INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

REVERSE AUTOMATIC DIFFERENTIATION FOR OPTIMUM DESIGN: FROM ADJOINT STATE ASSEMBLY TO GRADIENT COMPUTATION

F. Courty — A. Dervieux — B. Koobus — L. Hascoet

N° ????

Janvier 2002

THÈME 1

ISSN 0249-6399

REVERSE AUTOMATIC DIFFERENTIATION FOR OPTIMUM DESIGN: FROM ADJOINT STATE ASSEMBLY TO GRADIENT COMPUTATION

F. Courty , A. Dervieux , B. Koobus , L. Hascoet

Thème 1 — Réseaux et systèmes
Projet Tropics

Rapport de recherche n° ???? — Janvier 2002 — 33 pages

Abstract: The utilization of reverse mode Automatic Differentiation to the adjoint method for solving an Optimal Design problem is described. Using the reverse mode, we obtain the adjoint system residual in a rather efficient way. But memory requirements may be very large. The family of programs to differentiate involves many independant calculations, typically in parallel loops. Then we propose to apply a reverse differentiation "by iteration". This demands much less memory storage. This methods is used for the computing of the adjoint state and gradient related to the Optimal Design problem.

Key-words: automatic differentiation, optimal shape design, computational fluid dynamics, Navier-Stokes, compressible flow, adjoint, gradient

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

APPLICATION DE LA DIFFERENTIATION AUTOMATIQUE EN MODE INVERSE A UN PROBLEME D'OPTIMISATION DE FORMES: DE L'ASSEMBLAGE DE L'ETAT ADJOINT AU CALCUL DU GRADIENT

Résumé : Nous décrivons l'utilisation de la Différentiation Automatique en mode inverse, pour résoudre le système adjoint d'un problème d'Optimum Design. Le mode inverse permet d'obtenir le résidu du système adjoint de manière assez efficace. Il demeure cependant très coûteux en espace mémoire. Nous exploitons le fait que notre programme initial présente de nombreux calculs indépendants et en particulier des boucles parallélisables. Dans ce cas, nous proposons une différentiation inverse "par itération" qui consomme beaucoup moins de mémoire. Cette méthode est utilisée pour calculer efficacement un état adjoint et un gradient pour le problème d'Optimal Design.

Mots-clés : différentiation automatique, optimisation de formes, mécanique des fluides numérique, Navier-Stokes, écoulement compressibles, adjoint, gradient

1 Introduction

Adjoint systems are getting more and more popular among engineers and researchers working in optimal design of systems modelized by Partial Differential Equations (PDE).

The adjoint equation comes from the Optimal Control theory and expresses the fact that only one direction of the sensitivity of the state variable is needed to find a descent direction of the scalar cost functional, viz. the product of the state derivative by the derivative of the cost functional. Computing the complete sensitivity of the state variable with respect to the control, by the implicit function theorem, leads to solve as many linearised systems as the dimension of the control vector. With the adjoint approach, we need only to solve one linearised transposed system.

Since in practice the PDE is discretised and solved on a computer, there are two natural ways to build the adjoint:
(1) to derive the adjoint PDE and then to discretise it, or,
(2) to find the adjoint of the discretised state system.

It is likely that (2) be a little more complex than (1) when it is undertaken solely "by hand". But option (2) is more easy to validate (by comparison with divided differences for example) and provides an exact evaluation of the gradient, and therefore a fiable descent direction to the optimisation kernel.
Further, for both options, the development in engineer time may be enormous. Then the hope of an automatic differentiation in option (2) is of paramount importance.

Today's question is still to evaluate and improve the conditions in which an Automated Differentiation software answers to the issue of sensitivity software generation.

Mainly two modes of Automatic Differentiation are available today, namely the *forward* and the *reverse mode*.

The forward mode computes the product of the Jacobian of a program times a given direction vector, $\dot{y} = f'(x)\dot{x}$. Applying the forward mode leads to compute the sensitivity of the state variable. If the control has a dimension n, this sensitivity must be evaluated n times to get the whole Jacobian. This standpoint is much more accurate and robust than using divided differences, but not very efficient, and, in fact, for a rather large number n, not applicable to large scale models as those arising in aerodynamics when Navier-Stokes models are chosen.

The reverse mode computes the product of the transposed Jacobian of a program times a given vector, $f'^t(x).\overline{y}$. This returns a linear combination of the lines of the Jacobian, and therefore it is the natural technique to compute the adjoint.

Mohammadi ([15]) made a first application of the reverse mode to this problem. His strategy relies on the reverse differentiation of the complete simulation software suite, including flow assembly and solver, and objective functional evaluation. As a result of this differentiation, the adjoint system is introduced in a transparent way. But at the same time,

many extra problems arise such as the increased complexity of the solution algorithm and the massive use of storage. Both problems must be solved by tedious and delicate hand manipulations.

Another strategy is to apply reverse differentiation to the two smaller parts of the software that assemble (1) the state equations and (2) the cost functional. It assembles only the adjoint system residual, and then the user must use this residual in a piece of program that solves the adjoint system.

This is the standpoint of the present work. We propose a description of how the different terms of the adjoint equation and of the functional derivative can be obtained by applying the reverse mode using the ODYSSEE Automatic Differentiation tool.

We emphasize that the differentiation tool ODYSSEE will help to develop the routines assembling the residual of the adjoint state system but not to build the routine solving the adjoint system. Then it remains to apply a "matrix free" solver in order to iterate to a fixed point, making the residual vanish.
We thus propose a class of solution algorithms using a simplified Jacobian. This simplified Jacobian often exists in industrial software. The accuracy and efficiency of this approach is briefly analysed.

In Sec.2 we introduce the shape optimisation model problem. In Sec.3 we present our optimized method for applying the reverse mode to the differentiation of parallel loops. Sec.4 concentrates on the application of this optimized method to the adjoint assembly of the optimal problem of Sec.2. This is done without any matrix storage. Sec.5 proposes the solution algorithm for solving the adjoint system and completing the process to the objective functional gradient evaluation. Sec.6 presents a few numerical illustrations.

2 A shape optimization problem

This section is devoted to the description of the different steps involved for solving a shape optimisation problem. First we introduce the continuous shape optimisation problem. Then we introduce the discrete standpoint and finally we describe a gradient method with adjoint.

2.1 A model problem

We start with a model problem. It consists in solving an shape inverse/optimization problem for a family of 2D nozzles in which the flow is modelled by the Euler equations. By inverse/optimization we mean that the formulation is that of an optimization problem, but the precise description will be, for assessment purpose, that of an inverse problem. The flow model is moderately complex and the mesh is variable. The geometry is depicted in Fig.1, inlet and outlet are of constant section, and the median part can vary through the

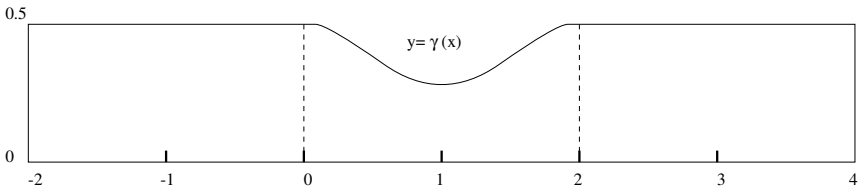


Figure 1: The nozzle model problem

ordinates of the upper boundary. Variations around this formulation have been studied in an European BRITE-ECARP project ([1][14]).

According to the formalism of optimal control theory, the control variable is a function γ of the abscissa which defines the domain shape (Fig.1), and to which corresponds a value of the objective or cost function j to be minimised with respect to γ. More precisely, the set of admissible controls is a subset Γ_{ad} of $\mathcal{C}^1([0,1], I\!R^2)$. To each γ in Γ_{ad} corresponds a smooth geometrical domain Ω_γ of the plan $I\!R^2$, limited by the $y = \gamma(x)$ curve (see Fig. 1) and the boundary of which is also denoted by γ.

Let us introduce:

$$\begin{array}{rcl} J : \Gamma_{ad} \times H^1(I\!R^2, I\!R^4) & \longrightarrow & I\!R \\ (\gamma, W) & \longmapsto & J(\gamma, W) = \displaystyle\int_\gamma (P(W) - P^{target})^2 d\sigma \end{array} \tag{1}$$

where $P(W)$ is the pressure of W, (that is $0.4(W_4 - 0.5(W_2^2 + W_3^2)/W_1)$), W being a function of the independent variables x and y, and P^{target} a prescribed pressure generally computed from a prescribed shape.

The optimization problem to be solved is written:

$$\text{Find } \gamma_0 \text{ in } \Gamma_{ad} \text{ such that } j(\gamma_0) = \min_{\gamma \in \Gamma_{ad}} j(\gamma) \tag{2}$$

with

$$j(\gamma) = J(\gamma, W(\gamma)) \tag{3}$$

and where $W(\gamma)$ is the solution of the state equation related to the control γ as follows:

$$\Psi(\gamma, W(\gamma)) \quad = \quad 0 \tag{4}$$

where

$$\Psi(\gamma, W) \quad = \quad \frac{\partial F(W)}{\partial x} + \frac{\partial G(W)}{\partial y} + \text{Boundary conditions} \tag{5}$$

in which:

$$W = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ e \end{pmatrix}, \quad F(W) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho u \frac{e+p}{\rho} \end{pmatrix}, \quad G(W) = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho v \frac{e+p}{\rho} \end{pmatrix} \tag{6}$$

where ρ is the density, u et v the velocity components following x and y, e the total energy, p the pressure given by:

$$p \;=\; P(W) \;\;=\;\; (\kappa - 1)[e - \frac{1}{2}\rho(u^2 + v^2)] \tag{7}$$

where κ is 1.4.

2.2 Discrete standpoint

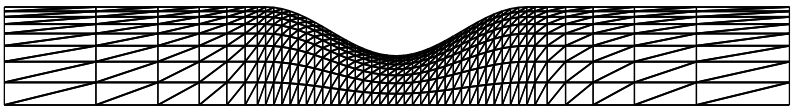


Figure 2: Nozzle mesh with 423 vertices

The mesh is made of triangles. It starts from an orthogonal triangulation with coordinates (x^1_{ij}, y^1_{ij}) of the constant-section nozzle geometry i.e. defined by $\gamma = 1$.
For any given control $(\gamma_i, i = 1, m)$, we derive a mesh on a accordion-like mode (Fig.2), that is by sliding vertices along vertical lines:

$$\forall i \in \{1, \ldots, m\},\ y^\gamma_{im'} = \gamma_i\ ;\ \ \forall j \in \{1, \ldots, m'\},\ y^\gamma_{ij} = \theta_j\ y^\gamma_{im'}\ +\ (1 - \theta_j)\ y^\gamma_{i1} \tag{8}$$

here m is the number of points in x direction in the interval $[0, 2]$, m' is the number of points in the y direction, and θ_j is a non-variable parameter specifying the vertical node distribution.
In the sequel, we represent by the vector X of dimension M $(M = mm')$ the coordinates of the mesh. The mesh coordinates X are thus defined by (8) as an explicit function of the control parameter γ.

To discretize the flux balance Ψ, we apply a vertex centered upwind finite-volume formulation (relying on median dual cells).

We denote by $V(j)$ the set of nodes (=vertices) that are neighbors of j, by ∂C_j the cell boundary associated with node j, by $\partial C_{jk} = \partial C_j \bigcap \partial C_k$,by $\vec{n}$ the outward pointing normal

vector to ∂C_j and by $\overrightarrow{\eta}_{jk} = \int_{\partial C_{jk}} \overrightarrow{n} d\sigma$.

We obtain the finite-volume formulation:

$$\Psi(X, W^n)_j \quad = \quad \sum_{k \in V(j)} \Phi(W^n_{jk}, W^n_{kj}, \overrightarrow{\eta}_{jk}(X)) + \text{ boundary conditions} \tag{9}$$

The function Ψ depends on state variable W through the two first variables W^n_{jk} and W^n_{kj} of the elementary inter-cell flux Φ and depends on the mesh coordinate X through the integrated normals $\overrightarrow{\eta}_{jk}(X)$. The W^n_{jk} and W^n_{kj} are linear extrapolations from cells j and k according to van Leer "MUSCL" ideas, extended to triangulations as in [4]. TVD limiters of van Albada type can be applied or not.

The upwinding in elementary flux Φ is the van Leer flux splitting (which is differentiable). We can then obtain the state solution iteratively from a linearised implicit time advancing:

$$\left(\frac{M}{\Delta t^n} + \Psi'(W^n) \right) \delta W^{n+1} \quad = \quad -\Psi(W^n) \tag{10}$$

M is the diagonal mass matrix, and when the time step Δt^n grows infinitely, Algorithm (10) would become a Newton iteration for solving $\Psi(W) = 0$.

The genuine Jacobian $\Psi'(W^n)$ is a high storage matrix and is not used in the computer code. Instead, we use the simplified "spatially first-order" Jacobian matrix A_1 of the state equation which results from the exact differentiation of the first-order accurate flux balance which uses constant cell values instead of linear interpolated ones:

$$\Psi_1(X, W^n)_j \quad = \quad \sum_{k \in V(j)} \Phi(W^n_j, W^n_k, \overrightarrow{\eta}_{jk}(X)) + \text{ boundary conditions.} \tag{11}$$

The linearised implicit time advancing then writes:

$$A_1 = (\Psi_1)'(W^n) \tag{12}$$

$$\bar{W}^{n+1} \; = \; W^n - \left(\frac{M}{\Delta t^n} + A_1 \right)^{-1} \Psi_2(\gamma, W^n). \tag{13}$$

We refer to [4] for a recent detailed description of the first-order Jacobian A_1. Operator A_1 acts as a preconditioner. Thanks to its quasi-diagonal dominance properties, the preconditioning system:

$$\left(\frac{M}{\Delta t^n} + A_1 \right) \; \delta w = RHS \tag{14}$$

is easily solved with complete or partial convergence with a Jacobi iteration (in [4], we apply a multi-grid iteration). The preconditioner relies on a direct neighbor approximation and the storage of its non-vanishing entries is affordable. This results in a solution strategy using

a reasonable memory storage and much more efficient that an explicit one.
This completes the description of the flow solver.

We consider now a *discrete optimization problem* with m real parameters:

$$\begin{array}{rcl} j : I\!R^m & \longrightarrow & I\!R \\ \gamma & \longmapsto & j(\gamma) \end{array}$$

$$\begin{array}{rcl} J : I\!R^m \times I\!R^N & \longrightarrow & I\!R \\ (\gamma, w) & \longmapsto & J(\gamma, w) \end{array}$$

where the dimension N of the state variable is four times the number of nodes (=vertices) $N = 4 \times mm'$.

The discrete variant of (2) is written:

$$\text{Find } \gamma_0 \text{ in a subset } \Gamma_{ad} \text{ of } I\!R^m \text{ such that } j(\gamma_0) = \min_{\gamma \in \Gamma_{ad}} j(\gamma). \tag{15}$$

with again:

$$j(\gamma) = J(\gamma, W(\gamma)) \tag{16}$$

where $W(\gamma)$ is the solution of the state equation:

$$\Psi(\gamma, W(\gamma)) = 0 \tag{17}$$

We observe that we can apply a chain rule for the differentiation of j, which leads to introduce an adjoint state, and allows to write the *optimality system* as follows:

$$\left\{ \begin{array}{rcl} \Psi(\gamma, W) & = & 0 \\ & & \text{(state)} \\ (\dfrac{\partial \Psi}{\partial W}(\gamma, W))^t \ \Pi & = & \dfrac{\partial J}{\partial W}(\gamma, W) \\ & & \text{(adjoint state)} \\ j'(\gamma) = \dfrac{\partial J}{\partial \gamma}(\gamma, W) - < \Pi, \dfrac{\partial \Psi}{\partial \gamma}(\gamma, W) > & = & 0 \\ & & \text{(optimality)} \end{array} \right. \tag{18}$$

In this system, we highlighted four expressions because they will be evaluated through Automatic Differentiation, as explained in Sec. 4

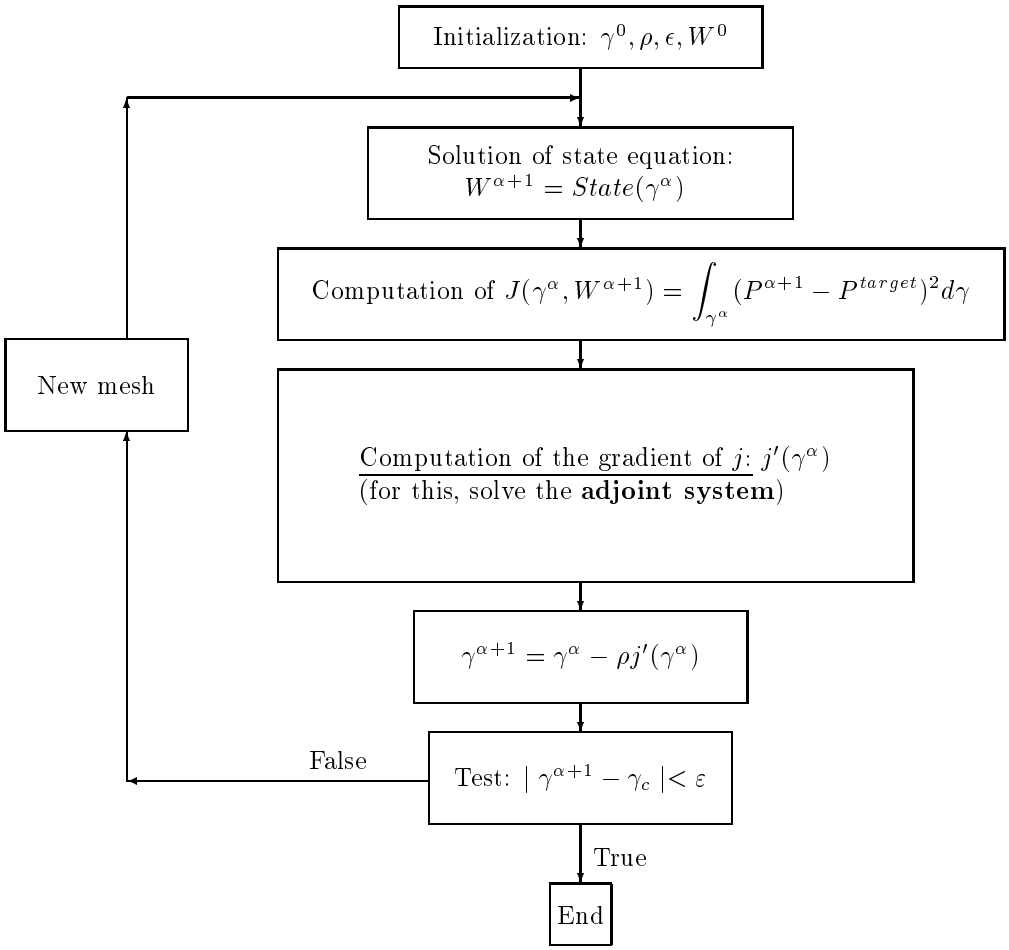


Figure 3: Description of the "adjoint algorithm"

2.3 Optimization with gradient

In Fig.3 is depicted the constant step gradient method with adjoint, as a reference algorithm for solving the optimization problem (15). The rest of the paper examines the way in developing the software computing the gradient j'.

3 An improved Reverse Mode of Automatic Differentiation

Automatic or Algorithmic Differentiation (A.D.) is getting more and more widely used in optimization problems. We refer the interested reader to the latest collection of articles [**?**] and to the recent monography [5].

A.D. permits differentiation of *programs*. By that, we mean that an A.D. tool takes as input a source program (typically in FORTRAN) that, given an argument $x \in I\!R^m$, computes some function $y = f(x) \in I\!R^n$. The A.D. tool generates a new source (FORTRAN) program that, given the argument x, computes some derivatives of f. There are several *modes* of A.D., according to which derivatives are computed. In the following, we will first present the *forward mode of A.D., on which we will show the fundamentals of A.D. Then we will present the more complex* reverse mode, which is what we need to compute adjoints. Finally, we will show how this reverse mode can be vastly improved in a specific case, and how this improvement can be automated.

3.1 The Forward mode of Automatic Differentiation

The *forward mode* of A.D. produces a *directional derivative*, i.e. a program that, given an argument $x \in I\!R^m$ and a direction vector $\dot{x} \in I\!R^m$, computes the directional derivative $\dot{y} = f'(x)\dot{x}$. This mode is sometimes referred to as the *direct mode*.

Fundamentally, A.D. identifies computer programs with a composition of mathematical functions. Precisely, any program P composed of a sequence of instructions $I_k, i \in [1..p]$ and that implements a function f, is such that

$$f = f_p \circ f_{p-1} \circ \ldots \circ f_1$$

where each f_k is the elementary function implemented by instruction I_k.

From the chain rule, and writing f' for the derivative (Jacobian matrix) of f, we get:

$$\begin{array}{rcl} f'(x) & = & (f'_p \circ f_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x)) \\ & & . (f'_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x)) \\ & & . \ \ldots \\ & & . (f'_1(x)) \ . \end{array} \tag{19}$$

To compute $\dot{y} = f'(x)\dot{x}$, the forward mode needs to compute the matrix×vector product $\dot{x}_1 = (f'_1(x)).\dot{x}$, then compute $\dot{x}_2 = (f'_2 \circ f_1(x)).\dot{x}_1$, and so on until

$$\dot{y} = \dot{x}_p = (f'_p \circ f_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x)).\dot{x}_{p-1} \ .$$

If we observe that x is the original argument, then $f_1(x)$ is the intermediate value after the first instruction of P, and more generally that $f_k \circ \ldots \circ f_1(x)$ is the intermediate value after execution of I_k, we see that the above process requires the intermediate values of the

execution of P in the *same order* in which they are computed by P. Therefore this process is straightforward to implement, by interleaving it with the original execution of P. Thus, the forward differentiation of P is a program $\dot{P}$, obtained from P by inserting before each instruction I_k a derivative instruction $\dot{I}_k$ that implements f'_p.

The forward mode computes a linear combination of the *columns* of the Jacobian matrix $f'(x)$. On the other hand, the adjoint that we need here is a linear combination of the *lines* of $f'(x)$. To compute this adjoint with the forward mode, we thus need to compute the m columns of $f'(x)$, which requires grossly m executions of the forward differentiated program. This can be quite expensive for large values of m. The *reverse mode* is a way to solve that.

3.2 The Reverse mode of Automatic Differentiation

The *reverse mode* of A.D. produces a *gradient*, i.e. a program that, given an argument $x \in I\!R^m$ and a line-vector $\overline{y}^t \in I\!R^n$, computes the product $\overline{y}^t.f'(x)$, which is also equal to the transposed of $f'^t(x).\overline{y}$. One can think of $\overline{y}$ as a weighting vector on y, the results of f, that defines a scalar composite result, of which we compute the gradient. This mode is sometimes referred to as the *adjoint mode*.

From Equation (19) and after transposition we obtain:

$$\begin{array}{rcl} f'^t(x).\overline{y} & = & (f_1'^t(x)) \\ & & .\ (f_2'^t \circ f_1(x)) \\ & & .\ \ldots \\ & & .\ (f_p'^t \circ f_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x)) \\ & & .\ \overline{y}\ . \end{array} \tag{20}$$

To compute $f'^t(x).\overline{y}$, and because matrix×vector products are so much cheaper than matrix×matrix products, the reverse mode starts to compute

$$\overline{y}_p = (f_p'^t \circ f_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x)).\overline{y}$$

then it computes

$$\overline{y}_{p-1} = (f_{p-1}'^t \circ f_{p-2} \circ \ldots \circ f_1(x)).\overline{y}_p$$

and so on until $f'^t(x).\overline{y} = \overline{y}_1 = (f_1'^t(x)).\overline{y}_2$.

This time, we observe that this process requires the intermediate values of the execution of P in the *inverse order* of their computation by P. One way to handle this is to recompute each intermediate value when required. This implies repeated computations of each f_k function, which is expensive in computation. Conversely, another way is to store all the intermediate results as they are computed, so as to retrieve them when required by the reverse instructions. This is expensive in memory space. There is an optimal tradeoff to be found between these two ways.

The reverse mode of the A.D. tool ODYSSÉE implements one such tradeoff. The reverse differentiated program runs recursively on its call tree. Each subroutine is available both in

its differentiated form and its original form. The differentiated execution is achieved by calling the main routine in its differentiated form. In its differentiated form, each differentiated routine starts by executing the original instructions, with additional storage instructions that save intermediate variables. During this so-called *forward sweep*, each call to a subroutine calls its non-differentiated version. This forward sweep is followed by the *reverse sweep*, that computes the products with each instruction's Jacobian matrix, in the reverse order. During the reverse sweep, the intermediate values stored beforehand are used. When the reverse sweep reaches a call to a subroutine, it restores the state of variables *just before* this subroutine was called, and it calls it again, this time in its differentiated form. Fig.4 shows

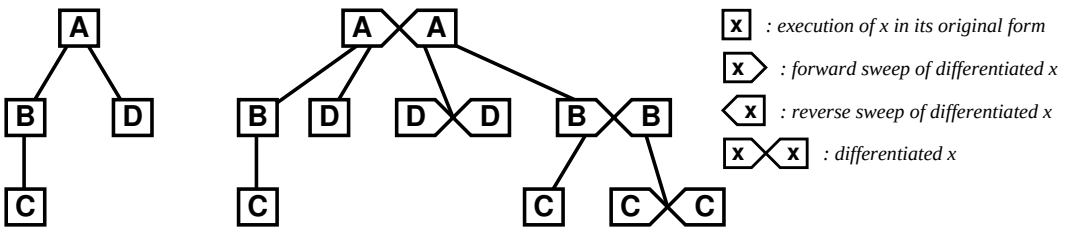


Figure 4: *Reverse mode A.D on a program Call Tree, showing storage/recomputation tradeoff*

the effect of this tactique on a small call tree. One can easily check that the amount of storage is grossly proportional to the depth of the call tree, and the amount of recomputation is grossly proportional to the square of this depth.

Despite of these problems of storage/recomputation tradeoffs, the reverse mode of A.D. has the advantage of returning in one call a linear combination of the lines of the Jacobian $f'(x)$. Therefore when $m > n$, the reverse mode is probably worth considering. Let us take a look again at our particular problem, summarized by Equations (18). Since the functional J is scalar-valued, the highlighted partial derivatives of J in the equations appear to be excellent candidates for the reverse mode (m is large, and $n = 1$). The highlighted products of transposes of Jacobians of Ψ times vectors are also good candidates, by definition of the reverse mode above.

To summarize, our approach is to apply A.D. in the reverse mode to compute the four highlighted expressions is Equations (18). We are now going to describe an improvement to the reverse mode, that reduces drastically the amount of memory necessary for storage of intermediate values.

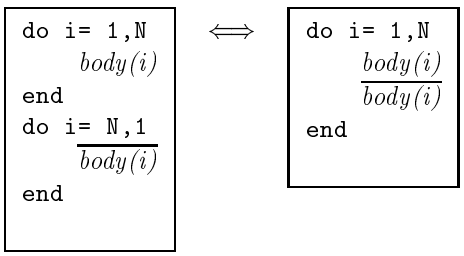
3.3 Adjoining Independant Computations

The main subroutine that we must differentiate in the reverse mode is the routine that computes Ψ. Using scientific programming vocabulary, this routine consists of several successive assembly loops on mesh entities. Application of the reverse mode to those long routines

leads to an extremely large consumption of memory space to store intermediate values. On large meshes, this may become more than what the underlying system can hold.

We remark that these assembly loops have independent iterations. In other words, they could be run in parallel. For these two reasons, let us call them *II*-loops. There are several specialized techniques to improve the memory consumption of the reverse mode for loops. Some are described by Griewank [5], another one was proposed by Hovland, Mohammadi, and Bischof [9]. We are going to use a new technique, specifically devised for *II*-loops, and introduced in [6], [8].

This technique relies on the fact that, for any *II*-loop, the standard reverse differentiation of the loop is equivalent to a more compact form, as shown on the following figure:



The notation $\overline{body}$ represents the reverse sweep corresponding to *body*. One can see on the figure that the forward sweep of each iteration *body(i)* is immediately followed by its reverse sweep $\overline{body(i)}$, and therefore the intermediate values stored are used immediately after. As a consequence, this manipulation reduces drastically the amount of memory used (by a factor N, size of the loop).

This transformation can be applied by hand on each *II*-loop in the reverse differentiated program, and this reduces memory consumption as foreseen. However, we would prefer to mechanize this tedious transformation. In its present form, Odyssée does not provide us with this transformation. This development is planned in the next release, but is not available now. In the meantime, we propose a partial mechanization of the process, based on the handling of subroutine calls in Odyssée (*cf* Fig.4). This combines application of the existing Odyssée tool with *hand manipulations*. Our goal is to minimize the work done by hand, and leave as much as possible to the automatic tool. This also minimizes the risk of manipulation errors.

3.4 Step-by-step checklist for Adjoining Independant Computations

The following checklist relies on the use of the Odyssée A.D. tool. However, it transposes easily to other A.D. tools.

Step 1: split parallel loops into subroutines.
Let us call `WFLO` the routine to differentiate, sketched on the next figure. It contains a number of *II*-loops. one of which (the third one...) is shown.

(Initial) $\Longrightarrow$

```
Routine WFLO(...)
...
Do i = 1,N
  body(i)
EndDo
...
End
```

The first step is a *manual preparation of the original program*, to extract *II*-loops and the bodies of *II*-loops into separate subroutines. Precisely, each *II*-loop must be replaced by a call to a new subroutine (e.g. `CALL WFLOLOOPn(...)`) with appropriate call arguments. The new subroutine must be defined, and its only contents is a loop, whose body is reduced to a call to another new subroutine (e.g. `CALL WFLOBODYn(...)`). Finally, this second subroutine must be defined, with the appropriate arguments, and its body is exactly the body of the original *II*-loop. This first step leads to the program illustrated by the following figure.

(by hand) $\Longrightarrow$

```
Routine WFLO(...)
...
call WFLOLOOP3(...)
...
End

Routine WFLOLOOP3(...)
Do i = 1,N
  call WFLOBODY3(i,...)
EndDo
End

Routine WFLOBODY3(...)
body(i)
End
```

Step 2: Run Odyssée, reverse mode, on the resulting program.
The second step is automatic. In the A.D. tool Odyssée, one must load the modified top routine to differentiate `WFLO`, along with all subroutines recursively called by it. In

particular, all the `WFLOLOOPn` and `WFLOBODYn` must be loaded, but also all subroutines called under them.

Recall that an A.D. tool performs a dependence analysis that determines, for each variables, whether it actually depends on the input differentiation variables. If a dependency occurs, then a variable must be defined to hold the derivative. When some called subroutine is not loaded into the A.D. tool, the tool makes the conservative assumption that each of its outputs may depend on each of its inputs. This leads to unwanted dependencies, and finally to unnecessary derivative variables and high memory consumption.

Then just call A.D. in the reverse mode, giving the name of the top routine to differentiate (here `WFLO`), and the name of the input differentiation variables (here `XFLO`).

Inside ODYSSÉE, this is simply achieved by the two following lines of commands:

```
   load wflo.f wfloloop1.f wflobody1.f ....
diff -cl -head wflo -vars XFLO -output wflocl.f
```

which create a new file `wflocl.f` that contains the differentiated subroutines.

The differentiated subroutines, called `WFLOCL`, `WFLOLOOPnCL` and `WFLOBODYnCL`, are sketched in the next figure. They obey ODYSSÉE's tactique for storage/recomputation tradeoff in the reverse mode, as shown on Fig.4. Each `*CL` subroutine consists of a forward sweep followed by a reverse sweep. Notice that this tactique implies calls to the original versions of the subroutines `WFLOLOOPn` and `WFLOBODYn`, which must therefore be linked to the differentiated code.

(ODYSS) $\Longrightarrow$

```
Routine WFLOCL(...)
...
call WFLOLOOP3(...)
...
call WFLOLOOP3CL(...)
...
End

Routine WFLOLOOP3CL(...)
Do i = 1,N
  Store(i)
  call WFLOBODY3(i,...)
EndDo
Do i = N,1,-1
  Restore(i)
  call WFLOBODY3CL(i,...)
EndDo
End

Routine WFLOBODY3CL(...)
  body(i)
  body-bar(i)
End
```

Step 3: enforce the efficient differentiation of *II*-loops

The third step is again a manual transformation, that leads to the improved, more efficient, differentiation of *II*-loops. It is advisable to perform these transformations on a copy of the generated file `wflocl.f`, say `wflocl_opt.f`.

The essence of the transformation is that each iteration of the *II*-loop must be followed immediately by its reverse sweep. This amounts to first fuse the two loops in `WFLOLOOP3CL`, into a single one. Then the call to the original routine `WFLOBODY3` is removed, along with the associated *Store* and *Restore* calls, and the arrays that were allocated for this store/restore mechanism. This leads to the program sketched below, which is what we wanted to obtain:

(by hand) $\Longrightarrow$

```
Routine WFLOCL(...)
...
call WFLOLOOP3(...)
...
call WFLOLOOP3CL(...)
...
End

Routine WFLOLOOP3CL(...)
Do i = 1,N









  call WFLOBODY3CL(i,...)
EndDo
End

Routine WFLOBODY3CL(...)
  body(i)
  body(i)
End
```

On some systems, it might be profitable to perform an extra transformation, namely to inline subroutine `WFLOBODY3CL` back into `WFLOLOOP3CL`. This may save the overhead of several subroutine calls. This did not prove profitable on our system, and therefore we do not show it here.

Step 4: clean the resulting differentiated program

There are some well-known problems and limitations with the current version of ODYSSÉE, that require a final cleaning step. Notice that these problems may not show up with other A.D. tools, and will be removed from the next version of ODYSSÉE. Let us just list them here:

- It often happens that ODYSSÉE creates new arrays, for the *Store* and *Restore* operations, dimensioned after arrays of the original routine. Sometimes it is impossible to guess the size of these arrays statically. These sizes are then denoted by variables named `ody***`, and the user is requested for the definition of these, through a special include file named named `odyparam.inc`. If this is not done, the compiler will complain.

- Due to a bug in the current version of ODYSSÉE, some uses of `PARAMETER` values may occur before their definition. This results in a message from the compiler.
- As we said earlier, there may be some unnecessary storage. Some values may be stored and retrieved, while they are not used in the differentiated instructions. The next version of ODYSSÉE tries to reduce the number of unecessary storage, but some will remain that can only be detected and removes by the end-user.
- Values of variables *before* initialization should not be saved/restored.

4 Low-storage gradient assembly with AD

By low-storage we mean that Jacobians are never computed explicitly, but instead their product with a vector, yielding results that are vectors of yet tractable storage. We show now that it is possible for two terms arising in the optimality conditions (18).

4.1 Assembly of the gradients of functional J

We consider the building of the highlighted derivatives in (18):

$$\left\{ \begin{array}{rcl} DJ_W & = & \dfrac{\partial J}{\partial W}(\gamma, W) \\ DJ_\gamma & = & \dfrac{\partial J}{\partial \gamma}(\gamma, W) \end{array} \right. \tag{21}$$

We assume that the dependency with respect to design parameters is accounted through the mesh coordinates X and we consider the differentiation with respect to X and to W.

For this, we shall differentiate the cost subroutine (`JCOST`) whose input are the mesh coordinates X_M (`XMESH(1:M)`) and the flow variables W_N (`WFLO(1:N)`), and whose output is the scalar cost (`CVALUE`).

Therefore the variable for differentiation is `XMESH(1:M)`, the result to differentiate is `CVALUE`.

We shall differentiate the cost subroutine (`JCOST`) with respect to the mesh coordinates (`XMESH(1:M)`). The application of ODYSSEE in *reverse or cotangent linear mode*, option `-cl`, to this context by:

```
diff -cl -h JCOST -vars XMESH
```

will generate a differentiated subroutine named (`JCOSTCL`) with the following extra inputs and outputs:

- an extra scalar input (`CVALUECL`),

- an extra array output, `XMESHCL(1:M)`, which will contain the gradient of J with respect to X.

We shall then differentiate the cost subroutine (`JCOST`) with respect to the flow variables (`WFLO(1:N)`). The application of ODYSSEE in *reverse or cotangent linear mode*, option `-cl`, to the *same* context by:

```
diff -cl -h JCOST -vars WFLO
```

will generate another subroutine, also named (`JCOSTCL`).

The new routine has the same inputs and outputs as (`SUBROUTINE JCOST`) except:

- an extra scalar input, `CVALUECL`,

- an extra array output, `WFLOCL(1:N)`, which will, contain the gradient of J with respect to W.

4.2 Assembly of the Adjoint equation residual

We assume that this residual evaluation will be enough for solving the adjoint system (i.e. solving it by a matrix-free algorithm, such as GMRES, for instance).

The state equation is written as the vanishing of its residual (Euler equation discretisation, including boundary conditions); we precise the matrices dimension for clarity:

$$(\Psi(X_M, W_N))_N \;=\; 0. \tag{22}$$

In the above function, the influence of the design parameter γ_m is applied through the mesh coordinates X_M.
The adjoint equation residual is then written as:

$$S_N \;=\; (\;(\frac{\partial \Psi}{\partial W}(X,W))^t_{N\times N}\;.\;\Pi_N\;)_N \;\;-\;\;(\frac{\partial J}{\partial W}(\gamma,W))_N\;. \tag{23}$$

We suggest to apply the reverse differentiation mode of ODYSSEE to the following *program transformation context*:

- routine to differentiate : let us call it "`SUBROUTINE FLUX`"; the input data are the mesh coordinates X_M (`XMESH(1:M)`), and flow variables W_N (`WFLO(1:N)`), and the output result is the state equation residual $\Psi(X,W)_N$ (`PSIFLU(1:N)`),

- active variable for differentiation : `WFLO(1:N)`,

- result to differentiate : `PSIFLU(1:N)`.

The application of ODYSSEE to this context

```
diff -cl -h flux -vars wflo
```

will generate a new subroutine named (`SUBROUTINE FLUXCL`) with same inputs and outputs except:
- an extra array input, `PSIFLUCL(1:N)`, which should be set at running time equal to Π_N, that will be the successive iterates of the adjoint state solution,

- an extra array output, `WFLOCL(1:N)`, which will, at running time contain the following computational result:

$$(\frac{\partial \Psi}{\partial W}(X,W))^t_{N\times N} \ . \ \Pi_N \ . \tag{24}$$

Therefore, the adequate call of `subroutine fluxcl` will return the desired result.

4.3 Assembly of the sensitivity of state residual

Keeping the notation of the previous section, we consider now the computation of:

$$T_m \ = \ (\frac{\partial \Psi}{\partial \gamma}(\gamma,W))^t_{m\times N} \ . \ \Pi_N \tag{25}$$

where Π is assumed to be given. In practice, we shall first differentiate with respect to the mesh coordinates X:

$$T_m \ = \ ((\frac{\partial \Psi}{\partial X}(X,W))_{N\times M} \ . \ (\frac{\partial X}{\partial \gamma}(\gamma))_{M\times m})^t \ . \ \Pi_N. \tag{26}$$

It is then possible to introduce the following transposition:

$$T_m \ = \ (\frac{\partial X}{\partial \gamma}(\gamma))^t_{m\times M} \ . \ (\ (\frac{\partial \Psi}{\partial X}(X,W))^t_{M\times N} \ . \ \Pi_N \ )_M \ . \tag{27}$$

Let us then get convinced that this expression can be directly obtained by *reverse* differentiation with ODYSSEE. Indeed, a way to understand this is that this tool, applied to `subroutine flux`, will formally produce the gradient of the functional:

$$F: \ X \ \longmapsto \ \Psi(X,W)^t_N.\Pi_N \ . \tag{28}$$

More practically, ODYSSEE will be applied to the following *program transformation context*:

- routine to differentiate : again the subroutine named (`FLUX`); with as input the mesh coordinates X_M, `XMESH(1:M)`, and the flow variables W_N, in `WFLO(1:N)`, and as output the state equation residual $\Psi(X,W)_N$, `PSIFLU(1:N)`,

- active variable for differentiation :`XMESH(1:M)`,

- result to differentiate : `PSIFLU(1:N)`.

The application of ODYSSEE to this context will generate a new subroutine named (`FLUXCL`), different from the one built in the previous section (*also saved beforehand with another name*). It has same the same inputs and outputs as the subroutine (`FLUX`) except:

- an extra array input, `PSIFLUCL(1:N)`, which should be set at running time equal to Π_N; this time , Π_N is the (converged) result of solving the adjoint state equation,

- an extra array output, `XMESHCL(1:M)`, which will, at running time contain the following computational result (of dimension M):

$$(\frac{\partial \Psi}{\partial X}(X,W))^t_{M\times N} \ . \ \Pi_N \ . \tag{29}$$

It remains to multiply by $(\frac{\partial X}{\partial \gamma}(\gamma)))^t_{m\times M}$.

4.4 Large arrays

In this section, we give precisions about the arrays that will contain the main intermediate results between the different phases of the assembly of the gradient.

It should be emphasized that intermediate large arrays such as:
- transformation of flow variables (from conservative to primitive,...),
- storage of metrics (normal vector to interfaces,...),
may lead the differentiator to build large matrices (such as the derivative of primitive flow variables with respect to conservative ones,...) *inside the differentiated routine.*

We concentrate now on the arrays that must be used as arguments of the subroutines. We shall state that:

All arrays are vectors, except the possible preconditioner of state and adjoint solution algorithms (that is a sparse matrix).

We state this with some details for the two most complex phases of the gradient computation.

4.4.1 Adjoint computation

As mentioned earlier, in the standpoint we propose in this work, the adjoint equation should be solved by a matrix-free iteration. Indeed, the differentiation of Sec.4.2 does not produce the adjoint matrix, but its product by a vector.
The matrix-free iteration can be an explicit pseudo-time advancing, subject to essentially the same "linear" stability conditions as the analogous explicit state system iteration

$$D_{\Delta t}\Pi \ + \ (\frac{\partial \Psi}{\partial W}(X,W))^t\Pi \ - \ (\frac{\partial J}{\partial W}(\gamma,W)) \ = \ 0 \ . \tag{30}$$

Another option is to apply a preconditioned fixed point algorithm in which the preconditioner can be the adjoint of the preconditioner used for solving the state equation:

$$A_1^t(\Pi^{iter+1} - \Pi^{iter}) \ = \ -(\frac{\partial \Psi}{\partial W}(X,W))^t\Pi^{iter} \ + \ (\frac{\partial J}{\partial W}(\gamma,W)) \ . \tag{31}$$

The only matrix stored is the preconditioner A_1^t.

4.4.2 Final derivatives

Once the adjoint state equation is solved, the adjoint Π is stored in a vector of dimension N. We examine now how to complete the computation of:

$$j'(\gamma) = (\frac{\partial J}{\partial \gamma}(\gamma,W)) - (\frac{\partial X}{\partial \gamma}(\gamma))^t(\frac{\partial \Psi}{\partial X}(X,W))^t\Pi \ . \tag{32}$$

By using Π as an input in the routine obtained by reverse differentiation in Sec.3.4, we obtain $(\frac{\partial \Psi}{\partial X}(X,W))^t\Pi$, a new array of dimension N.

The derivative of the mesh coordinates X with respect to the shape parameters is obtained either by hand differentiation or by Differentiation with the reverse mode. In that last case, the previous array $(\frac{\partial \Psi}{\partial X}(X,W))^t\Pi$ is used as an input in the cotangent code, allowing to obtain a new array, $(\frac{\partial X}{\partial \gamma}(\gamma))^t(\frac{\partial \Psi}{\partial X}(X,W))^t\Pi$, of dimension n.

We have demonstrated that the Automatic Differentiation with reverse mode allows the assembly of the adjoint system residual by using only one-entry arrays. Further, all the derivatives having a role in the optimality system are obtained in this way.

The rest of the software necessary to complete the sensitivity routine set is in the proposed strategy developed by hand and deals with
(1) the solution algorithm for the adjoint system,
(2) the final sum of the different terms.

The next part of the paper deals with the adjoint system solution algorithm.

5 Solution of the adjoint system

The above differentiation of state residual produces an adjoint state residual and not the adjoint Jacobian matrix. In order to find the adjoint state itself, making (by definition) this residual vanish, we need to apply a *matrix-free algorithm*, that is a solution method which does not need the exact matrix of the linear system to solve.

The discrete adjoint system comes from a linear partial differential system, involving first-order (spatial) derivatives. Let us assume this system is close to the steady Euler system or the corresponding linearised version and that its discretization can be solved in a similar manner. The choice of a matrix-free algorithm for solving an Euler-type system should *a priori* take into account the available experience in this field and also, in a more concrete standpoint, the solution algorithm already written in the flow analysis software under study. The development of solutions techniques for the Euler equations have been much influenced by the choice between explicit low-storage iteration and implicit -higher-storage-iteration. The implicit iteration involves generally the solution of a linearised system which preconditions the time advancing, allowing a faster convergence.

In Sec.4.4.1, we exhibited two possible answers:
- (a) we can consider solving it with an explicit time advancing iteration as in (30).
- (b) in the case where the initial software involves already a simplified preconditioner for the state equation, we can try to use it for solving the adjoint state equation as in (31).

With the option (a), we can use a time step deduced from the linear analysis that would apply to a similar time-advancing scheme for the state equation. Let us assume that the adjoint residual computation that we have derived (by Automatic Differentiation) is a few times, let us say k times more expensive in terms of computation time than the assembly of the state equation. Then each pseudo-time step of adjoint is k times more expensive than a pseudo-time step for state equation and thus this k factor would be the ratio between the computing costs of evaluations of adjoint state and direct state .
This evaluation can be slightly lowered by various factors such as the economy in time step size evaluation for adjoint, a better convergence of adjoint iteration due to linearity, a possible acceleration by residual averaging ([13]), and the possibly higher residual norm to impose before stopping adjoint iteration (higher that for state).

Note than the artificial time advancing can be replaced by a matrix-free linear GMRES (without preconditioner). This would result in a slightly larger complexity of each iteration but a faster convergence.

In our study, we consider option (b). It is indeed very natural to solve the adjoint state with the help of the transpose of the first-order Jacobian A_1 introduced in Sec. 2.2 as a preconditioner:

$$\left\{ \begin{array}{lcl} (A_1)^t \ \Pi^{(0)} & = & \dfrac{\partial J}{\partial W}(\gamma, W) \\ \\ (A_1)^t \ (\Pi^{(n+1)} - \Pi^{(n)}) & = & -(\frac{\partial \Psi_2}{\partial W})^t \ \prod^{(n)} + \dfrac{\partial J}{\partial W}(\gamma, W). \end{array} \right. \tag{33}$$

This algorithm applies a Defect Correction (**DeC**) iteration to a second-order accurate approximation of a first-order Friedrichs-type system preconditioned with a first order approximation. Desideri and Hemker ([3]) have extensively studied this type of iteration for the advection and Euler equations and proved that convergence rate can be mesh independant and as small as 0.5. In the sequel, we refer to algorithm (33) as the DeC one.

Remark Here we have not considered the addition of the mass-matrix $\dfrac{M}{\Delta t^n}$ term. It might be useful to re-introduce it when the linear system with A_1 reveals to be difficult to solve, but we have not studied this option in the present work.

6 Numerical illustrations

Showing how to use the gradient for optimizing the nozzle shape is out of the scope of this paper. We concentrate on the best way in computing the objective functional gradient. Efficiency, storage requirements and accuracy of the strategy proposed are measured and discussed for the model problem introduced in Sec.2.

6.1 Conditions of Numerical Experiments

We condider the following *target shape*:

$$\gamma_1 \ : \quad y(x) = \frac{1}{2} - \frac{1}{35} + \frac{1}{35} sin(\pi(x + \frac{1}{2})),$$

This shape γ_{target} is used for computing a *target pressure* and then specifies completely the objective cost function in continuous and discrete context.

The following *initial shape* is also considered:

$$\gamma_{init} \ : \quad y(x) = \frac{1}{2} - \frac{1}{40} + \frac{1}{40} sin(\pi(x + \frac{1}{2})).$$

The function γ_{init} specifies the initial domain Ω_{init} of the continuous optimisation loop.

For any mesh of Ω_{init}, the coordinates $\gamma_{init}(x_i)$ of γ_{init} of upper vertices specify the *discrete initial shape*, that is the point at which we want to compute the gradient of the discrete objective functional.

The flow conditions under study are defined as follows: farfield Mach number is 0.74.

Various meshes with respectively 400, 1240, 2760, and 4880 vertices are used in our experiments. In the present study we do not address the possible difficulties arising from truly unstructured meshes but instead we consider meshes of I-J type.

6.2 Direct state solution

In order to help analysis in the sequel, we depict the state variable for the initial shape. The flow is shows a classical transonic nozzle flow structure, with a supersonic pocket at nozzle, limited on downward (right) side by a shock (Mach is then as high as 1.4), the location of which is around the abscissa of 1.7, see the Mach number contours in Fig.5

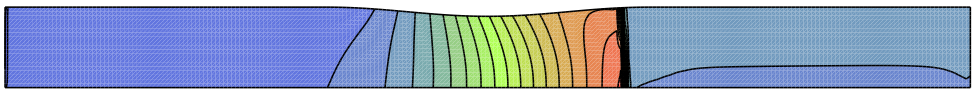


Figure 5: State solution, Mach number (without TVD limiters, 4880-node mesh)

6.3 Adjoint system solution

Let us assume that we apply a DeC loop with at each iteration a complete solution of the first-order system (in practise we have done this with many Jacobi iterations). Then the number of Dec iterations is not so small as predicted by Desideri and Hemker ([3]). But it takes for most meshes less than 200 DeC iterations for dividing the equation residual by 12 orders of magnitude.

In practice, we do not need a complete convergence of the first inner linear loop, but only a few tens to hundred Jacobi sweeps for a good convergence of DeC in less than 300 iterations. This small effort depends strongly on mesh size but the global scheme remains of comparable efficiency to the state solution algorithm. In particular, the comparably larger cost of co-state residual evaluation appears not to be a penalty when it is done only at most three hundred times.

Some contours of the adjoint state solution are depicted in Figs.6 and 7 with two different mesh fineness. Both computations are carried without TVD limiters in the whole chain of

calculations. The influence of limiters is discussed in the sequel.

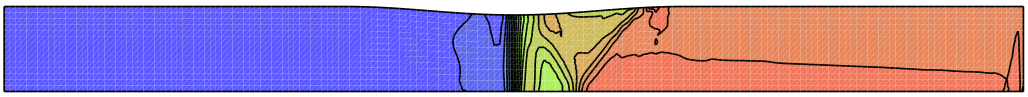


Figure 6: Adjoint state, second component, medium mesh of 2760 nodes

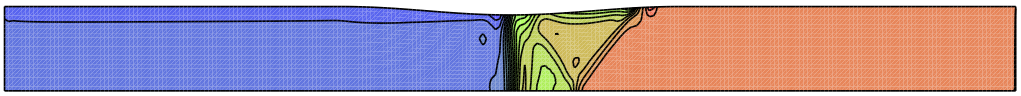


Figure 7: Adjoint state, second component, fine mesh of 4880 nodes

The understanding of the continuous adjoint system of Partial Differential Equations is a delicate question addressed already by several authors, see [2], [10], [11], [12]. One can remark indeed, that for the Euler equations, the adjoint system is first-order hyperbolic nature, linear, but non-conservative. It presents diverging characteristics at vicinity of state variable shocks, and converging characteristics near state variable sonic lines.

Concerning discontinuities, since the Euler solutions satisfying entropy condition are limits of viscous solutions, the adjoints are also limit of viscous adjoint solutions, which indicates that continuous matching could solve the apparent ambiguity of characteristics divergence.

As a result, a rather smooth behavior is observed on the adjoint in place where the state shows a strong shock.
On the contrary, direct-state sonic lines turn to produce on the adjoint a kind of linear shock.

6.4 Gradient validation

This strategy of exact analytic differentiation can be validated by comparison with divided differences. We emphasize that is not possible when using the direct discretization of the adjoint PDE equations.

We expect AD-based gradients to be at least as accurate as those provided by divided differences.

We present in Fig. 8 a comparison between the gradient values obtained with the presented analytic adjoint method and the values obtained by applying divided differences on each component of the parameter γ.

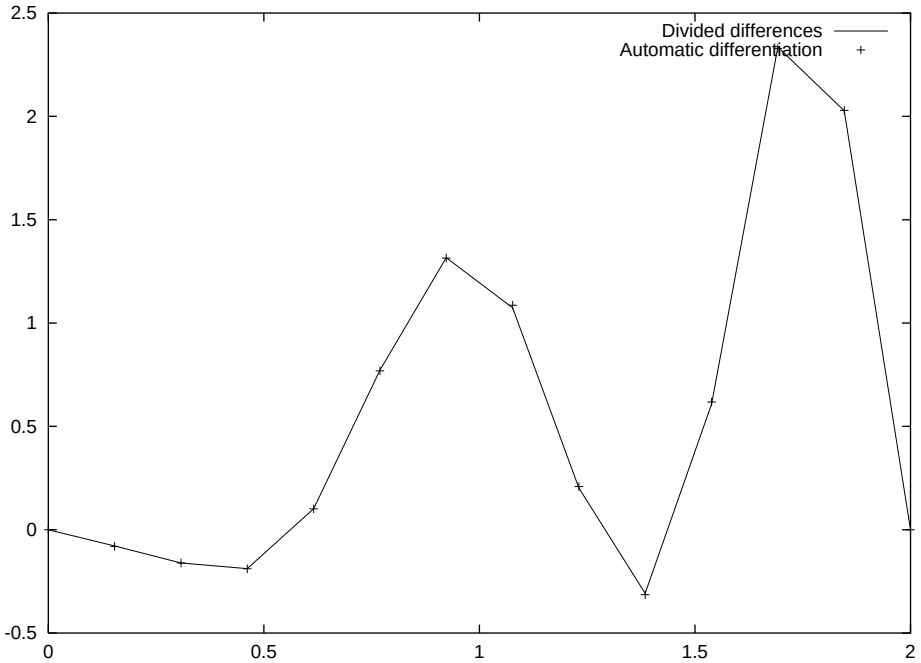


Figure 8: Gradient of the objective functional as a function of the horizontal independant variable. The continuous line is computed with the proposed exact adjoint method. The 14 crosses are obtained from applying sucessively divided differences to the 14 unknowns of the objective functional. Calculations are carried out for a mesh of 400 nodes

Although the mesh is a coarse one of 400 nodes, one must pay attention to the evaluation of the numerical gradients. Perturbations as small as 10^{-7} are applied to the control components, the state equation is solved to machine zero (10^{-14}), and second-order divided differences are applied. This does not produce the best accuracy for all components, but for any of these components, researching an adequate choice of the perturbation size leads to at least 6 digits identical to the A.D. results. The cost of such an evaluation is much larger than $2n$ times the solution of the state system for n parameters.

Let us remark that a much more accurate validation can also be performed by generating a forward mode sensitivity analysis with ODYSSEE.

Since the adjoint approach can be built also in a continuous (i.e. nondiscretized) context, it is reasonable to investigate the convergence of the discrete gradient to a continuous one. This study is sketched in Fig.9. We observe that most sensitive parameters correspond to sonic line and shock locations.

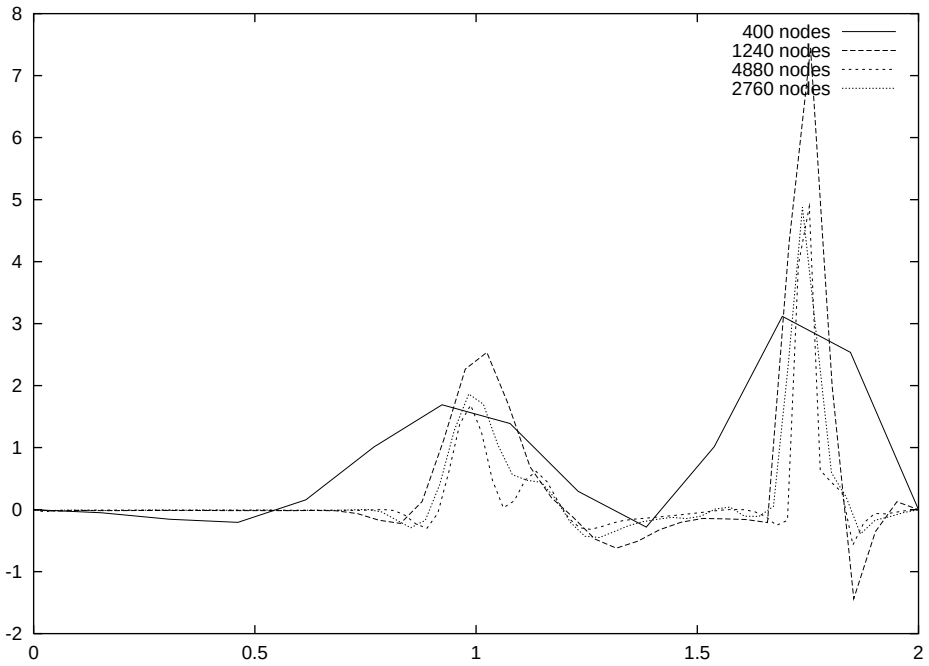


Figure 9: Convergence to the continuous limit for the gradient: discrete gradients from four meshes of 400, 1240, 2760, and 4880 nodes are presented

The above results where obtained without using the TVD limiters in the CFD kernel. We complete our examination of the gradient by a study of the impact of using limiters. We restrict our outputs to one mesh, with 1240 nodes. At this level of coarseness, the limiters have a medium influence on the state variables : inlet and outlet values are slightly changed, the shock moves a little downward and gets rid of the small overshoots, see the horizontal cut of Mach number in Fig. 10.

The adjoint state solution is more perturbed by this choice, since quasi-constant values show larger differences, see the horizontal cut of second component in Fig. 11.

In Fig. 12, we present the final impact of limiters on the gradient components. We observe that the global behavior is smoothened. At direct-state shock (abscissa near 1.7), the kind of Dirac is not moved, but is smoothened and has a lower peak. At sonic point,

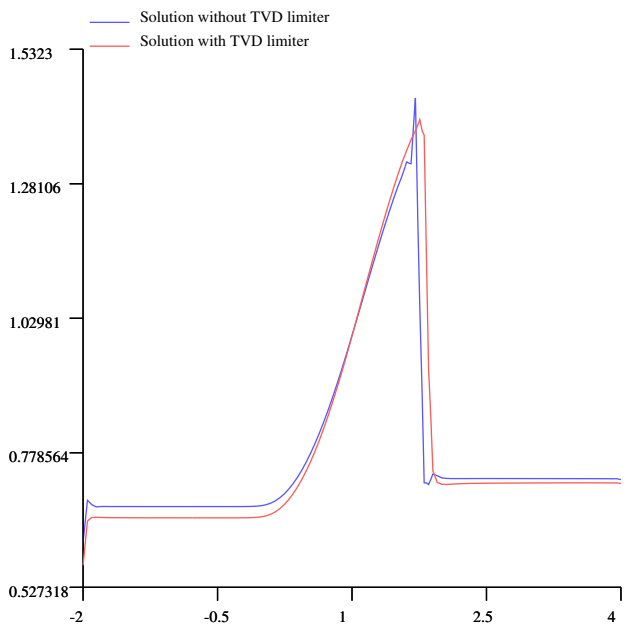


Figure 10: State solution, horizontal cut of the Mach number, with and without TVD limiters (1240-node mesh)

where we have a sort of shock for the adjoint state, limiters carry again some smoothing more difficult to interprete. On the other regions, values are unchanged while the adjoint shows different but constant values. This can be explained by the involvement of space derivatives in the gradient formulations, so that mean variations of state and adjoint have no consequence on the gradient.

7 Conclusion

Optimal design techniques are being renewed by adjoint methods. In the present report, we propose our adjoint-based gradient for a typical optimal design problem. For this, we describe a two-step strategy for the application of reverse mode automatic differentiation.

First, we explain how to derive an exact adjoint-based gradient without explicit storage of the Jacobian. This method is adapted to complex systems as those arising in compressible CFD, for which exact jacobian matrices are rarely stored, since they would take too much

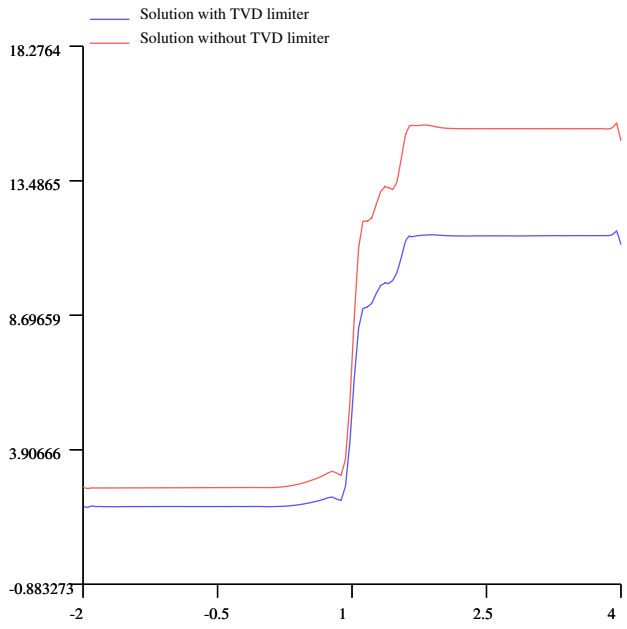


Figure 11: Adjoint state solution, horizontal cut of the second component, with (lower curve) and without TVD limiters (1240-node mesh)

memory space. The proposed method does not use a transposed Jacobian, but only the adjoint system residual.

Second, we explain how to obtain the adjoint residual routine by using the reverse mode of Automatic Differentiation. In the case of steady systems like those used in complex Continuum Mechanics, some manual pre- and post-processing are still necessary, because existing differentiation tools do not take advantage of the fact that iterations in assembly loops are independent. We describe the complete process leading to the assembly of the adjoint residual, and to the assembly of all other derivatives needed to computing the gradient.

The resulting differentiated software shows satisfying performances. We validated the accuracy of the gradient by comparison with divided differences.

In our strategy, Automatic Differentiation is applied only to the routines dealing with equations and functional assembly and never to solution algorithms. This means that new dedicated algorithms should be designed for the adjoint system.

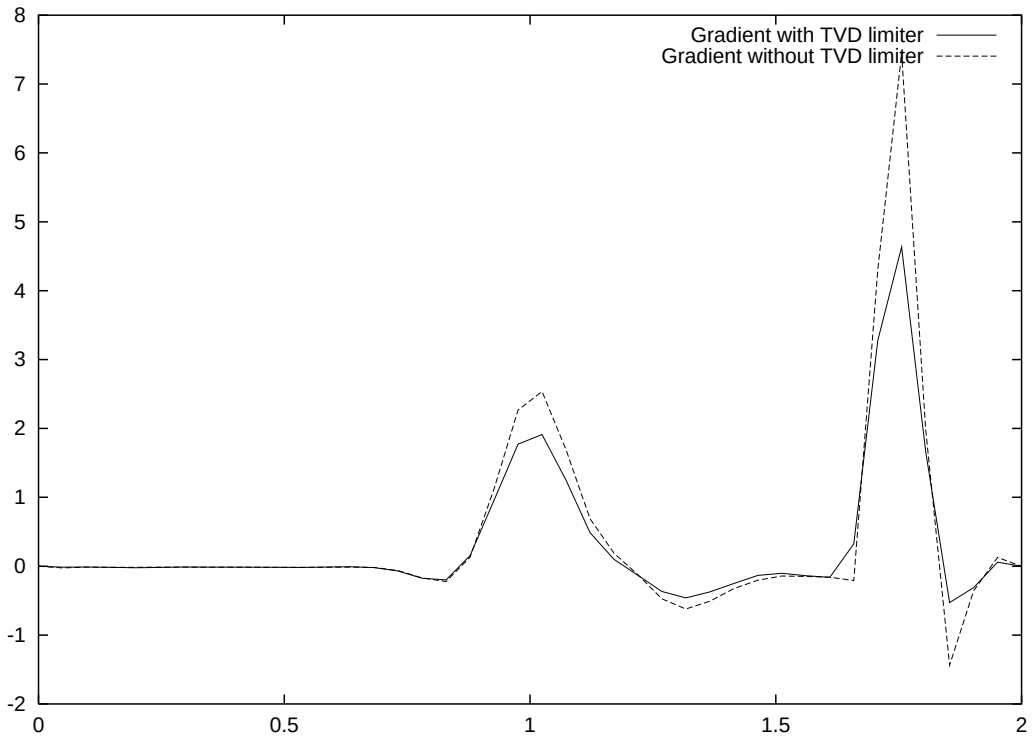


Figure 12: Gradient, with and without TVD limiters(1240-node mesh)

For solution adjoint system, we think that better algorithms can be imagined for replacing the rather rustical Defect Correction. They could be for example quasi-Newton ones, and would in particular allow efficient domain decomposition extensions.
For solution algorithms applying to the whole optimality system, we are thinking of two interesting directions. One is the adaptation of SQP optimization algorithms to large scale CFD systems. The other is the application of simultaneaous or one-shot iterations for the optimality system.
In both case, we need the adjoint residual, that will be computed in the way proposed here.

Acknowledgements

This work was partly supported by the EU Aeroshape project.

References

[1] Beux (F.). – Shape optimization of an Euler flow in a nozzle. in *Notes on numerical fluid mechanics*, vol. 55, Periaux *et al.* Eds, EUROPT - A European Initiative on Optimum Design Methods in Aerodynamics, pp. 115-131, Vieweg, Braunsweig Wiesbaden, 1994

[2] Cliff (E.) and Shenoy (A.) On the optimality system for the 1-D Euler flow problem *AIAA paper 96-3993, 6th AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, september 4-6, Bellevue, WA, 1996

[3] Desideri (J.-A.) and Hemker (P.W.). – Convergence analysis of iterative implicit and defect-correction algorithms for hyperbolic problems, *SIAM J. Sci. Comput.*, 88-118, 1995

[4] Francescatto (J.) and Dervieux (A.). – A semi-coarsening strategy for unstructured multigrid based on agglomeration, *International Journal for Numerical Methods in Fluids*, vol. 26, 1998, pp. 927–957.

[5] Griewank (A.). – Evaluating Derivatives Principles and Techniques of Algorithmic Differentiation. *Frontiers in Applied Mathematics* 19, SIAM, 2000

[6] Hascoet (L.), Fidanova (S.) and Held (C.). – Iteration-wise Adjoining, in AD2000, Nice 2000 accessible on site: `http://www-sop.inria.fr/tropics/Laurent.Hascoet/ index.html`

[7] Corliss, G. and Faure, C. and Griewank, A. and Hascoet, L. and Naumann, U. (editors). – Automatic Differentiation of Algorithms, from Simulation to Optimization. Springer, LNCSE, 2001

[8] Hascoet, L. and Fidanova, S. and Held, C., Adjoining Independent Computations, in [7], p. 185-190

[9] Hovland (P.), Mohammadi (B.) and Bischof (C.). – Automatic Differentiation of Navier-Stokes computations.– Argonne National Laboratory MCS-P687-0997, 1997

[10] Iollo (A.) and Salas (M.D.) Entropy jump across an inviscid shock wave *ICASE report 95-12 and Theoretical and Computational Fluid Dynamics*, 1995

[11] Iollo (A.) and Salas (M.D.) Contribution to the optimal shape design of two-dimensional internal flows with embedded shocks *ICASE report 95-20*, 1995

[12] Iollo (A.), Salas (M.D.) and Ta'asan (S.) Shape optimization governed by the Euler equations using an adjoint method *ICASE report 93-98*, 1993

[13] Jameson (A.). – *Aerodynamic design via control theory.* – Report 1824 MAE, Princeton University, New Jersey, 1988.

[14] Marco (N.) and Dervieux (A.). – Numerical optimizers for aerodynamic design using transonic finite-element solvers. – BRITE-ECARP Final report, July 1995.

[15] Mohammadi (B.). – Practical application to fluid flows of automatic differentiation for design problems. – von Karman Lecture Series, 1997

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399