the mid section. The angles θ_i must be bounded from below and from above. Taken from [20]

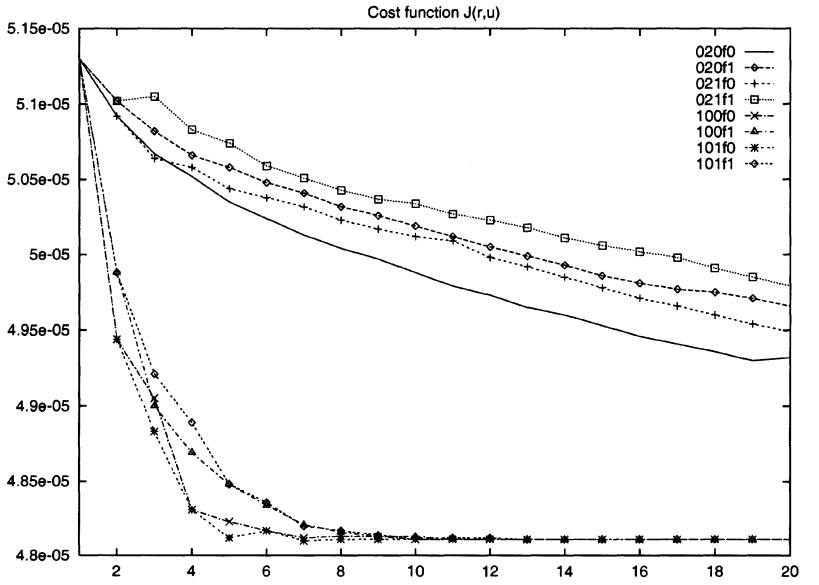


Figure 8.5. Convergence history of the cost function in the one shot optimization loop. Constrained case. Comparison between the standard line search strategy $ICURV = 0$ (top curves, used with different variants on $IFLEX$ and $ICOST$) and the improved linear search $ICURV = 1$ (bottom curves, used with different variants on $IFLEX$ and $ICOST$). Taken from [20]

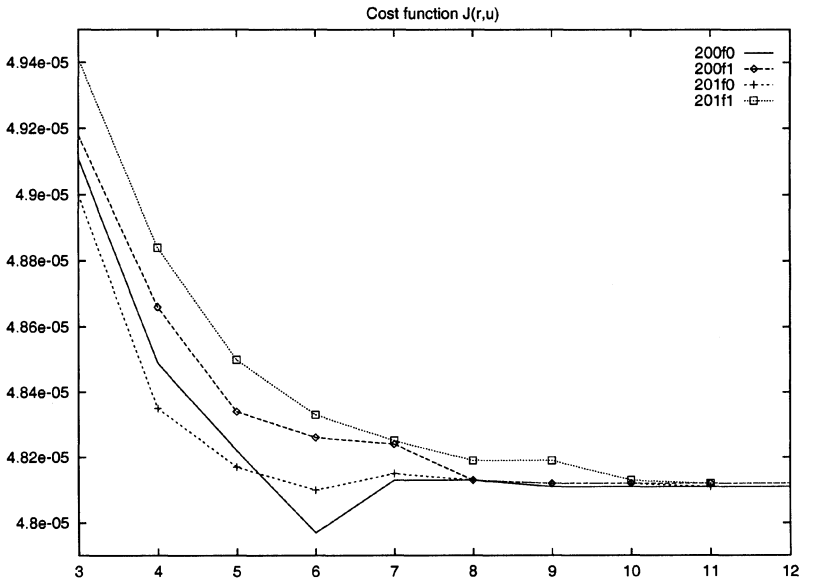


Figure 8.6. Convergence history of the cost function in the one shot optimization loop. Constrained case. Parabolic line search used with different variants on *IFLEX* and *ICOST*. Taken from [20]

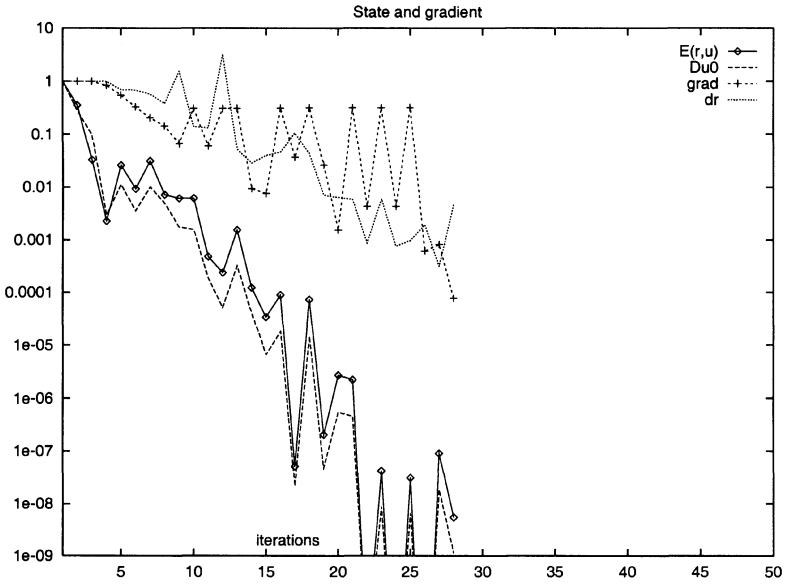


Figure 8.7. Convergence history of the gradient and residual in the one shot optimization loop. Constrained case. Linear line search used without deflection $IFLEX = 0$ and with $ICOST = 0$. Taken from [20]

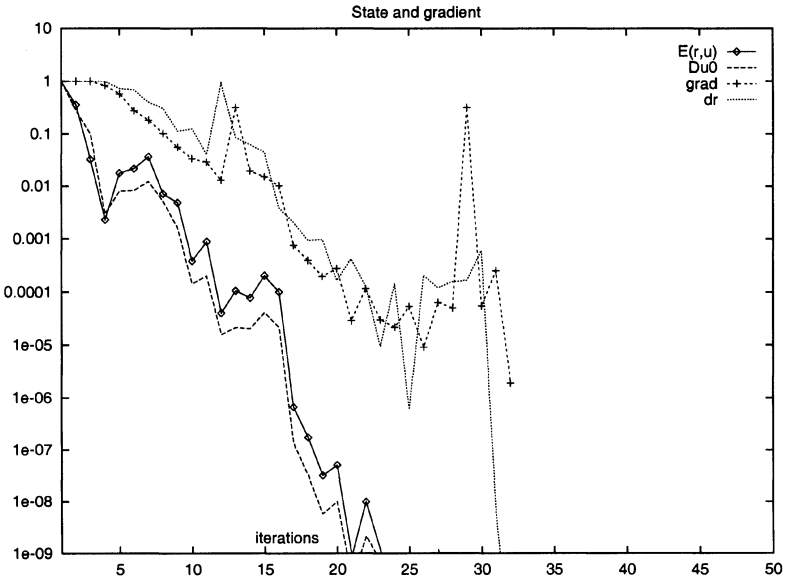


Figure 8.8. Convergence history of the gradient and residual in the one shot optimization loop. Constrained case. Linear line search used with deflection $IFLEX = 1$ and with $ICOST = 0$. Taken from [20]

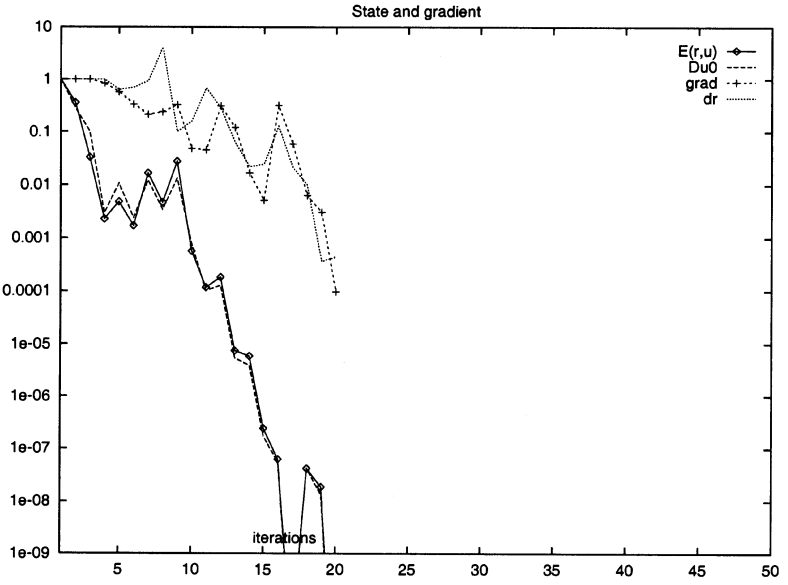


Figure 8.9. Convergence history of the gradient and residual in the one shot optimization loop. Constrained case. Parabolic line search used without deflection $IFLEX = 0$ and with $ICOST = 0$. Taken from [20]

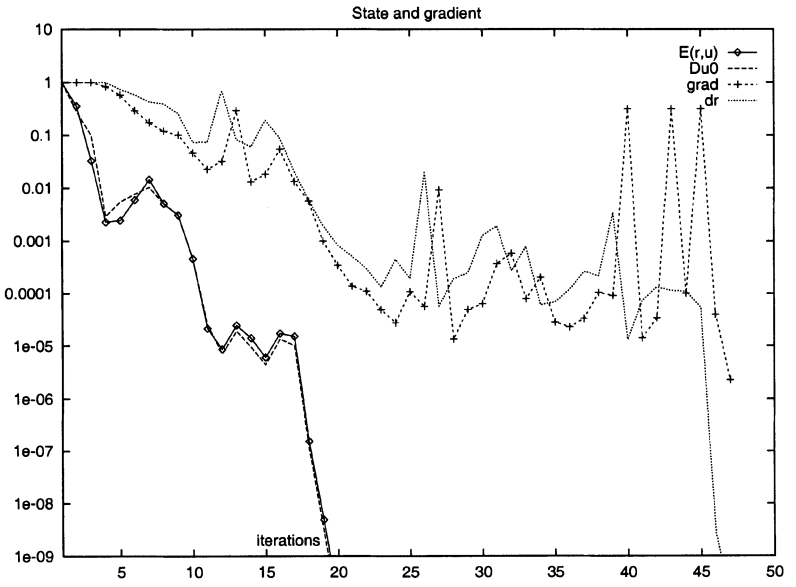


Figure 8.10. Convergence history of the gradient and residual in the one shot optimization loop. Constrained case. Parabolic line search used with deflection $IFLEX = 1$ and with $ICOST = 0$. Taken from [20]

Chapter 9

Conclusions

We have presented in this monograph a general and rather powerful practical numerical methodology for the local optimal design of a system whose behavior or performance is conditioned by its shape through the solution of a complex state equation. The main steps in this methodology consist in:

1. Choosing an adequate discretization strategy reducing the original optimization problem to a standard problem of mathematical programming in $\mathbb{R}^n$. The main tools needed here are a correct description and discretization of an initial shape to be used as reference configuration, an efficient control of the shape's deformation by spline interpolation of the contour normal's displacement, and a good mesh transformation strategy (Chapter 1).

2. Using an efficient constrained optimization method. Interior point algorithms solving the (Euler–Lagrange) optimality equations by a quasi-Newton algorithm while staying strictly inside the set of admissible designs are good candidates and turn out to be quite practical and efficient in shape optimization (Chapter 4).

3. Using a lagrangian approach to compute the gradients of the cost functions or of complex design constraints by adjoint state techniques (Chapter 6).

4. Generating the subroutines computing the adjoint variables or the different partial gradients of the Lagrangian by automatic code differentiation of the direct solvers (Chapter 5).

This approach is quite general. It also gives efficient and general tools for sensitivity analysis and has been illustrated in Chapter 7 on the optimization of several significant examples.

The individual tools are well documented in the literature, and the corresponding individual software can be found in commercial codes or as free software on the web. Our hope is that this monograph will help the reader to

E. Laporte et al., *Numerical Methods in Sensitivity Analysis and Shape Optimization*

assemble all these tools and to adapt them to his or her own optimization goals. The efficiency of the resulting approach will be mainly governed by:

- the relevance of the choice of the control parameters z and *cost function* j. A poorly designed cost function gives poor results. More generally, a complex behavior of the cost function with respect to the parameters z makes the optimization process lengthy and somewhat inefficient;
- the quality of the discretization (grid and scheme). The mesh deformation strategy should be robust and nevertheless provide a rather accurate discretization of the equations under study;
- the efficiency of the numerical solver used in the solution of the equations of state or of the adjoint equations of state;
- the performance of the initial shape.

The approach proposed herein remains nevertheless rather costly, especially when one needs to solve the state equation at each evaluation of the cost function. The cost is then directly related to the solution cost of the direct problem, to the solution cost of the adjoint problem, and to the memory required in running the subroutines obtained by automatic differentiation used in reverse mode. These costs are now affordable for the local optimization of large two-dimensional problems, or in the sensitivity analysis of large three-dimensional problems. They are still prohibitive when optimizing situations where the equations of state involve the solution of large three-dimensional viscous flow problems. In such cases, one can either use model reduction techniques which replace the state equation by simpler to solve approximations, or try to adapt the one shot methods briefly introduced in the last chapter. Both approaches are topics of intense research.

Appendix A

Subroutine cubspl

This annex contains a short version of the subroutine `cubspl` computing the cubic spline interpolation of the points $(\tau(i), c(1,i))_{i=1,n}$. More details can be found in [8].

```
      subroutine cubspl(tau,c,n)
c   input:
c         tau(i= 1:n)   position of  the interpolating points
c         c(1,i = 1:n)  value of the function at points tau(i)
c          c(2,1), c(2,n) value of end derivatives
c  output:
c          c(j=1:4, i = 1:n) spline coefficients.
      implicit none
      integer n,i,j,l,m
      double precision c(4,n),tau(n),divdf1,divdf3,dtau,g
c
      l = n-1
c
      do 10 m=2,n
        c(3,m) = tau(m) - tau(m-1)
        c(4,m) = (c(1,m) - c(1,m-1))/c(3,m)
 10   continue
c
      c(4,1) = 1.
      c(3,1) = 0.
c
      if (n .gt. 2) then
c
        do 20 m=2,l
          g = -c(3,m+1)/c(4,m-1)
          c(2,m) = g*c(2,m-1) + 3.*(c(3,m)*c(4,m+1)+
     &                              c(3,m+1)*c(4,m))
          c(4,m) = g*c(3,m-1) + 2.*(c(3,m) + c(3,m+1))
```

```
 20     continue
c
      endif
c
   30 do 40 j=1,1,-1
        c(2,j) = (c(2,j) - c(3,j)*c(2,j+1))/c(4,j)
 40   continue
c
      do 50 i=2,n
        dtau = c(3,i)
        divdf1 = (c(1,i) - c(1,i-1))/dtau
        divdf3 = c(2,i-1) + c(2,i) - 2.*divdf1
        c(3,i-1) = 2.*(divdf1 - c(2,i-1) - divdf3)/dtau
        c(4,i-1) = (divdf3/dtau)*(6./dtau)
 50   continue
c
      return
      end
```

Appendix B

Prototype Programmes for the Optimization Code

This appendix describes the main programme and the user defined subroutines used to run the interior point algorithm described in Chapter 4.

B.1 Main programme

```
      program main
c
      implicit double precision (a-h,o-z)
c
      parameter (nvar0   = 20  )
      parameter (ncstr0  = 50  )
      parameter (neq0    = 20  )
      parameter (nineq0  = 60  )
      parameter (ntcstr0 = 60  )
      parameter (nlmat0 =  70  )
      parameter (lenvlb0 = 20  )
      parameter (lenvub0 = 20  )
      parameter (liutil0 = 1   )
      parameter (lrutil0 = 1   )
c
c     All the previous parameters must be greater than zero and
c     nvar0 >= nvar, ncstr0 >= ncstr, ...
c
c     Input data on the box constraints
c
      integer          lvlb(nvar0),lvub(nvar0)
      double precision vlb(nvar0),vub(nvar0)
c
```

```
c     Input/output data
c
      double precision x(nvar0)
      double precision g(ncstr0)
      double precision lambda0(nineq0)
      double precision mu0(neq0)
c
c     Working arrays used in the key subroutines
c
      integer          ichsign(neq0)
      integer          iglow(lenvlb0),igup(lenvub0)
      integer          ivaux1(nlmat0)
c
      double precision gt(ntcstr0)
      double precision glow(lenvlb0),gup(lenvub0)
      double precision gradf(nvar0)
      double precision gradg(nvar0,ncstr0)
      double precision B(nvar0,nvar0)
      double precision lambda(nineq0),lambda1(nineq0)
      double precision omegaE(neq0),c(neq0)
      double precision omegaI(nineq0)
      double precision b0(nvar0+ntcstr0),b1(nvar0+ntcstr0)
      double precision d(nvar0),d0(nvar0),d1(nvar0)
      double precision mat((nvar0+ntcstr0),(nvar0+ntcstr0))
      double precision modg(ncstr0)
      double precision gpen(nvar0)
      double precision oldx(nvar0)
      double precision gdg(ncstr0)
      double precision gtl(ncstr0),gtr(ncstr0)
      double precision oldg(ncstr0)
      double precision oldgdg(ncstr0)
      double precision oldgf(nvar0)
      double precision oldgpen(nvar0)
      double precision gdgtl(ncstr0),gdgtr(ncstr0)
      double precision auxqn(nvar0),vaux1(nvar0),vaux2(nvar0)
      double precision vaux3(nlmat0)
c
c     fun.f computes the objective function and the response
c             constraints
c     gfun.f computes the derivatives
c
      external     fun,gfun
c
```

```
c     Working arrays to be used in the user's provided
c     subroutines fun or gfun
c
      integer          nprov
      double precision rutil(lrutil0)
      integer          iutil(liutil0)
c
c     Control parameters
c
      integer          idata(24)
      double precision data(9)
c
c     default parameters
c
      call fdata(data,idata)
c
c     STOP CRITERIA
c
c     - d0**2 less than  "eps-d02"
c
c     - The change of the objective was less than "eps-df" in the
c       last 3 iterations
c
c     - The square of the Lagrangian gradient is less than
c       "eps-glag"
c
c     - The number of iterates is equal to "itermax"
c
c     - The number of iterates is greater than "itermin"
c
c     LINE SEARCH
c
c     ilin = 1 - Wolfe's criterium with cubic interpolation
c     ilin = 0-  Armijo's search
c
      ilin=1
c
c     LINE SEARCH STOP CRITERIA
c
c     - If the number of iterates in line search is greater than
c       "Itlinmax" the iterates stops and gives an error message.
c
       epsd02 =1.0d-8
```

```
      epsdf  =1.0d-8
      epsglag=1.0d-8
      itermin=5
      itermax =100
      itlinmax=10
c
      nvar  =3
      ncstr =2
      neq   =1
      neqlin=1
      lenvlb=3
      lenvub=0
c
c       nvar    - number of variables
c       ncstr   - number of response constraints
c       neq     - number of equality contraints
c       neqlin  - number of linear equality constraints
c                 (neqlin.le.neq !!!)
c       lenvlb  - number of lower bounds on the variables
c       lenvub  - number of upper bounds on the variables
c
      ntcstr=ncstr+lenvlb+lenvub
      nineq =ntcstr-neq
      nlmat=ntcstr+nvar
c
c  Initial estimate of the variables
c
      x(1)=0.1d0
      x(2)=0.7d0
      x(3)=0.2d0
c
c  Vector of lower bounds
c
c     vlb(i) is the lower bound on x(i)
c
      vlb(1)=0.d0
      vlb(2)=0.d0
      vlb(3)=0.d0
c
c  Vector of upper bounds
c
c     vub(i) is the upper bound on x(i)
c     In this example there are no upper bounds
```

```
c
c       vub(1)=999.d+10
c       vub(2)=999.d+10
c       vub(3)=999.d+10
c
c  lvlb(i) =1: x(i) is bounded below
c  lvlb(i) =0: x(i) is not bounded below
c
        lvlb(1)=1
        lvlb(2)=1
        lvlb(3)=1
c
c  lvub(i) =1: x(i) is bounded above
c  lvub(i) =0: x(i) is not bounded above
c
        lvub(1)=0
        lvub(2)=0
        lvub(3)=0
c
         if(nvar.gt.nvar0 .or. ncstr.gt.ncstr0 .or.
     &      ntcstr.gt.ntcstr0 .or. neq.gt.neq0 .or.
     &      nineq.gt.nineq0 .or. lenvlb.gt.lenvlb0 .or.
     &      lenvub.gt.lenvub0 .or. nlmat.gt. nlmat0) then
            stop 'error, wrong parameters'
         endif
c
       call ipa(nvar,ncstr,ntcstr,neq,neqlin,nineq,lenvlb,lenvub,
     &          nprob,fun,gfun,rutil,iutil,lrutil,liutil,
     &          x,vlb,vub,lvlb,lvub,data,idata,
     &          f,g,gt,B,gradg,
     &          lambda,lambda0,lambda1,omegaE,omegaI,
     &          glow,gup,iglow,igup,gradf,oldgf,c,mu0,ichsign,
     &          modg,d,d0,d1,b0,b1,mat,nlmat,gpen,
     &          oldx,gdg,gtl,gtr,oldg,oldgdg,oldgpen,gdgtl,gdgtr,
     &          auxqn,vaux1,vaux2,vaux3,ivaux1,
     &          epsd02,epsdf,epsglag,ilin,itermin,itermax,
     &          itlinmax)
c
c     In soubroutine ipa:
c
c     x: input/ initial design variable, output/ optimum design
c                variables
c
```

```
c     f: output/ final objective function
c
c     g: output/ final constraints vector
c
c     lambda0: output/ final Lagrange Multipliers of inequality
c              constraints
c
c     mu0: output/ final Lagrange Multipliers of equality
c              constraints
c
      write(*,*)'x = ',x
      write(*,*)'f = ',f
      write(*,*)'g = ',g
      write(*,*)'lambda = ',lambda0
      write(*,*)'mu = ',mu0
      write(*,*)'# END #'
c
      end
```

B.2 Cost function and constraints

```
      subroutine fun (f,g,x,nprov,rutil,iutil,lrutil,liutil)
c
c  fun - computes f and g
c
c  f is the objective function
c  g(ncstr) is a vector with the response contraints ordered as
c  follows:
c
c  g(1),...,g(neqlin)     - Linear equality constraints
c  g(neqlin+1),...,g(neq) - Nonlinear equality constraints
c  g(neq+1),...,g(ncstr)  - Inequality constraints (it does not
c                           include box constraints)
c
c  iutil(liutil) - Integer utility vector, employed to store data
c                  in probXX, funXX and gfunXX.
c
c  rutil(lrutil) - Real double precision utility vector, employed
c                  to store data in probXX, funXX and gfunXX.
c
      integer          iutil(liutil)
      double precision rutil(lrutil)
c
      double precision x(3),f,g(2)
c
```

```
      f=(x(1)+3.d0*x(2)+x(3))**2+4.d0*(x(1)-x(2))**2

      g(1)=1.d0-x(1)-x(2)-x(3)
      g(2)=-x(1)**3+6.d0*x(2)+4.d0*x(3)-3.d0
c
      do 102 i=1,2
        g(i)=-g(i)
  102 continue
c
      return
      end
```

B.3 Gradients

```
      subroutine gfun(df,gradg,x,nprov,rutil,iutil,lrutil,
     &                liutil)
c
c  gfun - Computes derivatives of f and g
c
c  df is the derivative of f
c  dg(nvar,ncstr) is the jacobian of g
c
c  iutil(liutil) - Integer utility vector, employed to store data
c                  in probXX, funXX and gfunXX.
c
c  rutil(lrutil) - Real double precision utility vector, employed
c                  to store data in probXX, funXX and gfunXX.
c
      integer          iutil(liutil)
      double precision rutil(lrutil)
c
      double precision x(3)
      double precision df(3)
      double precision gradg(3,2)
c
      df(1)=10.d0*x(1)-2.d0*x(2)+2.d0*x(3)
      df(2)=-2.d0*x(1)+26.d0*x(2)+6.d0*x(3)
      df(3)=2.d0*(x(1)+3.d0*x(2)+x(3))
c
      gradg(1,1)=1.0d0
      gradg(2,1)=1.0d0
      gradg(3,1)=1.0d0
      gradg(1,2)=3.d0*x(1)**2
      gradg(2,2)=-6.d0
      gradg(3,2)=-4.d0
c
```

```
      return
      end
```

B.4 Typical values of control parameters

The subroutine fdata defines the default values.

```
      subroutine fdata(data,idata)
c      data(2)  # stopping criterium for the residual update
c               # d0^2
c      data(5)  # alpha, cf. equation (4.7) page 62
c      idata(11)  # 1,5,10 - printing level (1 prints nothing,
c                 #                         10 prints too much)
c      idata(12)  # 1- uses normed values of g and gradg;
c                 #           0- no normalization
c      idata(13)  # 1- assume linear evolution of Lagrangian
c                 # parameters to compute the first step for
c                 # the linear search
c      idata(14)  # 1- assume linear inequality constraints to
c                 # compute the first step for the linear search
c      idata(16)  # Lagrangian parameters update method
c                 # 0-dikin; 1-newton (cf. [22])
c      idata(17)  # matrix B update method: 1- Quasi-Newton
c                 #                         0- Identity
      integer          idata(24)
      double precision data(9)
c
c  Set real parameters to zero
c
      call dzerov(data,9)
c
      data(2)=1.0d-4
      data(5)=1.0d0
      do 100 i=1,24
        idata(i)=0
  100 continue
c
      idata(11)=1
      idata(12)=0
      idata(13)=0
      idata(14)=0
      idata(16)=1
      idata(17)=1
c
      return
      end
```

Appendix C

O∂*y*ssée User's Manual (short version)

This appendix presents a few basic instructions for running the automatic code differentiator O∂*y*ssée (see [14] for the complete manual):

- `load < file1> [< file2> ... < fileN> ]` loads the files file1, file2, ..., containing the Fortran units (subroutines or functions) to be differentiated;
- `listunits` displays the names and types of units which have been loaded with `load`;
- `getgraph < unit>` displays all dependencies between the different units originating from the unit given in argument;
- `getprogram < unit>` displays all units belonging to the dependencies originating from the unit given in argument;
- `getunit < unit> [< file> ]` displays the unit given in argument in the given file, or at the screen if there are no given file;
- `getinout < unit>` gives the input and output variables of the unit given in argument;
- `diff -h < unit> -tl|-cl -vars < x1> [< x2> ... < xN>]` differentiates the unit given in argument either in linear tangent mode (`-tl`) or in inverse mode (`-cl`) with respect to the variables x1, x2, ...

Appendix D

A Subroutine computing the Gradient with respect to the Grid for the Steady Aerodynamic Example

D.1 A typical aerodynamic cost function

The cost function itself

The subroutine below computes the pressure drag by the technique introduced in 7.3.4. The aerodynamic variable ua is given here in input.

```
      subroutine fcout(ns,coor,ua,logfr,coutj,teta,nseg,nubo)
c
      implicit none
      integer ns, nseg, nubo(2,nseg)
      real coutj, pres1, pres2, teta, cn, ct
      integer logfr(ns), is1, is2, iseg
      real gam, gam1, ua(4,ns)
      real coor(2,ns), delx, dely
c
      cn=0.
      ct=0.
c
      gam=1.4
      gam1=gam-1.0
      do iseg=1,nseg
        is1=nubo(1,iseg)
        is2=nubo(2,iseg)
        if ((logfr(is1) .eq. 2) .and. (logfr(is2).eq.2)) then
```

```
          pres1 =
     &     gam1*(ua(4,is1)-(0.5*(ua(2,is1)**2+ua(3,is1)**2))/
     &          ua(1,is1))
          pres2 =
     &     gam1*(ua(4,is2)-(0.5*(ua(2,is2)**2+ua(3,is2)**2))/
     &          ua(1,is2))
          delx=coor(1,is1)-coor(1,is2)
          dely=coor(2,is1)-coor(2,is2)
          cn=cn+delx*(pres1+pres2)
          ct=ct-dely*(pres1+pres2)
        endif
      enddo
      coutj=cn*sin(teta)+ct*cos(teta)
c
      return
      end
```

Gradient obtained by O∂yssée

The following subroutine has been obtained by O∂yssée after automatic differentiation of the previous subroutine in adjoint (reverse mode) with respect to the variables coor and ua.

```
      SUBROUTINE fcoutcl (ns, coor, ua, logfr, coutj, teta, nseg,
     & nubo, coorccl, uaccl, coutjccl)

      IMPLICIT NONE
      INTEGER ns, nseg, nubo(2,nseg)
      REAL coutj, pres1, pres2, teta, cn, ct
      INTEGER logfr(ns), is1, is2, iseg
      REAL gam, gam1, ua(4,ns)
      REAL coor(2,ns), delx, dely
      REAL sr01s, sr02s
      INTEGER odynseg
      PARAMETER (odynseg = 10)
      REAL coutjccl
      REAL coorccl(2,ns)
      REAL uaccl(4,ns)
      REAL delxccl
      REAL save16
      REAL save5
      REAL save4
      REAL save3
      REAL pres1ccl
      REAL save2
      REAL delyccl
      REAL save1
```

```
      REAL cnccl
      REAL pres2ccl
      REAL save18
      REAL ctccl
      REAL save17
      INTEGER save7(1:odynseg)
      REAL save15(1:odynseg)
      LOGICAL test9(1:odynseg)
      INTEGER save6(1:odynseg)
      REAL save14(1:odynseg)
      REAL save13(1:odynseg)
      REAL save12(1:odynseg)
      REAL save11(1:odynseg)
      REAL save10(1:odynseg)

C
C Initializations of local variables
C

      cnccl = 0.
      delyccl = 0.
      delxccl = 0.
      pres2ccl = 0.
      pres1ccl = 0.
      ctccl = 0.
C
C Trajectory
C

      save1 = coutj
      save2 = cn
      cn = 0.
      save3 = ct
      ct = 0.
      save4 = gam
      gam = 1.4
      save5 = gam1
      gam1 = gam-1.0
      DO iseg = 1, nseg
        save6(iseg) = is1
```

```
        is1 = nubo(1,iseg)
        save7(iseg) = is2
        is2 = nubo(2,iseg)
        test9(iseg) = (logfr(is1).EQ.2).AND.(logfr(is2).EQ.2)
        IF (test9(iseg)) THEN
          save10(iseg) = pres1
          pres1 =
     &     gam1*(ua(4,is1)-(0.5*(ua(2,is1)**2+ua(3,is1)**2))/ua(1
     &    ,is1))
          save11(iseg) = pres2
          pres2 =
     &     gam1*(ua(4,is2)-(0.5*(ua(2,is2)**2+ua(3,is2)**2))/ua(1
     &    ,is2))
          save12(iseg) = delx
          delx = coor(1,is1)-coor(1,is2)
          save13(iseg) = dely
          dely = coor(2,is1)-coor(2,is2)
          save14(iseg) = cn
          cn = cn+delx*(pres1+pres2)
          save15(iseg) = ct
          ct = ct-dely*(pres1+pres2)
        END IF
      END DO
      save16 = sr01s
      sr01s = sin(teta)
      save17 = sr02s
      sr02s = cos(teta)
      save18 = coutj
      coutj = cn*sr01s+ct*sr02s
C
C Transposed linear forms
C

      coutj = save18
      cnccl = cnccl+coutjccl*sr01s
      ctccl = ctccl+coutjccl*sr02s
      coutjccl = 0.
      coutjccl = 0.
      sr02s = save17
      sr01s = save16
      DO iseg = nseg, 1, -1
        IF (test9(iseg)) THEN
          ct = save15(iseg)
```

```
      pres1ccl = pres1ccl-ctccl*dely
      pres2ccl = pres2ccl-ctccl*dely
      delyccl = delyccl-ctccl*(pres1+pres2)
      cn = save14(iseg)
      pres1ccl = pres1ccl+cnccl*delx
      pres2ccl = pres2ccl+cnccl*delx
      delxccl = delxccl+cnccl*(pres1+pres2)
      dely = save13(iseg)
      coorccl(2,is1) = coorccl(2,is1)+delyccl
      coorccl(2,is2) = coorccl(2,is2)-delyccl
      delyccl = 0.
      delx = save12(iseg)
      coorccl(1,is1) = coorccl(1,is1)+delxccl
      coorccl(1,is2) = coorccl(1,is2)-delxccl
      delxccl = 0.
      pres2 = save11(iseg)
      uaccl(4,is2) = uaccl(4,is2)+pres2ccl*gam1
      uaccl(2,is2) =
&      uaccl(2,is2)-pres2ccl*(((1./ua(1,is2)**2)*(((2*ua(2,is
&     2))*0.5)*ua(1,is2)))*gam1)
      uaccl(3,is2) =
&      uaccl(3,is2)-pres2ccl*(((1./ua(1,is2)**2)*(((2*ua(3,is
&     2))*0.5)*ua(1,is2)))*gam1)
      uaccl(1,is2) =
&      uaccl(1,is2)+pres2ccl*(((1./ua(1,is2)**2)*(0.5*(ua(2,i
&     s2)**2+ua(3,is2)**2)))*gam1)
      pres2ccl = 0.
      pres1 = save10(iseg)
      uaccl(4,is1) = uaccl(4,is1)+pres1ccl*gam1
      uaccl(2,is1) =
&      uaccl(2,is1)-pres1ccl*(((1./ua(1,is1)**2)*(((2*ua(2,is
&     1))*0.5)*ua(1,is1)))*gam1)
      uaccl(3,is1) =
&      uaccl(3,is1)-pres1ccl*(((1./ua(1,is1)**2)*(((2*ua(3,is
&     1))*0.5)*ua(1,is1)))*gam1)
      uaccl(1,is1) =
&      uaccl(1,is1)+pres1ccl*(((1./ua(1,is1)**2)*(0.5*(ua(2,i
&     s1)**2+ua(3,is1)**2)))*gam1)
      pres1ccl = 0.
    END IF
    is2 = save7(iseg)
    is1 = save6(iseg)
  END DO
```

```
      pres1ccl = 0.
      pres2ccl = 0.
      delxccl = 0.
      delyccl = 0.
      cnccl = 0.
      ctccl = 0.
      gam1 = save5
      gam = save4
      ct = save3
      cn = save2
      coutj = save1
      RETURN
      END
```

Optimized gradient

The subroutine below is a manually optimized version of the previous subroutine obtained by O∂yssée by automatic differentiation of the initial drag calculation subroutine in adjoint (reverse mode) with respect to the variables coor and ua. A few intermediate variables and calculations have been suppressed.

```
      SUBROUTINE fcoutcl (ns, coor, ua, logfr, teta, nseg,
     & nubo, coorccl, uaccl, coutjccl)

      IMPLICIT NONE
      INTEGER ns, nseg, nubo(2,nseg)
      REAL pres1, pres2, teta
      INTEGER logfr(ns), is1, is2, iseg
      REAL gam, gam1, ua(4,ns)
      REAL coor(2,ns), delx, dely
      REAL coutjccl
      REAL coorccl(2,ns)
      REAL uaccl(4,ns)
      REAL delxccl
      REAL pres1ccl
      REAL delyccl
      REAL cnccl
      REAL pres2ccl
      REAL ctccl

C
C Initializations of local variables
C

      delyccl = 0.
      delxccl = 0.
```

```
      pres2ccl = 0.
      pres1ccl = 0.
C
C Trajectory
C

      gam = 1.4
      gam1 = gam-1.0
C
C Transposed linear forms
C

      cnccl = coutjccl*sin(teta)
      ctccl = coutjccl*cos(teta)
      coutjccl = 0.
      DO iseg = nseg, 1, -1
        is1 = nubo(1,iseg)
        is2 = nubo(2,iseg)
        IF ((logfr(is1).EQ.2).AND.(logfr(is2).EQ.2)) THEN
          pres1 =
     &     gam1*(ua(4,is1)-(0.5*(ua(2,is1)**2+ua(3,is1)**2))/ua(1
     &    ,is1))
          pres2 =
     &     gam1*(ua(4,is2)-(0.5*(ua(2,is2)**2+ua(3,is2)**2))/ua(1
     &    ,is2))
          delx = coor(1,is1)-coor(1,is2)
          dely = coor(2,is1)-coor(2,is2)
          pres1ccl = pres1ccl-ctccl*dely
          pres2ccl = pres2ccl-ctccl*dely
          delyccl = delyccl-ctccl*(pres1+pres2)
          pres1ccl = pres1ccl+cnccl*delx
          pres2ccl = pres2ccl+cnccl*delx
          delxccl = delxccl+cnccl*(pres1+pres2)
          coorccl(2,is1) = coorccl(2,is1)+delyccl
          coorccl(2,is2) = coorccl(2,is2)-delyccl
          delyccl = 0.
          coorccl(1,is1) = coorccl(1,is1)+delxccl
          coorccl(1,is2) = coorccl(1,is2)-delxccl
          delxccl = 0.
          uaccl(4,is2) = uaccl(4,is2)+pres2ccl*gam1
          uaccl(2,is2) =
     &     uaccl(2,is2)-pres2ccl*ua(2,is2)*gam1/ua(1,is2)
```

```
          uaccl(3,is2) =
     &     uaccl(3,is2)-pres2ccl*ua(3,is2)*gam1/ua(1,is2)
          uaccl(1,is2) =
     &     uaccl(1,is2)+pres2ccl*((1./ua(1,is2)**2)*(0.5*(ua(2,is
     &    2)**2+ua(3,is2)**2)))*gam1
          pres2ccl = 0.
          uaccl(4,is1) = uaccl(4,is1)+pres1ccl*gam1
          uaccl(2,is1) =
     &     uaccl(2,is1)-pres1ccl*ua(2,is1)*gam1/ua(1,is1)
          uaccl(3,is1) =
     &     uaccl(3,is1)-pres1ccl*ua(3,is1)*gam1/ua(1,is1)
          uaccl(1,is1) =
     &     uaccl(1,is1)+pres1ccl*((1./ua(1,is1)**2)*(0.5*(ua(2,is
     &    1)**2+ua(3,is1)**2)))*gam1
          pres1ccl = 0.
        END IF
      END DO
      RETURN
      END
```

D.2 A full subroutine for computing the gradient with respect to the grid

The programme below implements the complete algorithm for computing the cost function and its gradient, using the methodology introduced in 7.3.4. It is based on a simplified version of the NSC2KE software, which only computes inviscid flows. The subroutine computing the adjoint states at the end have been obtained by automatic differentiation.

```
      subroutine nsc2 (ns, nt, nu, nuedg, nseg, nubo, nfr, nufr,
     & nslog2, log2, nslog5, log5, logfr, coor, coutj, grad,
     & nordre, ktmax, roin, ruxin, ruyin, ein, cfl, resf, teta)
c
      implicit none
c
c  Input data
c
      integer ns, nt, nu(3,nt), nuedg(3,nt), nseg, nubo(2,nseg),
     & nfr, nufr(nfr), nslog2, log2(nslog2), nslog5,
     & log5(nslog5), logfr(ns)
      real coor(2,ns)
      integer nordre, ktmax
      real roin, ruxin, ruyin, ein, cfl, resf, teta
c
c  Output data
```

```
c
      real coutj, grad(2,ns)
c
c  Local Variables
c
      integer nn, nnt, nnsg
      parameter (nn = 15000, nnt = 2*nn, nnsg = 3*nn)
      integer nvar
      parameter (nvar = 4)
      real airs(nn), airt(nnt), vno(nn), vnox(nn), vnoy(nn),
     & vnocl(nnsg)
      real un(nvar,nn), ua(nvar,nn)
      integer is, ivar, kt
      real som
      real coorccl(2,nn), uaccl(nvar,nn), unccl(nvar,nn),
     & djdw(nvar,nn), djdx(2,nn), tmp(nvar,nn)
      real airsccl(nn), airtccl(nn), vnoclccl(3,nnsg),
     & vnoccl(nn), vnoxccl(nn),v noyccl(nn)
c
c  Calculation of the local geometric characteristics of the grid
c
c     - calculation of the volumes of the control cells and of
c       the triangles
c
      call aires (ns, nt, nu, coor, airs, airt)
c
c     - calculation of the  normal
c
      call seg2dr (ns, nt, coor, logfr, nu, vnocl, nseg, nubo,
     & vno, vnox, vnoy, nufr, nuedg, nfr)
c
c  Initialization of the state vector
c
      do is=1,ns
        ua(1,is)=roin
        ua(2,is)=ruxin
        ua(3,is)=ruyin
        ua(4,is)=ein
      enddo
c
      kt=0
c
```

```
c  Saving the present value of the state variable
c
      do is=1,ns
        do ivar=1,nvar
          un(ivar,is)=ua(ivar,is)
        enddo
      enddo
c
c  Iterations in time
c
100   kt=kt+1
c
      call resiter (ns, nt, nseg, coor, airs, nu, nubo, vnocl,
     & vno, vnox, vnoy, un, ua, nordre, roin, ruxin, ruyin, ein,
     & cfl, nslog2, log2, nslog5, log5)
c
      do is=1,ns
        do ivar=1,nvar
c
c  Calculation of the total residual
c
          som=som+(ua(ivar,is)-un(ivar,is))**2.
c
c Saving the present value of the state variable
c
          un(ivar,is)=ua(ivar,is)
        enddo
      enddo
      som=sqrt(som)/float(ns)
c
c  Stopping tests for the loop in time
c
      if (kt  .eq. ktmax) then
c
c  end by reaching the maximal number of time steps
c
        goto 300
      endif
      if (som .lt. resf) then
c
c  end by getting a small residual
c
        goto 300
```

```
      endif
      goto 100
300   continue
c
c  Calculation of the cost function
c
      call fcout (ns, coor, ua, logfr, coutj, teta, nseg, nubo)
c
c  Calculation of the gradients  djdx and djdw
c
      do is=1,ns
        djdx(1,is)=0.
        djdx(2,is)=0.
        do ivar=1,nvar
          djdw(ivar,is)=0.
        enddo
      enddo
      call fcoutcl (ns, coor, ua, logfr, teta, nseg,
     & nubo, djdx, djdw, 1.)
c
c  Initialization of the adjoint state
c
      do is=1,ns
        do ivar=1,nvar
          unccl(ivar,is)=djdw(ivar,is)
        enddo
      enddo
c
      kt=0
c
c  Adjoint loop
c
500   kt=kt+1
c
      do is=1,ns
        coorccl(1,is)=0.
        coorccl(2,is)=0.
        airsccl(is)=0.
        vnoccl(is)=0.
        vnoxccl(is)=0.
        vnoyccl(is)=0.
      enddo
      do is=1,nseg
```

```
        vnoclccl(1,is)=0.
        vnoclccl(2,is)=0.
        vnoclccl(3,is)=0.
      enddo
      do is=1,ns
        do ivar=1,nvar
          uaccl(ivar,is)=unccl(ivar,is)
          tmp(ivar,is)  =unccl(ivar,is)
          unccl(ivar,is)=djdw(ivar,is)
        enddo
      enddo
c
      call resitercl (ns, nt, nseg, coor, airs, nu, nubo, vnocl,
     & vno, vnox, vnoy, un, ua, nordre, roin, ruxin, ruyin, ein,
     & cfl, nslog2, log2, nslog5, log5, coorccl, airsccl,
     & vnoclccl, vnoccl, vnoxccl, vnoyccl, unccl, uaccl)
c
c  Calculating the residual in the adjoint loop
c
      do is=1,ns
        do ivar=1,nvar
          som=som+(unccl(ivar,is)-tmp(ivar,is))**2.
        enddo
      enddo
      som=sqrt(som)/float(ns)
c
c Stopping tests for the adjoint  loop
c
      if (kt  .eq. ktmax) then
c
c end by reaching the maximal number of time steps
c
        goto 700
      endif
      if (som .lt. resf) then
c
c end by getting a small residual
c
        goto 700
      endif
      goto 500
700   continue
c
```

```
c  Differentiation with respect to the geometry
c
      call seg2drcl (ns, nt, coor, logfr, nu, vnocl, nseg, vno,
     & vnox, vnoy, nufr, nuedg, nfr, coorccl, vnoclccl, nubo,
     & vnoccl, vnoxccl, vnoyccl)
c
      call airescl (ns, nt, nu, coor, airs, airt, coorccl,
     & airsccl, airtccl)
c
c              final calculation of the gradient
c
      do is=1,ns
        grad(1,is)=coorccl(1,is)+djdx(1,is)
        grad(2,is)=coorccl(2,is)+djdx(2,is)
      enddo
c
      return
      end
```

Bibliography

[1] L. Armijo. Minimisation of functions having Lipschitz continuous first partial derivatives. *Pacific J. Mathematics*, 16:1–3, 1966.

[2] K.J. Arrow, L. Hurwicz, and H. Uzawa. *Studies in Nonlinear Programming.* Stanford University Press, 1958.

[3] F. J. Baron, G. Duffa, F. Carrere, and P. Le Tallec. Optimisation de forme en aérodynamique. *CHOCS, Revue scientifique et technique de la Direction des Applications Militaires du CEA*, 1994.

[4] M.P. Bendsoe and N. Kikuchi. Generating optimal topologies in structural design using a homogenisation method. *Comp. Meth. Appl. Mech. Eng.*, pages 197–224, 1988.

[5] J.F. Bonnans, J.C. Gilbert, C. Lemaréchal, and C. Sagastizabal. *Optimisation Numérique.* Springer Verlag, 1998.

[6] J. Cea. *Optimisation : théorie et algorithmes.* Dunod, Paris, 1971.

[7] R. Dautray and J.L. Lions. *Analyse mathématique et calcul numérique pour les sciences et les techniques. Vol. 1.* INSTN: Collection Enseignement. Masson, Paris, 1987. Modèles physiques, with the collaboration of Michel Cessenat, André Gervat and Hélène Lanchon, Reprinted from the 1984 edition.

[8] C. de Boor. *A Practical Guide to Splines*, volume 27 of *Applied Mathematical Sciences.* Springer-Verlag, New York-Berlin, 1998.

[9] J.E. Dennis and R.B. Schnabel. *Numerical Methods for Unconstrained Opimization and Nonlinear Equations.* Prenctice Hall, 1983.

[10] P. Deuflhard. Global inexact Newton methods for very large scale nonlinear problems. In S. Mc Cormick, editor, *Proceedings of the Copper Mountain Conference*, April 1990.

[11] S. C. Eisenstat, H.C. Elman, and M. Schultz. Variational iterative method for nonsymmetric systems of equations. *SIAM J. Numer. Anal.*, 20:345–357, 1983.

[12] A.S. El-Bakry, R.A. Tapia, T. Tsuchiya, and Y. Zhang. On the formulation and theory of the primal dual Newton interior point method for nonlinear programming. *J. Opt. Theory and Appl.*, 89:507–541, 1996.

[13] F. Eyssette, C. Faure, J.-C. Gilbert, and N. Rostaing-Schmidt. Applicability of automatic differentiation on a system of partial differential equations governing thermohydraulic phenomena. Research Report RR-2795, INRIA Sophia Antipolis, 1996.

[14] C. Faure and Y. Papegay. O∂yssée version 1.6 — manuel de l'utilisateur. Rapport Technique RT-0211, INRIA Rocquencourt, Novembre 1997.

[15] C. Fleury and V. Braibant. Structural optimisation: a new dual method using mixed variables. *Int. J. Num. Meth. Eng.*, pages 409–428, 1986.

[16] B. Van Leer G. D. Van Albada. Flux vector splitting and Runge–Kutta methods for the euler equations. *ICASE 84-27*, June 1984.

[17] J. C. Gilbert, G. Le Vey, and J. Masse. La différentiation automatique de fonctions representées par des programmes. Rapport de Recherche RR1557, INRIA Rocquencourt, 1991.

[18] P. Gill, W. Murray, and M. Wright. *Practical Optimisation.* Academic Press, 1981.

[19] G. Golub and C. Van Loan. *Matrix Computations.* North Oxford Academic, 1983.

[20] M. Halard. *Conception Optimale de Formes en Elasticité Nonlinéaire.* Thèse, Université de Paris Dauphine, October 1999.

[21] J. Herskovits. A two-stage feasible directions algorithm for nonlinear constrained optimization. *Mathematical Programming*, pages 19–38, 1986.

[22] J. Herskovits. An interior point technique for nonlinear optimization. Rapport de Recherche RR1808, INRIA Rocquencourt, 1992.

[23] J. Herskovits. *Advances in Structural Optimization : A View on Nonlinear Optimization.* Kluwer Academic Publishers, 1995.

[24] J. Herskovits, E. Laporte, P. Le Tallec, and G. Santos. A quasi-newton interior point algorithm applied to constrained optimum design in computational fluid dynamics. *Revue Européenne des Éléments Finis*, 5(5-6):595–517, 1996.

[25] M. Hestenes. Multiplier and gradient methods. *J. Opt. Theory Apl.*, 4:303–320, 1969.

[26] W. Hoyer. Variants of the reduced newton method for nonlinear equality constrained optimization problems. *Optimization*, 7:757–774, 1986.

[27] T.J.R Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis.* Prenctice Hall, 1987.

[28] H.B. Keller. The bordering algorithm and path following near singular points of higher nullity. *SIAM J. Sci. Stat. Comput.*, 4:573–582, 1983.

[29] M. Kleiber, H. Antumez, H.D. Tien, and P. Kowalczyk. *Parameter Sensitivity in Nonlinear Mechanics.* J. Wiley, New York, London, 1997.

[30] M.-H. Lallemand. *Schémas décentrés multigrilles pour la résolution des équations d'Euler en éléments finis.* Thèse, Université de Provence, Centre Saint Charles, March 1998.

[31] E. Laporte. *Optimisation de formes pour écoulements instationnaires.* Thèse, Ecole Polytechnique, September 1998.

[32] E. Laporte and P. Le Tallec. Shape optimisation in unsteady flows. Rapport de Recherche RR-3693, INRIA, Mai 1999.

[33] A. Marrocco. Simulations numériques dans la fabrication des circuits à semiconducteurs (process modelling). Rapport de Recherche RR-0305, INRIA Rocquencourt, mai 1984.

[34] M. Masmoudi. *Outils pour la Conception Optimale de Formes.* Thèse d'état, Université de Nice, 1987.

[35] M. Minoux. *Programmation mathématique : théorie et algorithmes.* Dunod, Paris, 1988.

[36] B. Mohammadi. Fluid dynamics computation with nsc2ke: a user-guide: release 1.0. Rapport Technique RT0164, INRIA Rocquencourt, 1994.

[37] B. Mohammadi. Practical applications to fluid flows of automatic differentiation for design problems. *VKI lecture*, 1997.

[38] B. Mohammadi and O. Pironneau. *Analysis of the k-epsilon turbulence model.* RAM: Research in Applied Mathematics. Masson, Paris, 1994.

[39] B. Mohammadi and O. Pironneau. New tools for optimum shape design. *CFD Review, Special Issue*, 1995.

[40] J. M. Ortega and E. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, 1970.

[41] E.R. Panier, A.L. Tits, and J.N. Herskovits. A qp-free globally convergent, locally superlinearly convergent algorithm for inequality constrained optimization. *SIAM J. Control Optim.*, 26:788–811, 1988.

[42] O. Pironneau. *Optimal shape design for elliptic systems.* Springer Verlag, 1983.

[43] O. Pironneau. *Finite element methods for fluids.* John Wiley & Sons Ltd., Chichester, 1989. Translated from the French.

[44] F.A. Potra. An $o(nl)$ infeasible-interior-point algorithm for lcp with quadratic convergence. *Annals of Operations research*, 62:81–102, 1996.

[45] M. J. D. Powell. *Variable Metric Methods for Constrained Optimization — The State of the Art.* Springer-Verlag, 1983.

[46] G. Kuruvila S. Ta'asan. Aerodynamic design and optimization in one-shot. *AIAA paper 92-0025*, 1992.

[47] Y. Saad and M. Schultz. A generalised minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[48] G. Strang. *Introduction to Applied Mathematics.* Wellesley-Cambridge Press, Wellesley, Massachussets, 1986.

[49] K. Svanberg. The method of moving asymptotes: a new method for structural optimisation. *Int. J. Num. Meth. Eng.*, 24:359–373, 1987.

[50] A. Tits and J. L. Zhou. *Large Scale Optimization. State of the Art. - A simple Quadratically Convergent Interior Point Algorithm for Linear Programming and Convex Quadratic Programming.* Kluwer Academic Publishers, 1993.

[51] P. Wolfe. Convergence conditions for ascent methods. *SIAM Review*, pages 226–235, 1969.

[52] O. Zienkiewicz. *The Finite Element Method in Engineering Science.* McGraw-Hill, 1977.

Index