Optimization Loops for Shape and Error Control

Alain Dervieux [1] **Laurent Hascoët** [1] **Mariano Vázquez** [1,2] **Bruno Koobus** [1,3]

Abstract

This paper proposes a strategy to derive an adjoint-based optimization code from a numerical simulation code. The strategy involves utilization of Automatic Differentiation (AD), and in fact was a motivating application for several studies and improvements of AD. The strategy gives best results when interfaced with modern optimization methods, such as "one-shot" or "multi-level". This paper presents a set of application examples where this strategy was used, and gives some experimental results. We also discuss some theoretical questions related to the computation of gradients and the optimization process in general.

Key Words: Automatic Differentiation, Optimization, Computational Fluid Dynamics

1 INTRODUCTION

For a long time, industrial CFD numerical studies have been addressing simulation quasi-exclusively. Now more and more studies address the next step, i.e. Optimal Control problems, starting from some existing simulation code. This can be done with any kind of simulation code, although nowadays this is still limited in most cases to steady models. Optimal Control on genuine unsteady simulations still proves too expensive in an industrial context. Furthermore, Optimal Control methods can also address other questions, such as the control of numerical errors that we will describe below.

Development of an Optimal Control application starts from an existing Simulation application, and reuses key parts of it. Let us identify these parts. Given a set of parameters, a simulation software gives a prediction of a physical process. For instance a numerical wind tunnel predicts the flow of air around a plane shape, using only computation. The predicted flow is the unique solution of a (set of) mathematical equation which we call the *state equation*, according to Optimal Control terminology. Numerical resolution of the discretized state equation involves in fact two important parts.

1 INRIA Sophia-Antipolis, France
2 Universitat de Girona, Spain
3 Université de Montpellier II, France

SAROD-2005
Published in 2005 by Tata McGraw-Hill

- One is the assembling part: for given arbitrary values of the state variables W, and using values of external parameters γ (e.g. the geometry), it computes a *residual* array, which reflects how the state variables satisfy the state equation. It is therefore

  state-assembler: $(\gamma, W) \mapsto \Psi(\gamma, W)$

  where Ψ is in fact the left-hand side of the discretized state equation

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

- The other is the resolution algorithm: For the given fixed external parameters γ, it uses the residual returned by the state-assembler to produce the state solution $W(\gamma)$ that nullifies the residual (or at least makes it sufficiently small).

  state-resolution: $\gamma \mapsto W(\gamma)$
  such that $\Psi(\gamma, W(\gamma)) = 0.$

  This is most often done iteratively, by incremental modifications of an arbitrary initial state, each modification driven by the residual for the current state.

To turn a simulation application into an Optimal Control application requires an additional ingredient, the *objective functional* that will evaluate a scalar cost for any possible parameters and state. Given a set of parameters γ, which we now view as *control parameters*, and given the corresponding solution state $W(\gamma)$, it computes one (or several) optimization criterion, i.e. the value of the objective functional for these control parameters and state. This objective functional takes

into account all industrial targets and constraints for a given process or product.

objective-assembler: $(\gamma, W) \mapsto J(\gamma, W)$

The goal of Optimal Control is to find control parameters γ which will make the objective functional smaller.

When the size of the parameter array is small, many methods can find the minimum easily enough. However for large numbers of control parameters, approaches that use analytic gradients become necessary. We advocate the following strategy to obtain these gradients:

- from the state-assembler, develop a **gradient-assembler** that computes the residual corresponding to derivatives of the state equation. This part of the strategy relies on *Automatic Differentiation* (AD).
- develop an adequate resolution algorithm for the gradient, that uses the gradient-assembler to iteratively find the requested gradient.

In this paper we analyze this strategy to compute gradients, which ultimately yields an Optimal Control loop. One objective is to maximize the re-use of code from simulation codes into the optimization code. This strategy strongly relies on Automatic Differentiation and especially on the ability of its "*reverse mode*" to produce efficient adjoint codes. It motivated a lot of improvements and research in AD, that we will describe in part. We demonstrate the strategy on a few illustrative applications. The outline is as follows: in section 2, we describe our model of optimal control problems, and the general strategy to solve them. Section 3 describes several example problems which can be expressed in the general framework of optimal control. Section 4 discusses some points specific to the Automatic Differentiation technique that is used in particular to build the *gradient-assembler*. In parallel, section 5 discusses the resolution algorithms that we use to obtain the gradients. Finally section 6 shows the results obtained on the example problems of section 3.

2 THE OPTIMAL CONTROL MODEL

Our framework is the following general constrained minimization problem:

$$\text{Arg Min } J(\gamma, W), \quad \text{subject to} \quad \Psi(\gamma, W) = 0 \tag{1}$$

where the minimum is taken with respect to the composite variable $x = (\gamma, W)$. In other words, we want to find the $x_{opt} = (\gamma_{opt}, W_{opt})$ that minimizes the *objective functional* $J(x)$, where x_{opt} must in addition satisfy the *equality constraint* $\Psi(x) = 0$.

If we define the reduced objective functional

$$j(\gamma) = J(\gamma, W(\gamma)), \tag{2}$$

where $W(\gamma)$ is the unique W solution of equation $\Psi(\gamma, W) = 0$, our problem is equivalent to minimizing the reduced objective functional j:

$$\text{Arg Min } j(\gamma). \tag{3}$$

Let us assume that the Jacobian

$$A = \frac{\partial \Psi}{\partial W} \tag{4}$$

is always invertible. Then the minimum we are looking for is the solution of the following *Karush-Kuhn-Tucker* (KKT) system:

$$\begin{cases} \Psi(\gamma, W) = 0 & \textit{(State)} \\ \dfrac{\partial J}{\partial W}(\gamma, W) - \left(\dfrac{\partial \Psi}{\partial W}(\gamma, W)\right)^* \cdot \Pi = 0 & \textit{(Adjoint state)} \\ \dfrac{\partial J}{\partial \gamma}(\gamma, W) - \left(\dfrac{\partial \Psi}{\partial \gamma}(\gamma, W)\right)^* \cdot \Pi = j'(\gamma) = 0 & \textit{(Optimality)} \end{cases} \tag{5}$$

This is the system that the Optimal Control loop must solve. Formally, this involves as usual an assembly step and a resolution step. The assembly step will take as input the current value of the variables that will eventually hold the result, which are:

- the control parameters γ,
- the state variables W,
- the *co-state* or *adjoint* variables Π.

The assembly step will compute each left-hand-side in system (5), and the resolution step will use them to update the variables until the residual is zero. Since the system is non-linear, this process will be iterative. Thus assembly and resolution will be called repeatedly.

We observe that parts of both steps are already available in the existing simulation code. Specifically, the assembly of the Ψ residual, and the resolution for W, i.e. what concerns the non differentiated symbols.

For the assembly part, What is missing is the terms that involve derivatives of Ψ and J. We will derive their code from the assembly code of Ψ and the computation code of J, using Automatic Differentiation (AD). We remark that the two terms that involve derivatives of Ψ are indeed of the *transposed-Jacobian-times-vector* kind. The same holds for the terms that

involve derivatives of J, only in the degenerate case of a single row Jacobian. Therefore, we will use the so-called *reverse mode* of AD which is able to produce code that computes transposed-Jacobian-times-vector derivatives in remarkably few computations. We will detail this in section 4.

For the resolution part, we need a composite algorithm that will combine

1. the existing resolution of the state equations, yielding W,
2. with a resolution algorithm for the adjoint state equations, yielding Π,
3. and with a minimization algorithm for the optimality equations, yielding the optimal control parameters γ_{opt}

We must develop the algorithms for Π and γ. In theory, the algorithm for Π and its usage in the assembly of $j'(\gamma)$ could be generated automatically, by reverse-mode AD of the existing algorithm for $j(\gamma)$. However, for efficiency reasons explained in section 5, we think it is better to write the resolution for Π by hand. Moreover, the resolution for Π can make use of crucial parts of the existing resolution for W, and is itself a key component to be reused in many places, as we show for the second-order derivatives that are needed for robust optimization, *cf* section 3.2.

3 EXAMPLES

We choose three example applications of our strategy, for optimal shape design, robust optimal design, and mesh optimization.

3.1 Shape design

This example was presented in [16, 12]. The goal is to evaluate and minimize the *sonic boom downwards emission* (SBDE), modeled as the volume integral of the squared pressure gradient in an observation box Ω_B below a supersonic aircraft, as shown in Fig. 1. The objective functional j combines this integral of the pressure gradient with deviations from prescribed lift L_0 and drag D_0, with relative coefficients α_1, α_2 and α_3.

$$j(\gamma) = \alpha_1(D-D_0)^2+\alpha_2(L-L_0)^2+\alpha_3\int_{\Omega_B}|\nabla p|^2 \quad (6)$$

Values of L, D, and p are derived simply from W, which is the solution of $\Psi(\gamma, W) = 0$, the 3D Euler equations around the geometry specified by the control parameters γ. Practically, Ω_B is a part of the computational domain placed just below the airplane. Of course, in presence of shocks, the value of the SBDE term is infinite in the continuous case, but it is finite (and mesh dependent) in the discrete case.

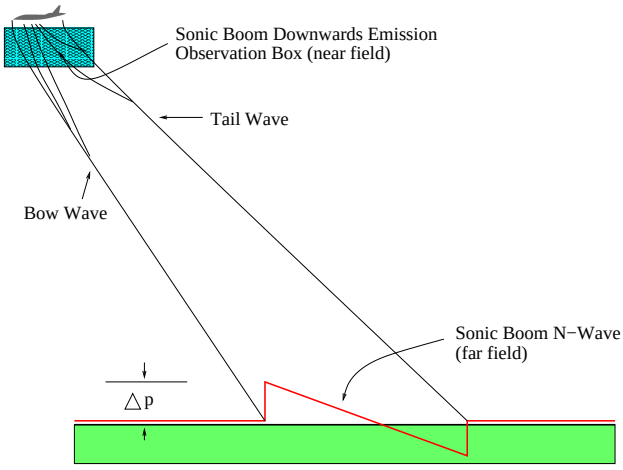

Fig.1: The sonic boom problem

3.2 Second derivatives for robust optimization

In this example, we suppose again that we simulate some state W that depends on some control parameters γ through a state equation, and search the γ that minimizes a given objective functional J. In addition, we want to study the influence of the uncertainties on (a part of) γ on the optimal J. This requires the second derivative $j''(\gamma)$. For example, the objective functional may somehow incorporate the goal of minimizing $j''(\gamma)$, thus making the optimum found more robust or tolerant to small variations of γ.

We can differentiate the equation of $j'(\gamma)$ in the KKT system (5), which yields equation (7). We took away the evaluation point (γ, W) for readability.

$$j''(\gamma) = \frac{d}{d\gamma}\left(\frac{\partial J}{\partial \gamma}\right) - \frac{d}{d\gamma}\left(\frac{\partial \Psi^*}{\partial \gamma}\right)\cdot\Pi - \frac{\partial \Psi^*}{\partial \gamma}\cdot\frac{d\Pi}{d\gamma} \quad (7)$$

Every term in equation (7) can be computed in the manner presented in section 2: using a suitable resolution algorithm to solve linear equations whose right-hand side are assembled through Automatic Differentiation of existing code.

Let's focus for example on the term $\frac{d\Pi}{d\gamma}$. The adjoint state equation in the KKT system (5) is an implicit definition of Π, which we can differentiate with respect to γ. Thus we obtain

$$\underbrace{\left(\frac{\partial \Psi}{\partial W}\right)^*\cdot\frac{d\Pi}{d\gamma}}_{B_1} = \underbrace{\frac{d}{d\gamma}\left(\frac{\partial J}{\partial W}\right)}_{B_2,\ B_3} - \underbrace{\frac{d}{d\gamma}\left(\frac{\partial \Psi^*}{\partial W}\cdot\Pi_\gamma\right)}_{B_4,\ B_5}$$

in which we observe that terms in boxes B_3 and

B_5 already appeared in the adjoint state equation of the KKT system (5), and we know already that the reverse mode of Automatic Differentiation will provide the code for them. The terms in boxes B_2 and B_4 are tangent derivatives that will be obtained by AD in tangent mode of the code for boxes B_3 and B_5 respectively. This introduces a new usage of AD, which we call the "tangent-on-reverse" mode, which is presently under development.

We observe finally that the resolution in box B_1 is the same as the resolution for Π in our general framework, only with a different right-hand side. We can therefore reuse the same specific resolution algorithm as we used for Π. Notice however that one invocation will only return one column of $\frac{d\Pi}{d\gamma}$, which is now a matrix. This must be iterated for each component of γ. This is not surprising since second derivatives are indeed larger objects than adjoint states.

3.3 Numerical error control by mesh optimization

A good mesh adaption requires a good optimization criterion, i.e. a good measure of the quality of a given mesh. The goal of this application example is to find, for a given numerical problem, the optimal mesh *density* (a scalar function on the computation domain Ω) that minimizes the approximation error.

However, the approximation error is a complex non-local function of mesh fineness. To overcome this difficulty, our idea is to define the error as the solution of a linear error system whose right hand side is the truncation error and therefore depends only on the local quality of the mesh. This is based on *a priori* estimates.

Call u the continuous solution of a Dirichlet problem:

$$\Delta u = f \ \text{ on } \Omega \ ; \ u = 0 \ \text{ on } \partial\Omega. \tag{8}$$

Consider a family of meshes $\mathcal{M}_h$ of Ω, each of them following a mesh density d_h. Each d_h is derived from a fundamental d by $d_h = h \times d$. Thus increasing h means uniformly increasing the mesh fineness.

For a given $\mathcal{M}_h$, we get a discrete solution u_h. The approximation error function can be split in two:

$$u - u_h \ = \ (u - \Pi_h u) \ + \ (\Pi_h u - u_h) \tag{9}$$

where we introduce $\Pi_h u$, the projection of u on V_h. V_h is the subspace of functions that are $\mathcal{P}_1$ on $\mathcal{M}_h$, i.e. continuous, and linear on each cell of $\mathcal{M}_h$.

We choose the implicit error $u_h - \Pi_h u$ as our model for the approximation error. Many studies have chosen the interpolation error $u - \Pi_h u$ instead. We shall see in the results part 6.2 the advantages of our choice.

In any case, the behavior of the two terms is similar in the following respect: for all test function v on Ω, one can show that

$$\int_\Omega \nabla(u_h - \Pi_h u)\cdot\nabla\Pi_h v \, d\Omega = \int_\Omega \nabla(u - \Pi_h u)\cdot\nabla\Pi_h v \, d\Omega \tag{10}$$

A truncation error analysis shows that the right-hand side of (10) is in turn equal to

$$\int_\Omega h^2 g \, v \, d\Omega \ + \ O(h^3) \ , \tag{11}$$

where g is a long but simple expression involving 4^{th} derivatives of u and d. In practice, derivatives of u will be evaluated from the discrete u_h. Finally if we define W as the solution of the Dirichlet problem

$$\Delta W = -g \ \text{ on } \Omega \ ; \ W = 0 \ \text{ on } \partial\Omega \ , \tag{12}$$

the expression (11) is in turn equal to

$$\int_\Omega h^2 \nabla W \cdot \nabla v \, d\Omega \ + \ O(h^3) \ . \tag{13}$$

Comparing the left-hand side of (10) and expression (13), we boldly choose as our error model the L^2-norm of W. Full justification is out of the scope of this paper and is available in [4, 5].

We end up with an Optimal Control problem: find the (parameterized) mesh density function d that minimizes the objective functional

$$j(d) \ = \ J(d, W(d)) \ , \tag{14}$$

where J probably involves the norm of its second argument, and where this second argument W is defined implicitly as the solution of the Dirichlet problem (12). Although the theoretical justification relies on the fact that we solve a Dirichlet problem, extension is possible to more general equations and indeed practical application is already in progress on realistic CFD equations.

4 REVERSE-MODE AUTOMATIC DIFFERENTIATION

In this section we describe Automatic Differentiation (AD), focusing on the so-called reverse mode, which will be used intensively. As we saw in section 2, it turns out that Optimal Control problems make an intensive use of derivatives of the form transposed-Jacobian-times-vector, or, equivalently when the function has a scalar result, the whole Jacobian row vector. We will show why the reverse mode of AD is the most efficient way to get these derivatives.

Then we shall discuss the time-memory tradeoffs that must be made to use reverse-mode AD on large industrial-size applications. We shall show how specific data-flow analyses on the program to be differentiated can help create an efficient differentiated program.

In general, it requires an AD tool to perform AD. Several AD tools propose the reverse mode. For the applications presented in this paper, we used the tool TAPENADE [11], which is developed and distributed by our research team. The data-flow analyses that we describe are implemented and tested inside TAPENADE.

4.1 Principles of reverse AD

Given a source program `P` that evaluates a function F that goes from input x to result $y = F(x)$, AD is able to create a new source program that computes *analytical* derivatives of F. In particular the reverse mode of AD creates a source program $\overline{\texttt{P}}$ that computes $F'^*(x) \cdot \overline{y}$ for any given vector $\overline{y}$. In the special case where y is scalar, if we just take $\overline{y}$ to be one, $\overline{\texttt{P}}$ returns the row vector $F'(x)$, i.e. the gradient.

To explain the principle of the reverse mode, let's suppose for the sake of simplicity that `P` is a simple list of elementary statements $I_k, k \in [1..p]$. Calling f_k the function implemented by I_k, the F computed by `P` is

$$F = f_p \circ f_{p-1} \circ \cdots \circ f_1 \ .$$

Using the chain rule, the Jacobian F' of F is:

$$\begin{aligned} F'(x) = \ & (f'_p \circ f_{p-1} \circ f_{p-2} \circ \cdots \circ f_1(x)) \\ & . (f'_{p-1} \circ f_{p-2} \circ \cdots \circ f_1(x)) \\ & . \ \cdots \\ & . (f'_1(x)) \ . \end{aligned} \tag{15}$$

Let us call for short $x_0 = x$ and $x_k = f_k(x_{k-1})$. The transposed-Jacobian-times-vector product that we need writes:

$$F'^*(x).\overline{y} = f'^*_1(x_0).f'^*_2(x_1).\ldots.f'^*_p(x_{p-1}).\overline{y} \tag{16}$$

The reverse differentiated program $\overline{\texttt{P}}$ will evaluate equation (16), for any given x and $\overline{y}$, from *right to left*, because matrix×vector products are much cheaper than matrix×matrix products. In theory, the computation cost of $\overline{\texttt{P}}$ is only a very small constant multiple of the cost of `P`.

In contrast, evaluating the expression in (16) from left to right would result in computing $F'^*(x)$ explicitly, and this has a cost which is proportional to the dimension of x. In our examples, this can be a large number. This explains why reverse AD is definitely the most efficient way to compute the derivatives needed for our Optimal Control problems.

Evaluating (16) from right to left results in the following structure of the reverse differentiated program:

$$\begin{aligned} & \overline{y}_{p-1} := f'^*_p(x_{p-1}).\overline{y} \\ & \ldots \\ & \overline{y}_{k-1} := f'^*_k(x_{k-1}).\overline{y}_k \\ & \ldots \\ & \overline{y}_0 \ := f'^*_1(x_0).\overline{y}_1 \\ & \texttt{return } \overline{y}_0 \end{aligned}$$

On this structure the main drawback of reverse AD becomes apparent: the intermediate values x_k are used in the *reverse of their computation order* in P. A typical way to handle this is to run P first, this time storing the intermediate values x_k. This defines the *forward sweep* $\overrightarrow{\texttt{P}}$, which must be run first. Then comes the *backward sweep* $\overleftarrow{\texttt{P}}$, which consists of the differentiated instructions above, with additional instructions inserted to progressively restore the intermediate values x_k.

The forward and backward sweeps interact using a stack, and we shall call *restoration* of a variable the couple of statements that push this variable to the stack in $\overrightarrow{\texttt{P}}$ and pop it from the stack in $\overleftarrow{\texttt{P}}$. Before each instruction I_k in $\overrightarrow{\texttt{P}}$, we consider the few variables that I_k may overwrite. For each such variable, if its present value is required in the derivatives of $I_1; \ldots; I_k$, then it must be restored. Detection of the variables required by the derivatives of instructions from I_1 to any I_k is called the *TBR (To Be Restored)* analysis, and is a standard data-flow analysis for AD.

This way, the stack grows reasonably slowly with the size, i.e. execution time, of `P`. Yet the growth is linear, and for a very large `P`, radical time-memory tradeoffs must come into play, which we discuss in the next section.

4.2 Data-flow analyses for time-memory tradeoffs

The general time-memory tradeoff is called *checkpointing*, illustrated on Fig. 2. The forward sweep

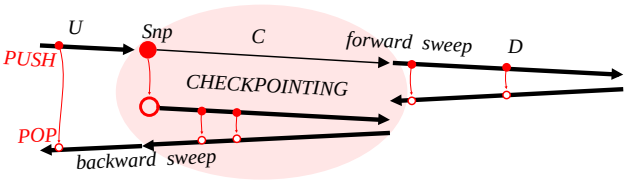

Fig.2: The Checkpointing time-memory tradeoff

goes from left to right, the backward sweep goes from right to left. Each I_k is vertically aligned with its

derivative I'_k. Suppose P is split in three successive fragments U, C, and D. Checkpointing C amounts to running C without any restoration push. When the backward sweep reaches back fragment C, the intermediate values are missing. To keep things going, C is run a second time, now like a real forward sweep with the push statements, and then the backward sweep can resume execution till the end.

Duplicated execution of C obliges us to save a sufficient number of variables, called a *Snapshot* (*Snp*). Snapshots are usually smaller than the total number of push performed by C. All in all a good choice of checkpoints, most probably nested, results in a stack size that grows only like the logarithm of the size of P, at the cost of repeated executions that make the execution time increase, by a factor which is also of the order of the logarithm of the size of P [9].

Because it is essential to keep the stack size low, we studied the checkpoint mechanism, looking for minimal *snapshots*. Let's go back to Fig. 2 and find what must be in the snapshot *Snp*. The goal is that the second execution of C runs exactly like the first. Using standard data-flow analysis terminology, a sufficient condition is that the **use** set of C is not overwritten between its two executions. In other words, execution of $\texttt{push}(Snp);\texttt{C};\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}};\texttt{pop}(Snp)$, must modify no variables in **use**(C). Introducing the **out** set of variables possibly modified by a piece of code, the constraint writes:

$$\mathbf{out}\left(\texttt{push}(Snp);\texttt{C};\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}};\texttt{pop}(Snp)\right)\cap\mathbf{use}(\texttt{C})=\emptyset \tag{17}$$

Classically, the **out** sets of successive code fragments accumulate. However, the push and pop pairs remove variables from the **out** sets. Therefore equation (17) rewrites as:

$$\left(\mathbf{out}(\texttt{C})\cup\mathbf{out}(\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}})\right)\setminus Snp\cap\mathbf{use}(\texttt{C})=\emptyset \tag{18}$$

And the smallest *Snp* that obeys this constraint is:

$$Snp=(\mathbf{out}(\texttt{C})\cup\mathbf{out}(\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}}))\cap\mathbf{use}(\texttt{C}) \tag{19}$$

Now, we observe that the **out** set of a forward-backward pair such as $\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}}$ depends on the set **req** of required variables imposed on it by the *TBR* analysis. Indeed, if variable v is added into **req**, then if D modifies v, $\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}}$ must restore it. In any case, v is removed from $\mathbf{out}(\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}})$. Therefore we have two options:

- **eager snapshot:** we keep the **req** set before D to the **req** before C, i.e. the variables required by $\overleftarrow{\texttt{U}}$ derivatives of U.
- **lazy snapshot:** we add to the **req** set before D all the variables in **use**(C), and then we know that

$$\mathbf{out}(\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}})\cap\mathbf{use}(\texttt{C})=\emptyset \tag{20}$$

and *Snp* is reduced to $\mathbf{out}(\texttt{C})\cap\mathbf{use}(\texttt{C})$, at the expense of more restorations inside $\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}}$.

Experimental measurements show that the lazy snapshot option generally performs better, although this depends on the code. Table 1 shows the effect of the two options on the computation of the gradient of a classical 2D Navier-Stokes solver. We observe an interesting 25% gain in memory for the lazy snapshot option. CPU time is also improved marginally, probably because less memory traffic also means less CPU time.

Snapshot:	**eager**	**lazy**
Memory (Mbytes)	248.1	184.7
CPU (seconds)	25.2	22.3

Table 1: Comparison of **eager** and **lazy** snapshot strategies on the gradient of a 2D Navier-Stokes solver

Whatever the option chosen, this definition of the snapshot correctly handles successive checkpoints. Suppose that the D program fragment is split again, to feature a second checkpoint C2. Suppose that C uses a variable v but doesn't modify it, whereas C2 uses and modifies v. v is not modified elsewhere. In other words:

$$v\in\mathbf{use}(\texttt{C});\quad v\notin\mathbf{out}(\texttt{C});\quad v\in\mathbf{use}(\texttt{C2});\quad v\in\mathbf{out}(\texttt{C2})$$

Equation (19) tells us that $v\in Snp(\texttt{C2})$. If we use eager snapshots, then with the help of a good "**out**" analysis, we find that $v\notin\mathbf{out}(\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}})$ because $v\in Snp(\texttt{C2})$, and thus $v\notin Snp(\texttt{C})$. On the other hand if we use lazy snapshots, $v\notin Snp(\texttt{C})$ simply because $Snp(\texttt{C})$ is now only $\mathbf{out}(\texttt{C})\cap\mathbf{use}(\texttt{C})$, even without the need for a good "**out**" analysis on $\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}}$.

This is particularly important for the derivatives of the assembly phases, in which the current state variables are in general used at several places to compute the residuals, and are only modified once, at the end of the (pseudo-)time step, to hold the next state.

5 RESOLUTION ALGORITHMS

Assume that, with the help of Automatic Differentiation applied to the assembly routines of the original simulation code, we have obtained the assembly routines for the different ingredients of the KKT system (5). Specifically, we now have routines that, given

a γ and a W, compute efficiently

$$\frac{\partial J}{\partial W}(\gamma, W) \quad \text{and} \quad \frac{\partial J}{\partial \gamma}(\gamma, W) \ ,$$

and given an additional argument Π,

$$\left(\frac{\partial \Psi}{\partial W}(\gamma, W)\right)^* \cdot \Pi \quad \text{and} \quad \left(\frac{\partial \Psi}{\partial \gamma}(\gamma, W)\right)^* \cdot \Pi \ .$$

In section 5.1 we will discuss the manners to obtain the gradient $j'(\gamma)$. We can then use an optimization algorithm to minimize the functional j. This can be any"from the shelf" optimization software, which will iteratively call our routines for $j(\gamma)$ and $j'(\gamma)$. Sections 5.2 and 5.2 review two important improvements for a better efficiency.

5.1 Computing the gradient of the objective functional

Our goal is now to compute the gradient $j'(\gamma)$. We will apply a procedure that follows from system (5) line by line:

1. first solve the state equations, yielding W
2. then solve the adjoint state equations, yielding Π
3. finally assemble the residual of the optimality equations, yielding $j'(\gamma)$.

In this section we do not address the topmost optimization loop that reduces $j'(\gamma)$ to zero.

Resolution of the state equations (step 1) is of course already available in the initial simulation code. We assume, as it is generally the case, that this resolution uses a matrix-free iterative solver which repeatedly calls the assembly of the state residual Ψ. For example, it can be a pseudo-unsteady explicit time-stepping or a GMRES quasi-Newton iteration.

It is important to understand why we choose to go through step 2, i.e. explicitly solve for the adjoint state Π. Why don't we instead ask directly the AD tool to reverse-differentiate the routine that computes $j(\gamma)$? This would return the gradient $j'(\gamma)$. In fact, this has been done before with success, e.g. in [13]. But this straightforward approach has several severe drawbacks, that we shall put in two categories for discussion.

The first category of drawbacks is about efficiency. The differentiated code uses an enormous amount of memory, related to the reverse mode principle sketched in section 4.1. Essentially, each of the non-converged iterates of the state W need be stored. In the present state of the art, even with data-flow analyses such as those in section 4.2, radical manual post-processing of the differentiated code is necessary. Moreover, the systematic approach differentiates computations that are in fact irrelevant, such as evaluation of the time-step, and this hampers efficiency, requiring manual post-processing. The last drawback in this category is the fact that we cannot expect the derivatives to converge at the same rate as W. In other words, it is questionable to perform the same number of iterations to converge on the derivatives during the backward sweep, than to converge on W during the forward sweep. Unfortunately, this is exactly what straightforward AD does.

One can think of an elegant way to overcome these drawbacks, which we might call "fixpoint-conscious-AD". We could modify the reverse-AD model for fixpoint iterations so that none of the iterates W_k of W is stored, except the final W_N, which is converged up to the prescribed ε. The backward sweep of the differentiated program would repeatedly use the values from W_N, even when reversing the computations of another time step $k \neq N$. This clearly solves the memory question. Moreover, this allows the backwards sweep to perform a different number of iterations, and in particular to use a specific stopping criterion for the backward iterative loop, involving convergence of the derivatives themselves. Fixpoint-conscious AD could even be automated inside AD tools, freeing us from the tedious and error-prone post-processing task. Another approach in [10], computes W and the derivatives simultaneously, thus saving storage.

The second category of drawbacks comes from the iteration algorithm itself. If explicit pseudo-time stepping is used, the state iteration is a linear fixed point, and the transposed iteration performed by the differentiated code will also be stable and converging. On the other hand, if the state iteration is far from linear, typically because of line-searches or orthonormalization, then there is no guarantee that the differentiated (i.e. transposed) iteration is stable nor convergent, let alone efficient. Finally, in the case of non-linear iterations, there is very little mathematical insight of the consequences of freezing the state to W_N.

therefore we recommend in general not to differentiate the fixed point iteration itself. We recommend instead to re-use the iteration algorithm, possibly changing the pre-conditioner which has to be simply transposed. In [3], this strategy is applied, using a first-order simplified Jacobian as a pre-conditioner.

5.2 One-shot optimization

Modern finite-dimensional optimization methods relying on adjoints are issued from the Sequential Quadratic Programming (SQP) methodology. A popular prototype is the Byrd-Omojokun algorithm, see [14]. This algorithm in its basic form assumes that

the resolution of the different linearizations of state systems (Newton iteration of state and solution of adjoint) are not expensive However, this assumption is not valid in Optimal Shape Design. This fact has lead some authors to attack the problem using *one-shot* (or *progressive*, or *simultaneous*) algorithms [15, 6], which are based in the following two principles:

- Use discipline-specific iterative, maybe nonlinear solvers (for example, pseudo unsteady solvers for Fluid Mechanics) for state and co-state.
- Iterate simultaneously the three equations of the KKT system.

The cost by iteration of these algorithms is much lower with a comparable convergence. Assuming they converge in a number of iterations independent from the discretization fineness, it follows that they are potentially able to reach optimal complexity, in the sense that the solution costs k times the resolution of the state equation, k being independent from the number of control parameters. The efficiency of this method is shown for example in [4, 7]. But the question of independence from the discretization fineness remains to be addressed.

5.3 Multi-level Optimization

Large scale problems coming from Partial Differential Equations generally result in a conditioning which is poor and getting poorer as the number of degrees of freedom grows. The reason for this can be found either through a direct analysis of discrete eigenvalues as the number of unknowns increases, or through an analysis of the continuous -functional- problem and the continuous version of the algorithm. Shape design problem possess a continuous formulation and the corresponding sensitivity has been analyzed by Hadamard about one century ago. It appear that the gradient is a non-bounded operator. Its usage leads to ill-conditioned iterations. This problem can be tackled by applying an **additive multilevel pre-conditioner** B, which is applied to the iterative procedure:

$$\gamma^{n+1} = \gamma^n - \rho B g_{L^2}, \qquad (21)$$

At each iteration n the correction coming from the optimization process is updated. The correction consists of a step-length factor ρ multiplying the preconditioned gradient. The self-adjoint invertible operator B is chosen in order to recover the degree of regularity lost by the L^2 gradient g_{L^2}. We refer to [2, 1, 7] for theoretical aspects and applications.

6 SOME RESULTS

The application to second derivatives for robust optimization is not yet implemented and no results can be shown for it.

6.1 Shape design

To begin with, here are the results of the optimal shape design example described in section 3.1.

Our optimal control model takes as input the simulation code that computes W. This code uses an upwind Euler solver, running on an unstructured mesh made of 981822 tetrahedra. During optimization, the shape is changed by moving the nodes on the boundary of the mesh along normals to that boundary, but this is implemented by a *transpiration condition* in order to avoid costly remeshings.

Although we target only optimization of the shape of the wings, the flow W, the adjoint Π, and the gradient $j'(\gamma)$ are computed on the complete geometry. Fig. 3 shows the resulting gradient of the objective functional (6) on the skin, at first optimization cycle. Darker colors indicate the places where moving

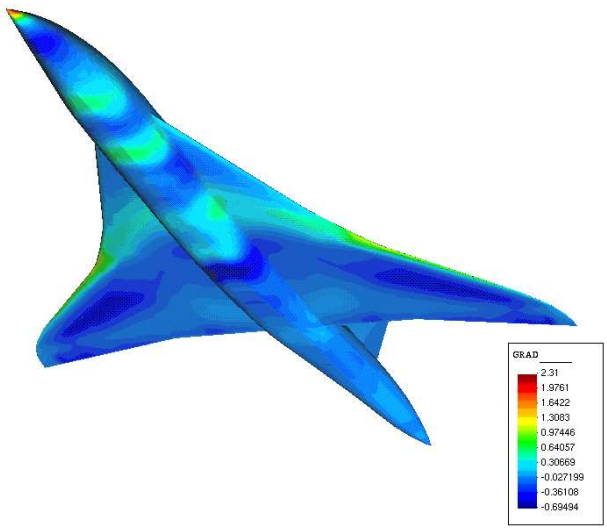

Fig.3: Gradient of the Objective Functional on the skin

the shape along the normal to the skin improves the objective functional more strongly.

Fig. 4 shows the evolution of the pressure in box Ω_B after 8 optimization cycles. We observe that the shock produced by the outboard part of the wings is dampened. However, close to the fuselage, the pressure peak has slightly increased after the optimization process. This increase is tolerable, compared to the reduction obtained on the end of the wings. Physically, this is coherent with the fact that the Mach cone is 34° wide for the speed considered, and therefore only the outboard part of the wings cuts through this cone, so that the sharpest pressure gradient is produced ahead of the outboard portion of the wing.

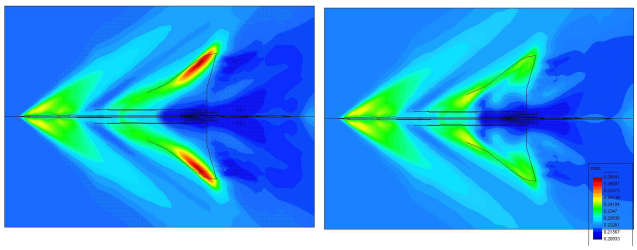

Fig.4: Pressure in a plane below the aircraft. *Left:original. Right:optimized.*

As far as performance is concerned, we observe that computation of the gradient $j'(\gamma)$ is about four times as slow as the computation of the state W. This is quite reasonable indeed, and is consistent with the results shown in [3, 8]. It is true that the ratio between the assembly of the state residual and the assembly of the adjoint residual is more of the order of seven, which is about the average ratio for codes generated by reverse AD. But this slowdown is dampened by the fact that each assembly of the state residual is followed by the pre-conditioned iteration algorithm, which takes about the same time, and also the same time for the adjoint state Π. One step of the state resolution costs therefore 2, whereas one step of the adjoint resolution costs 8, and therefore the overall factor of 4.

6.2 Numerical error control by mesh optimization

For this numerical study of our mesh optimization method of section 3.3, we suppose we know the exact solution of the numerical problem. On the domain Ω equal to the unit square, we choose

$$u(x,y) \quad = \quad (x^2 - x)(y^2 - y) \ . \qquad (22)$$

The objective functional to minimize is simply the norm of our approximation error model W. Several simplifications are made on the function g, which we will not detail here.

The gradient of the discrete functional is obtained by an adjoint method and developed with the help of the TAPENADE AD tool [11]. A nonlinear conjugate gradient method converges to a (possibly local) minimum in a hundred iterations. In these preliminary experiments, we are only interested in finding the optimal mesh density d: we do not regenerate a new mesh from d, as should be done in real situations. The density d is parameterized by its values on a Cartesian grid of 11×11 vertices.

All figures show the values of the slice $y = 0.5$.

We start with a uniform node density. At the end of the Optimization cycle, the objective functional has decreased by 50%, and the resulting mesh density d is shown on Fig. 5. It increases in the center and decreases on the boundary. As a result of this optimization, the approximation error W is reduced at every point of Ω, as shown on Fig. 6.

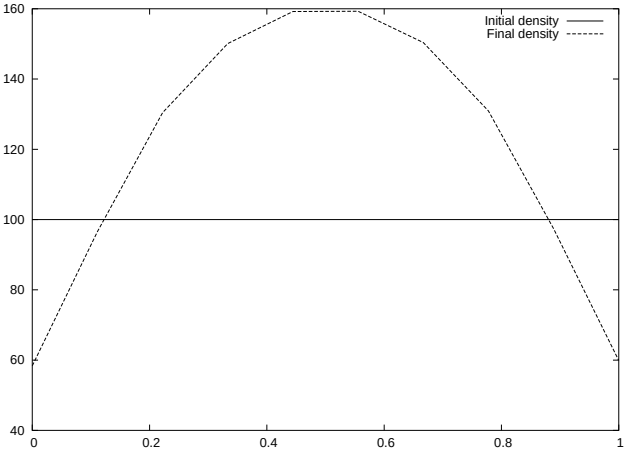

Fig.5: Minimization of L^2 error over the whole domain: initial (line) and final (dashes) mesh densities d

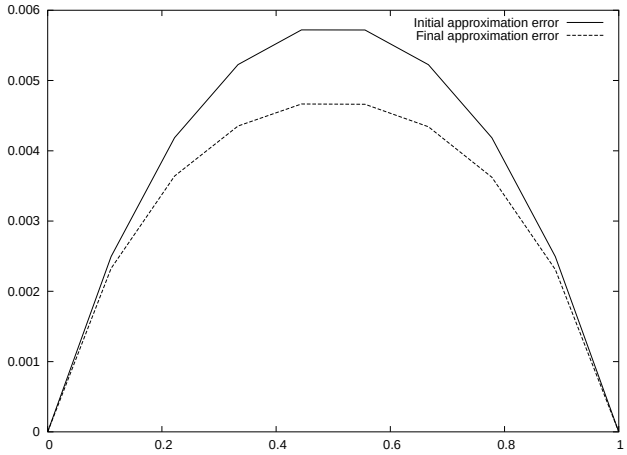

Fig.6: Minimization of L^2 error over the whole domain: initial (line) and final (dashes) approximation errors W

A particular interest of this adjoint based error formulation with respect to local truncation error is that we can modify the objective functional, e.g. minimize the approximation error measured on a specified subset of Ω. Let us for example restrict the objective functional to the integral over the right half part Ω_R of the domain ($x > 0.5$).

If we worked only with a local truncation model, then we would obtain an optimal mesh without nodes on the left part of the domain. This is good for the

interpolation error over Ω_R, but not so good for the approximation error in Ω_R: for the approximation error, values over the mesh are strongly correlated. The optimal mesh density given by our algorithm on Fig. 7 shows only a slight concentration of nodes in the right part of computational domain. Accordingly, the approximation error itself, shown on Fig. 8 is noticeably decreased on the right side, and marginally increased on the left.

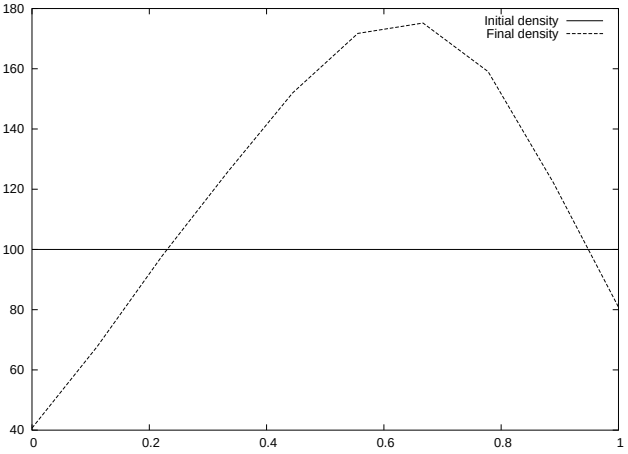

Fig.7: Minimization of L^2 error over right half-domain: initial (line) and final (dashes) mesh densities d

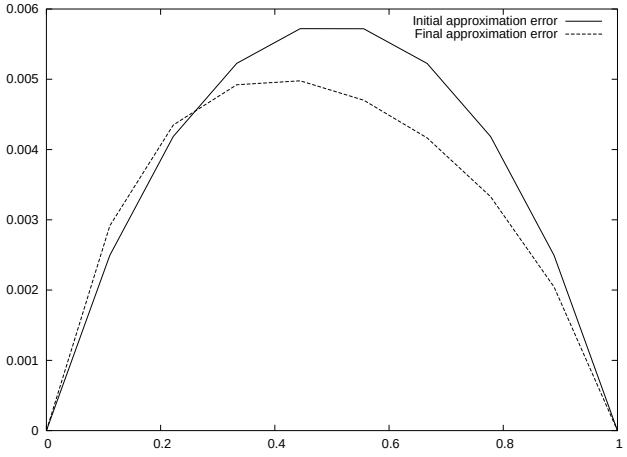

Fig.8: Minimization of L^2 error over right half-domain: initial (line) and final (dashes) approximation errors W

7 CONCLUSION

This paper addresses adjoint-based optimization. This topic concerns a large range of applications in mechanical simulation, optimal control, and optimal design.

In this paper, we present one possible strategy which, starting from simulation tools, derives an adjoint-based optimization suite. This strategy relies in part on Automatic Differentiation to build the assembly routines for adjoint states, and in part on modern optimization techniques. Another possible strategy is presented in [8].

This strategy is illustrated on a set of examples coming from different fields of numerical science.

Adjoint-based methods are particularly interesting because they compute gradients in a time which is independent of the number of control parameters, unlike other methods based on divided differences or even evolutionary approaches. The present strategy advocates other techniques to gain efficiency, by using one-shot or multi-level optimization algorithms which reduce dramatically the number of state and adjoint state evaluations required to minimize the objective.

Although it partly relies on hand-coding, this strategy has the interest of re-using many parts of the given simulation code: the solvers and pre-conditioners used to solve the state equations are re-used to solve the adjoint equations.

The adjoint assembly routines are built with Automatic Differentiation, which amounts in a sense to code re-use. In particular, a complex assembly routine, e.g. higher-order approximation, results in an adjoint assembly routine of the same approximation order at a very low programming cost.

REFERENCES

[1] F. Courty and A. Dervieux. Multilevel functional Preconditioning for shape optimisation. submitted to *Int. Journal on CFD*, 2005.

[2] F. Courty and A. Dervieux. *A SQP-like one-shot algorithm for optimal shape design*. Springer, 2005. to appear.

[3] F. Courty, A. Dervieux, B. Koobus, and L. Hascoet. Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. *Optimization Methods and Software*, 18(5):615–627, 2003.

[4] F. Courty, T. Roy, B. Koobus, and A. Dervieux. Mesh adaptation by Optimal Control of a continuous model. Research Report 5585, INRIA, 2005.

[5] F. Courty, T. Roy, B. Koobus, M. Vázquez, and A. Dervieux. Error analysis for P1-exact schemes. In *Finite Element for Flow Problems, Swansea, UK*, 2005.

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

[7] A. Dervieux, F. Courty, T. Roy, M. Vázquez, and B. Koobus. Optimization loops for shape and error control. In *PROMUVAL Short Course on*

Multidisciplinary Modelling, Simulation and Validation in Aeronautics, Barcelona, june 28-29, 2004. CIMNE, 2004. extended version INRIA Research Report 5413.

[8] M. Giles. Using Automatic Differentiation for adjoint CFD code development. In *Post-SAROD workshop, Bangalore, India*, 2005.

[9] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[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 and V. Pascual. Tapenade 2.1 user's guide. Technical Report 0300, INRIA, 2004.

[12] L. Hascoet, M. Vázquez, and A. Dervieux. Automatic differentiation for optimum design, applied to sonic boom reduction. In V.Kumar et al., editor, *Proceedings of the International Conference on Computational Science and its Applications, ICCSA'03, Montreal, Canada*, pages 85–94. LNCS 2668, Springer, 2003.

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

[14] J. Nocedal and S.-J. Wright. *Numerical Optimization*. Springer, Series in Operations Research, 1999.

[15] 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.

[16] M. Vázquez, A. Dervieux, and B. Koobus. Aerodynamical and sonic boom optimization of a supersonic aircraft. Research report 4520, INRIA, 2002.