Reverse Automatic Differentiation for Optimum Design: from Adjoint State assembly to Gradient Computation

Francois Courty*, Alain Dervieux*, Bruno Koobus†, Laurent Hascoët*

June 27, 2003

Abstract

Gradient descent is a key technique in Optimal Design problems. We describe a method to compute the gradient of a optimization criterion with respect to design parameters. This method is hybrid, using Automatic Differentiation to compute the residual of the adjoint system, and using this residual in a hand-written solver that computes the adjoint state and then the gradient. Automatic Differentiation is here used in its so-called *reverse* mode, with a special refinement for *gather-scatter* loops. The hand-written solver uses a matrix-free algorithm, preconditioned by the first-order derivative of the flux function. This method was tested on a typical optimal design problem, for which we give validation and performance results.

1 Introduction

Gradient descent is one of the key techniques for optimization, in particular for optimal design of systems governed by *Partial Differential Equations (PDE's)*. The so-called *adjoint systems* are getting increasingly popular to compute gradients.

Adjoint systems may be *(i)* hand-written from the initial PDE, and then discretized and solved by program [15]. Alternatively *(ii)*, the *reverse mode* of *Automatic Differentiation (AD)* can generate a new program that solves the adjoint system, from the program that solves the original PDE's. These two approaches are essentially very similar, and have comparable complexities. They can be compared as follows:

(i) A hand-written adjoint is very sound mathematically. Because the process starts back from the PDE's, it can be understood and trusted

*Projet Tropics, INRIA, France

†Université de Montpellier, France

better than *(ii)*. This process implies a new separate implementation phase to solve the adjoint system. During this manual phase, mathematical knowledge of the problem can be translated into many hand-coded refinements. But this may take an enormous engineering time. Except for special strategies (see [8]), this approach does not produce an exact gradient of the discrete functional, and this can be a problem if using optimization methods based on descent directions.

(ii) A pure AD adjoint is in fact the adjoint of the discrete functional computed by the software, which is piecewise differentiable. It produces exact derivatives almost everywhere. Theoretical results [7] guarantee convergence of these derivatives when the functional converges. This strategy gives reliable descent directions to the optimization kernel, although the descent step may be tiny, due to discontinuities. Validation is easier (e.g. by divided differences) because this computes the adjoint of the original program itself. Most importantly, AD adjoint is *generated* by a tool. This saves a lot of development and debug time. But this systematic approach leads to massive use of storage, requiring code transformation by hand to reduce memory usage. Mohammadi's work [13] [17] illustrates the advantages and drawbacks of this approach.

We propose a new strategy to solve the discrete adjoint system, which is a compromise between the two approaches above. Similarly to approach *(i)*, we write the adjoint system, but directly in the discrete case. These equations must be assembled and then solved. To do this, we first identify some parts of the discrete state equation and cost functional, whose differentials appear in the adjoint system. These differentials are evaluated by subroutines generated by the reverse mode of AD, like in approach *(ii)*. Then we use these subroutines to assemble the residual of the discrete adjoint system, and finally we solve this system to obtain the discrete adjoint. Notice that for this resolution, we have to design a matrix-free algorithm, because only the residual of the adjoint system is available.

Let us examine the benefits of our strategy, compared to previous related works. The class of problems we are addressing, as well as our application example, has two characteristics that we will take profit of: these are *steady models*, and their implementation uses many loops with *independent iterations* (*II*-loops for short), such as gather-scatter loops. These characteristics are independent, and can be exploited separately:

- **steady models:** In the general case of unsteady models, the adjoint, whether obtained by hand or through AD, requires storing the whole trajectory data, because reverse AD uses this data in the *reverse* of the order in which it is computed by the program. This storage is often extremely large, unless a strategy of recomputation is applied, *cf* [9].

Similarly, in the case of *steady models*, brute-force reverse AD would require the same enormous memory. Can we avoid this using the fact that the model is steady? Actually this is possible, as shown in [10], where computation of the adjoint state uses the iterated states in the direct order. Alternatively, most researchers (see for example [13]) use only the fully converged state to compute the adjoint. This is usually implemented by a hand modification of the code generated by AD. But this is delicate and error-prone. Our hybrid strategy also uses the converged state. AD is applied to the assembly loops, thus still saving a lot of discretization and implementation effort. On the other hand, a specific resolution algorithm is devised, specially adapted for the adjoint state. This technique requires no hand modification of the AD generated code.

- ***II*-loops:** Inside the part of the code which is differentiated automatically, we find many loops with independent iterations. Our contribution is a specific AD reverse mode for these loops, that requires far less memory. To our knowledge, this technique has been used occasionally [19], by hand, with no formal insurance of correctness. We gave a formal description of the technique in a previous paper [12] and a proof of correctness in [11]. Here we propose a systematic treatment of these loops inside our AD tool TAPENADE. The TAMC AD tool [6] also uses a special strategy for parallel loops. But TAMC implements a *"recompute-all"* strategy in the reverse mode, that spares memory at the cost of duplicate execution of the original program's instructions. Therefore the problem of memory is less crucial, and comparison with our approach, based on a *"store-all"* strategy, is difficult. Nevertheless, they are similar in the sense that both rely on the observation that the adjoints of independent computations are independent.

This paper is organized as follows. Section 2 presents our model application, from Computational Fluid Dynamics. The PDE's are discretized and solved by a program. Then we build the discrete adjoint PDE's, inside which we identify the sub-expressions that will be evaluated through AD. Section 3 describes the actual use of our AD tool TAPENADE, emphasizing the specific improvement for *II*-loops. Section 4 describes the final assembly and resolution of the gradient, using the AD-generated subprograms. Section 5 shows some numerical results.

2 A problem of shape optimization

We demonstrate the interest of the proposed method on a rather academic but yet representative problem: the optimization of the shape of a 2D nozzle, in which the flow is modeled by the Euler equations. Figure 1 shows the geometry and the triangular mesh. The median part of the nozzle can vary through its upper boundary, and the mesh varies accordingly. This problem has been studied in an European BRITE-ECARP project [1][16].

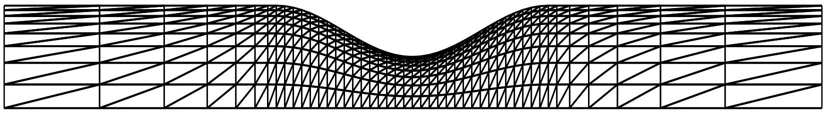

Figure 1: Nozzle mesh with 423 vertices

The control variable is a function γ of the abscissa which defines the shape of the upper boundary. For each value of γ, the flow $W(\gamma)$ in the nozzle is defined by the Euler equations

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

where

$$\Psi(\gamma, W) = \frac{\partial F(W)}{\partial x} + \frac{\partial G(W)}{\partial y}$$

completed with boundary conditions, and where

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

in which ρ is the density, (u, v) the velocity following (x, y), e the total energy, and p the pressure defined by

$$p = P(W) = (\kappa - 1)[e - \frac{1}{2}\rho(u^2 + v^2)] \quad \text{with} \quad \kappa = 1.4$$

On the flow is defined a cost function $j(\gamma) = J(\gamma, W(\gamma))$, which measures the discrepancy with a prescribed pressure P^{target} on the upper boundary

$$J(\gamma, W) = \int_\gamma (P(W) - P^{target})^2 d\sigma$$

The optimization problem is to minimize j with respect to γ.

We discretize this problem as follows. The mesh is originally an orthogonal triangulation of the rectangular domain, when $\gamma = 1$. For arbitrary γ, given by the sequence $\gamma_i = \gamma(x_i)$, for $i = 1, m$, the mesh is deformed by sliding the vertices along vertical lines. Thus the coordinates X of the mesh vertices are an explicit function of the γ_i.

The flux balance Ψ is discretized using an upwind finite-volume formulation, vertex-centered. This gives Ψ_1 at first-order and Ψ_2 at second-order:

$$\begin{aligned}
\Psi_1(X,W)_j &= \sum_{k\ neighbor\ of\ j} \Phi(W_j, W_k, \overrightarrow{\eta}_{jk}(X)) + \Phi_B(W_j, \overrightarrow{\eta}_j(X)) \\
\Psi_2(X,W)_j &= \sum_{k\ neighbor\ of\ j} \Phi(W_{jk}, W_{kj}, \overrightarrow{\eta}_{jk}(X)) + \Phi_B(W_j, \overrightarrow{\eta}_j(X))
\end{aligned}$$

where the elementary inter-cell flux Φ depends on the state variable W at first order through W_j and W_k, values of W at vertices j and k, and at second order through W_{jk} and W_{kj}, the linear interpolations on the j-k boundary according to van Leer "MUSCL" ideas, extended to triangulations as in [5]. The Φ_B component accounts for the boundary fluxes. The discretizations also depend on the mesh coordinate X through the integrated normal vectors $\overrightarrow{\eta}_{jk}(X)$. The upwinding in elementary flux Φ is the van Leer flux splitting, which is differentiable.

We could solve for the state iteratively, with a linearized implicit pseudo-time advancing (using n as the iteration index):

$$\left(\frac{M}{\Delta t^n} + \frac{\partial \Psi_2}{\partial W}(\gamma, W^n)\right)(W^{n+1} - W^n) = -\Psi_2(\gamma, W^n)$$

where M is a diagonal mass matrix. When the time step Δt^n grows infinitely, this becomes a Newton iteration for solving $\Psi(\gamma, W) = 0$. For the sake of readability, note that from now on, γ explicitly replaces X in the arguments of Ψ.

But the genuine Jacobian $\frac{\partial \Psi_2}{\partial W}(\gamma, W^n)$ uses too much memory space. Instead, we use in the computer code the simplified "spatially first-order" Jacobian matrix $A_1 = \frac{\partial \Psi_1}{\partial W}(\gamma, W^n)$. The linearized implicit time advancing then writes:

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

Operator A_1 acts as a preconditioner. We refer to [5] for a recent detailed analysis of A_1. Thanks to its quasi-diagonal dominance properties, the preconditioning system:

$$\left(\frac{M}{\Delta t^n} + A_1\right)\ \delta W = RHS$$

is easily solved with complete or partial convergence with a Jacobi 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 that uses a reasonable memory storage and that is much more efficient that an explicit one.

The discretized optimization problem is to minimize $j(\gamma) = J(\gamma, W(\gamma))$, but this time with $W(\gamma)$ solution of the discretized state system

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

Applying the chain rule in the differentiation of j, and introducing an adjoint state Π, we obtain the following optimality system:

$$\left\{\begin{array}{rcll} \Psi(\gamma, W) & = & 0 & \textit{(state system)} \\[2em] \frac{\partial J}{\partial W}(\gamma, W) - (\frac{\partial \Psi}{\partial W}(\gamma, W))^t \;.\; \Pi & = & 0 & \textit{(adjoint system)} \\[2em] j'(\gamma) = \frac{\partial J}{\partial \gamma}(\gamma, W) - (\frac{\partial \Psi}{\partial \gamma}(\gamma, W))^t \;.\; \Pi & = & 0 & \textit{(optimality condition)} \end{array}\right. \quad (1)$$

In this system, we highlighted the four expressions that we identified as differentials of expressions from the discretized original problem (J and Ψ). These differentials will be evaluated by subroutines generated by the reverse mode of AD, as described in section 3. Notice that the reverse mode does not explicitly generate the $\frac{\partial \Psi}{\partial W}^t$ and $\frac{\partial \Psi}{\partial \gamma}^t$ matrices, but only computes their product with a given vector. This is exactly what we will need in the sequel.

3 Using and Improving AD in Reverse mode

Automatic or Algorithmic Differentiation (AD) differentiates *programs*. By that, we mean that an AD tool takes as input a source computer program that, given an argument x, computes some function $f(x)$. The AD tool generates a new source program that, given the argument x, computes some derivatives of f. We refer the interested reader to the latest collection of articles [2] and to the recent monograph [9]. What we need here is the so-called *reverse* mode of AD, which we will present shortly. Then we will show how this reverse mode can be vastly improved in a specific case, and how this improvement can be automated.

Fundamentally, AD identifies computer programs with a composition of mathematical functions. Precisely, any program P composed of a sequence of instructions $I_k, k \in [1..p]$ and that implements a function $f, I\!R^m \rightarrow I\!R^n$, 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:

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

The *reverse mode* of AD produces a *gradient*, i.e. a program that, given x and a vector $\overline{y} \in I\!R^n$, computes the product $f'^t(x).\overline{y}$. 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 also referred to as the *adjoint mode*. From equation (2) and after transposition we obtain:

$$f'^t(x).\overline{y} = \begin{array}{l} (f'^t_1(x)) \\ .\,(f'^t_2 \circ f_1(x)) \\ .\,\ldots \\ .\,(f'^t_p \circ f_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x))\ .\ \overline{y} \end{array}$$

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

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

then it computes

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

and so on until

$$f'^t(x).\overline{y} = \overline{y}_0 = (f'^t_1(x)).\overline{y}_1$$

We observe that this process requires the intermediate values of the execution of P in the *inverse order* of their computation in 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 a tradeoff to be found in between, but in any case this is the main drawback of the reverse mode of AD.

Let us focus on the way our AD tool TAPENADE[14] handles this reverse mode. The choice is to *store* the intermediate values, and to retrieve them when needed. To begin with, suppose that the program to differentiate is a single subroutine. Let us call x_k the successive sets of intermediate values:

$$\begin{array}{rcl} x_0 & = & x \\ x_k & = & f_k(x_{k-1}) \qquad \textit{for } k = \textit{1 to } p \end{array}$$

The *reverse* differentiated program is composed of two successive parts. The first part, called the *forward sweep*, computes and stores the successive x_k. Therefore, the forward sweep is just a copy of the original program, interleaved with storage instructions. The second part, called the *adjoint sweep*, actually computes the derivatives, i.e. for $k = p$ down to 1, retrieves the set of intermediate values x_{k-1} and just after computes:

$$\overline{y}_{k-1} = (f_k'^t(x_{k-1})).\overline{y}_k$$

In the general case, there are many subroutines, arranged in a *Call Tree*. The tradeoff between storage and recomputation comes in at this level, as shown on figure 2. When a subroutine A calls a subroutine B, the forward sweep of A just calls the original B. Thus no storage is done inside B. During the adjoint sweep of A, one cannot simply call the adjoint sweep of B, because the intermediate values are lost. One must thus execute B again, this time as a forward sweep, and then do the adjoint sweep of B. Of course some values must be stored to allow for duplicate execution of B, but this is negligible compared to the size of intermediate values.

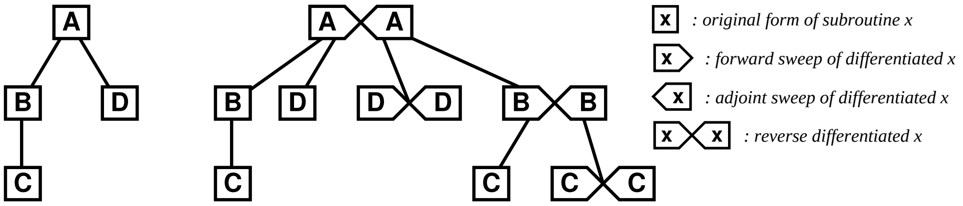

Figure 2: *storage/recomputation tradeoff on a Call Tree*

3.1 Improved AD for loops with independent iterations

Here we discuss an improvement to the reverse mode of AD, that vastly reduces memory consumption, for the very frequent case of loops whose iterations are data-independent. Notice that there exists a general framework to reduce this memory use, known as checkpointing [9]. For assembly loops, ad-hoc techniques have been proposed, for example by Hovland, Mohammadi, and Bischof [13], consisting in checkpointing each loop body.

Our strategy first observes that these assembly loops (*gather-scatter*) are essentially parallel. More precisely, there are no data dependences between different loop iterations, i.e. no loop-carried dependencies. In other words, iterations could be run in any order. We call such loops *II*-loops, for *Independent Iterations*. In [12], we introduced a new technique for the reverse differentiation of *II*-loops. This technique states that the standard reverse differentiation of an *II*-loop is equivalent to the improved form shown on figure 3. The notation $\overline{body}$ represents the adjoint sweep corresponding to

Standard:		*Improved:*
`do i= 1,N` *body(i)* `end` `do i= N,1` $\overline{body(i)}$ `end`	$\Longleftrightarrow$	`do i= 1,N` *body(i)* $\overline{body(i)}$ `end`

Figure 3: *Equivalent transformation of a reverse-differentiated II-Loop*

the forward sweep *body*. On the right part of figure 3, the forward sweep of each iteration *body(i)* is immediately followed by its adjoint sweep $\overline{body(i)}$, and therefore the intermediate values stored are used immediately after, and the corresponding memory is then available for next iteration. As a consequence, this manipulation reduces drastically the amount of memory used (by a factor N, size of the loop).

In [12], we gave the proof that the two forms are equivalent. We use the fact that the data-dependence graph of the adjoint sweep is isomorphic to the data-dependence graph of the forward sweep. This last point is demonstrated in [11]. Therefore, since the forward sweep is a loop with data-independent iterations, so is the loop of the adjoint sweep. So we can change the iteration order of the adjoint loop from `1` up to `N`. Observing that there are no data-dependences from the forward sweep to the adjoint sweep, apart from the storage/retrieval of intermediate values, we can then fuse the forward and adjoint loops together into a single loop.

3.2 Actual usage of the AD tool

We are now ready to use the reverse mode of AD to get the programs that evaluate the four differential expressions highlighted in equations (1). We start from the implementation of the discretized Ψ_2, which we find in the original PDE's solver. We also have at hand an implementation of the cost function J.

The cost $J(\gamma, W)$ is implemented by a function named `JCOST`, with two array arguments, `XMESH` and `WFLO`, representing γ and W. Since J is scalar-valued, its Jacobians $\frac{\partial J}{\partial W}$ and $\frac{\partial J}{\partial \gamma}$ are single-line matrices. The reverse differentiated function computes $f'^t(x).\overline{y}$ for any given $\overline{y}$, which is here a single-element vector. Therefore, in this special case, feeding $\overline{y} = 1.0$ to the reverse differentiated function directly yields the desired Jacobians. Thus we differentiate the function `JCOST` in the reverse mode, for output `JCOST` with respect to input `XMESH`. We obtain a subroutine which, given `XMESH`, `WFLO`, and `JCOSTB` (representing $\overline{y}$, initialized to 1.0), computes array `XMESHB` that holds $\frac{\partial J}{\partial \gamma}$. Similarly, substituting WFLO for XMESH in the above, we obtain a subroutine that computes $\frac{\partial J}{\partial W}$.

The second-order flux Ψ_2 is implemented by a subroutine named FLUX, whose inputs are again XMESH and WFLO, and whose output PSIFLUX holds $\Psi_2(\gamma, W)$. This time the Jacobians $\frac{\partial \Psi_2}{\partial W}^t$ and $\frac{\partial \Psi_2}{\partial \gamma}^t$ are really matrices, too large to be computed explicitly. To solve for Π in the adjoint system of system (1), we will use *matrix-free* techniques, such as GMRES [18], which only require a subroutine that computes

$$(\frac{\partial \Psi_2}{\partial W}(\gamma, W))^t \, . \, \Pi$$

for any given Π. This subroutine is exactly what is built by reverse AD of subroutine FLUX, for output PSIFLUX with respect to input WFLO. Similarly, substituting WFLO for XMESH in the above, we obtain a subroutine that computes $(\frac{\partial \Psi_2}{\partial \gamma}(\gamma, W))^t.\Pi$ for any given Π, as required in the optimality condition of system (1).

These differentiated programs are particularly efficient thanks to the improvement for *II*-loops. Now we will combine these differentiated programs to solve equations (1) efficiently, leading to the gradient $j'(\gamma)$. This is described in section 4.

4 Assembling the Gradient using AD routines

The gradient $j'(\gamma)$ is computed in two steps. First, we solve the adjoint system to get the adjoint state Π. Then we feed this Π into the optimality condition to compute $j'(\gamma)$. During this process, we take great care of the size of arrays. Thus, using a matrix-free technique allows us not to store the Jacobians of Ψ_2. In fact, the only matrix stored is the preconditioner A_1^t.

4.1 Defect-Correction resolution of the adjoint system

We need to solve the adjoint system by a matrix-free algorithm. Practically, we should take advantage of the existing resolution algorithm for the state system Ψ itself. From a more general point of view, let us assume that the adjoint system is close to the steady Euler system or to its corresponding linearized version and that its discretization can be solved in a similar manner. There is a considerable experience in solving Euler-type systems. One essential choice is between explicit (low storage) iteration and implicit (higher storage) iteration. The implicit iteration generally involves the resolution of a linearized system which preconditions the time advancing, resulting in a faster convergence. We considered two approaches for matrix-free resolution of the adjoint system, and we implemented the second approach.

- We can use an explicit pseudo-time advancing iteration.

$$D_{\Delta t}\Pi \ + \ (\frac{\partial \Psi_2}{\partial W}(\gamma, W))^t \Pi \ - \ (\frac{\partial J}{\partial W}(\gamma, W)) \ = \ 0$$

  Here we can use a time step deduced from the linear analysis that would apply to a similar time-advancing scheme for the state equation. Assuming that the AD-generated routine which computes

$$(\frac{\partial \Psi_2}{\partial W}(\gamma, W))^t.\Pi$$

  is slower by a small factor k than the routine which computes $\Psi_2(\gamma, W)$, and since this computation dominates the total computation time, this k is grossly the ratio between the evaluation costs of the direct state W and the adjoint state Π. This ratio can vary slightly, with some gains in evaluating the time step size for the adjoint, a better convergence of adjoint iteration due to linearity, and some losses such as the possibly higher residual norm to impose before stopping adjoint iteration (higher than for the 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.

- We can apply a preconditioned fixed point algorithm in which the preconditioner can be the adjoint of the preconditioner A_1 used for solving the state equation

$$A_1^t(\Pi^{iter+1} - \Pi^{iter}) \ = \ -(\frac{\partial \Psi_2}{\partial W}(\gamma, W))^t \Pi^{iter} \ + \ (\frac{\partial J}{\partial W}(\gamma, W)).$$

  This very natural approach leads to our implemented algorithm:

$$\begin{cases} A_1^t \ \Pi^0 & = \ \frac{\partial J}{\partial W}(\gamma, W) \\ A_1^t \ (\Pi^{n+1} - \Pi^n) & = \ -(\frac{\partial \Psi_2}{\partial W})^t \ \Pi^n + \frac{\partial J}{\partial W}(\gamma, W) \end{cases}$$

  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 [4] have extensively studied this type of iteration for the advection and Euler equations and proved that convergence rate can be mesh independent and as small as 0.5.
  Remark: here we have not considered the addition of the mass-matrix $\frac{M}{\Delta t^n}$ term. It might be useful to re-introduce it when solving the linear system with A_1 turns out to be difficult.

4.2 Computation of the gradient

There is no major difficulty left for computing the gradient:

$$j'(\gamma) = \frac{\partial J}{\partial \gamma}(\gamma, W) - (\frac{\partial \Psi_2}{\partial \gamma}(\gamma, W))^t.\Pi$$

We use the AD-provided routine that computes $(\frac{\partial \Psi_2}{\partial \gamma}(\gamma, W))^t.\Pi$, and we feed it with the Π computed above. Again, the Jacobian matrix $(\frac{\partial \Psi_2}{\partial \gamma})^t$ is never explicitly computed nor stored, and the result is a single vector. Similarly, we saw that another AD-provided routine directly computes the second vector $\frac{\partial J}{\partial \gamma}$, and we just substract these two vectors to obtain $j'(\gamma)$.

5 Numerical Application

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 to compute the gradient of the objective functional. We discuss efficiency and accuracy of our proposed strategy for the model problem introduced in section 2.

5.1 Resolution of the adjoint system

We focus here on the execution time for solving the adjoint system for Π, compared to the time for solving the state system for W. Our timings show that solving for Π is about 4 times as long as solving for W. Both operations have the same overall structure. They both iterate until convergence of W_n [*resp.* Π_n], and each step consists of two main parts: an assembly of a residual $r_n = -\Psi_2(\gamma, W^n)$ [*resp.* $-(\frac{\partial \Psi_2}{\partial W}(\gamma, W))^t\Pi^n + (\frac{\partial J}{\partial W}(\gamma, W))$], followed by a number of Jacobi iterations to solve for the increment δW [*resp.* $\delta\Pi$], defined by the linear system $(\frac{M}{\Delta t^n} + A_1)\delta W = r_n$ [*resp.* $A_1^t \delta\Pi = r_n$].

We first investigated the tradeoff between the degree of precision of the Jacobi iterations and the overall number of iterations until convergence. When we solve fully for the increment, with a few hundreds of Jacobi iterations, the overall number of iterations is not as small as predicted by Desideri and Hemker [4], but for most meshes, 200 iterations are enough to divide the residual by 10^{12}. On the other hand, if we solve less completely for the increment with, say, ten times fewer Jacobi iterations, then convergence is achieved in less than 300 overall iterations. Therefore, this second approach is more efficient.

With this second approach, the assembly of residual $-\Psi_2(\gamma, W^n)$ takes roughly as long as the solving for the increment δW, and also as long as the solving for $\delta\Pi$. On the other hand, the assembly of the adjoint residual takes roughly seven times longer than the assembly of the state residual. This ratio of seven is higher than the three to four ratio predicted by the theory of Automatic Differentiation. We can think of two possible reasons: one is the presence of small subroutines, for which the storage/recomputation tradeoff of TAPENADE (figure 2) generates unnecessary duplicate executions. The other reason is the presence of very long expressions, that generate many duplicated expressions in the differentiated code. We believe that an improved code analysis in TAPENADE can yield a ratio better than seven.

Finally, counting 1 for the time of assembly of the state residual, we obtain a total of 2 for one overall iteration of the state, and 8 for one overall iteration of the adjoint. In both cases, convergence requires 300 steps. This explains the ratio of four that we observed between solving for Π and solving for W.

5.2 Validation of the gradient

We validate the gradient $j'(\gamma)$, where γ is defined by 14 control points $\gamma_i = \gamma(x_i)$ for $i = 1, 14$. The gradients are trivial for $i = 1$ and $i = 14$, so these points are not presented. This validation is done for a subsonic flow and for a supersonic flow, in order to make a more complete coverage of the differentiated code. On each case and for each control point, table 1 shows the relative error between the values of $j'(\gamma)$ computed by our algorithm and the corresponding values evaluated by Divided Differences (DD). The correspondence is good taking into account the usual inaccuracy of Divided Differences. Notice that these DD values are most expensive to compute, essentially for two reasons:

- Two evaluations of the objective function j are required for each control parameter. This makes a total of 28 evaluations here.
- Each of these 28 iterations indeed requires numerous executions of j to select a good ϵ.

x_i:	0.154	0.308	0.461	0.615	0.769	0.923
subsonic case:	1.0e-7	1.7e-7	2.3e-7	3.2e-7	1.4e-7	5.4e-9
supersonic case:	4.0e-6	1.3e-5	1.4e-5	9.7e-8	1.1e-7	1.6e-7
x_i:	1.077	1.231	1.385	1.538	1.692	1.846
subsonic case:	1.3e-8	1.5e-8	2.0e-5	8.2e-9	1.4e-7	4.0e-8
supersonic case:	2.4e-7	3.0e-8	8.1e-8	6.3e-8	3.7e-8	1.1e-8

Table 1: *Relative errors between gradients computed by Automatic Differentiation and by Divided Differences ($\epsilon = 10^{-8}$).*

5.3 Discrete adjoint versus continuous adjoint

As we saw in section 1, an alternative approach uses a continuous (*i.e.* nondiscretized) adjoint. It is then interesting to investigate the convergence of our discrete gradients to the continuous gradient. The results are shown on figure 4. We observe that the discrete gradients vary substantially, es-

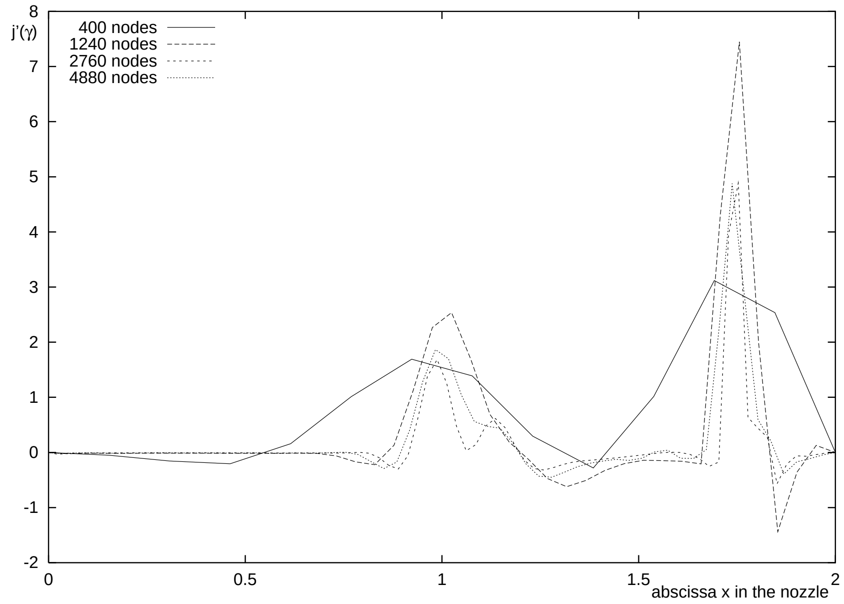

Figure 4: *Evolution of the discrete gradient as the mesh is progressively refined*

pecially at some locations that correspond to sonic lines and shocks in the state W. A priori, to optimize a discrete functional corresponding to a given discretization, it is safest to use the discrete gradient based on the same discretization. Strictly speaking, using a continuous adjoint only leads to an approximation of this discrete gradient, and figure 4 shows that this approximation can be very coarse. Therefore, using the continuous adjoint is dangerous and may degrade the results of optimization.

6 Conclusion

Adjoint methods are giving a new impulse to optimal design techniques. Here, we describe a method for the computation of an adjoint-based gradient for a typical optimal design problem. This method has essentially two levels.

At the upper level, we explain how to derive an exact adjoint-based gradient without storing the Jacobian explicitly. This is valid even for systems as complex as those arising in compressible CFD. For these systems, the exact Jacobian matrices are often too large to be stored. Here, we do not use the (transposed) Jacobian, but only the adjoint system residual.

At the lower level, we explain how to obtain the adjoint residual by using the reverse mode of Automatic Differentiation. In the case of steady systems like those used in complex Continuum Mechanics, the original program contains a large number of loops with independent iterations, such as the frequent *gather-scatter* loops. For these loops, we propose an improved reverse mode of AD, which uses far less memory space. We describe the complete process leading to the assembly of the adjoint residual, and to the assembly of all other derivatives needed at the upper level.

The resulting differentiated software completely computes the gradient, with satisfying performances. The gradients obtained are validated by comparison with divided differences.

In our strategy, Automatic Differentiation is applied only to the routines dealing with equations and cost function assembly and never to solution algorithms. In practice, the user still has to select and use an appropriate solver for the adjoint equation. In this work, we have chosen a pure Defect Correction fixed point. In the future, for solving the adjoint system, we think that better algorithms could be designed. They could be for example quasi-Newton, and would allow for efficient domain decomposition extensions. To solve the complete optimality system, we think of two interesting directions. One is the adaptation of SQP optimization algorithms to large scale CFD systems. The other is the application of simultaneous [3] or one-shot [20] iterations for the optimality system. In both cases, we need the adjoint residual, that we will compute as shown here.

Acknowledgments

This work was partly supported by the EU Aeroshape project.

References

[1] F. Beux. Shape optimization of an euler flow in a nozzle. *Notes on numerical fluid mechanics*, 55:115–131, 1994.

[2] G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann(editors). *Automatic Differentiation of Algorithms, from Simulation to Optimization.* Springer, 2001. Selected proceedings of AD2000, Nice, France.

[3] A. Dadone and B. Grossman. Progressive optimization of inverse fluid dynamic design problems. *Computer and Fluids*, 29:1–32, 2000.

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

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

[6] R. Giering. Tangent linear and adjoint model compiler, users manual. Technical report, 1997. http://www.autodiff.com/tamc.

[7] J.C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21, 1992.

[8] M.-B. Giles. Adjoint methods for aeronautical design. In *Proceedings of the ECCOMAS CFD Conference*, 2001.

[9] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Frontiers in Applied Mathematics, 2000.

[10] A. Griewank and Ch. Faure. Reduced Gradients and Hessians from Fixed Point Iteration for State Equations. *Numerical Algorithms*, 30(2):113–139, 2002.

[11] L. Hascoet. The data-dependence graph of adjoint programs. research report 4167, INRIA, 2001.

[12] L. Hascoet, S. Fidanova, and C. Held. Adjoining independent computations. *in [2]*, pages 185–190, 2001.

[13] P. Hovland, B. Mohammadi, and C. Bischof. Automatic differentiation of navier-stokes computations. Technical Report MCS-P687-0997, Argonne National Laboratory, 1997.

[14] INRIA Tropics team. On-line documentation of the Tapenade AD tool. Technical report. http://www.inria.fr/tropics.

[15] A. Jameson. Aerodynamic design via control theory. Report 1824 MAE, Princeton University, New Jersey, 1988.

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

[17] B. Mohammadi. Practical application to fluid flows of automatic differentiation for design problems. *Von Karman Lecture Series*, 1997.

[18] Y. Saad and M.H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *J. Sci. Comput.*, 7:856–869, 1986.

[19] C. Sevin. *Optimisation de formes en mécanique des fluides numérique.* PhD thesis, Université Pierre et Marie Curie, 1999.

[20] S. Ta'asan, G. Kuruvila, and M.D. Salas. Aerodynamic design and optimization in one shot. In *30th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, AIAA Paper 91-0025*, 1992.