Automatic Differentiation for Optimum Design, Applied to Sonic Boom Reduction

Laurent Hascoët[1], Mariano Vázquez[1,2], and Alain Dervieux[1]

[1] Projet Tropics, INRIA, France
[2] GridSystems, Mallorca, Spain

Abstract. We propose a methodology for adjoint-based optimum design that combines hand-coding with Automatic Differentiation in the reverse mode, therefore managing to keep the memory cost acceptable. This methodology also involves improvements to the reverse mode differentiation, based on dataflow information on the original program. We validate the method on a real-size 3D optimum design problem: reducing the sonic boom under a supersonic aircraft.

1 Introduction

This paper presents a methodology for adjoint-based optimization applied to real-size problems in optimum design. We advocate the use of Automatic Differentiation (AD) [1, 2] in the *reverse mode*, to produce derivatives in a safe, automated, and reproducible way. However, we do not advocate using AD on complete resolution algorithms, because the result must then be reorganized by hand to achieve efficiency. Alternately, we propose to go back up to the mathematical *adjoint equations*, and write by hand a new solver for those. Inside these equations, we identify derivative terms that we will generate by reverse-mode Automatic Differentiation, rather than discretize and implement them. This methodology thus combines AD on large parts of the original flow solver (flow residual assembly and cost function), together with hand implementation of a special solver for the adjoint.

We also propose important improvements to the code generated by AD, that use dataflow information to reduce the memory needed by the reverse-differentiated code, down to an acceptable size. We consider gather-scatter loops, first described in [3], and make use of *in-out* analysis to reduce the memory used by the *checkpoint* mechanism. These improvements are progressively being included into the AD tool TAPENADE [4]. We illustrate this methodology on a concrete shape optimal design problem in aeronautics. Reducing the sonic boom on the ground is one of the challenges to be solved before a new generation of supersonic aircraft can appear. We concentrate on reducing the downwards sonic boom emission, preserving other aerodynamic properties, following a method first proposed in [5].

This paper is organized as follows. Section 2 describes the continuous model for the "sonic boom" application problem. Section 3 introduces our methodology

that combines Automatic Differentiation and hand-coding of the adjoint solver. Section 4 presents the numerical results for the actual optimization of the Sonic Boom for a supersonic business jet. Section 5 gives the basics of reverse AD, describes the validation process, and focuses on the dataflow-based improvements that reduce memory consumption. We conclude in Sect. 6.

2 Continuous Model and Cost Functional

We must select a model for the fluid flow around the plane. We choose the inviscid Euler equations for perfect diatomic gas, together with *transpiration* boundary conditions on the skin of the plane. Here's the reason: The optimization cycle usually causes repeated modifications of the shape of the plane, and therefore of the mesh itself. However, when the modification of the mesh is small, this costly remeshing can be avoided by introducing *transpiration* conditions [6, 7]: the computational domain and the mesh geometry thus remain fixed.

Let us now describe the model that we use to estimate the intensity of the sonic boom on the ground. Any solid moving at supersonic speed develops a system of shock waves. For simple solids, there are mainly two shock waves, a bow wave at the front, and a tail wave at the rear. For complex shapes such as airplanes, the shock waves are many, but as they get farther from the object, they coalesce into a two-shock system as shown in Fig. 1. What we minimize is a measure of the intensity of the shocks, described for example in [8].

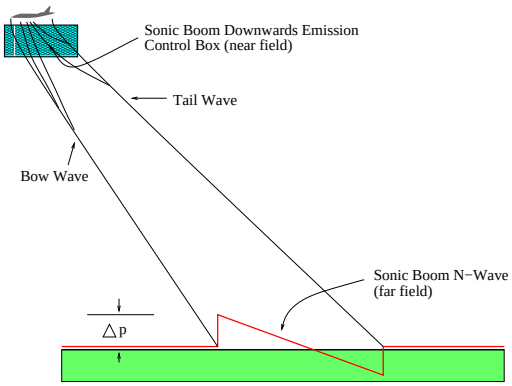

Fig. 1. The sonic boom: shock wave patterns on the near and far fields

To find a geometry of the supersonic plane that minimizes the sonic boom on the ground, we need a model that relates this geometry to the sonic boom. Unfortunately, a direct evaluation, from the 3D Euler equations, of the shock signature on the ground (the *far field*) is still computationally out of reach. What we can indeed compute is the flow in a given box just below the plane (the *near field*). See [8, 9] about the issue of propagation from the near field to the far field, which is usually done in 2D, in a vertical plane below the flight direction.

We propose a simplified optimization process. We assume that the source of the shock on the ground is the shock pattern in the near field. Therefore, for a flow W, itself depending on the shape parameters γ, what we minimize is the integral of the squared gradient of the pressure $p(W)$ on the near field. For regularity reasons, this squared gradient is better integrated on the *volume* of the near field box Ω^B. This choice is discussed in [5]. Since we also want to keep the lift $L(W)$ and drag $D(W)$ close to prescribed values L_0 and D_0, the cost functional j is given by (1), where constants α_1, α_2 and α_3 set the relative weights of the three criteria.

$$j(\gamma) = \alpha_2(L(W)-L_0)^2 + \alpha_1(D(W)-D_0)^2 + \alpha_3 \int_{\Omega^B} (\nabla p(W))^2 \, dV \qquad (1)$$

This optimization criterion, although computed only on the near field, is general and involves most aspects of the sonic boom on the ground. We believe that it can indicate new aerodynamical shapes, that retain good flight performance. Other works on sonic boom reduction attack directly the complete aircraft problem. We propose instead to focus on *isolated* parts of the plane, e.g. the wings, gaining insight about what is really producing the sonic boom, as a preliminary step to the full optimization of the plane.

3 Evaluation of the Gradient on the Discretized Model

The discrete flow model is based on a mixed finite-volume/finite-element approximation. It applies MUSCL-type upwinding on unstructured tetrahedra. The accuracy is second order. Steady solutions are obtained by an implicit time stepping. As we come closer to the steady solution, this progressively becomes a pseudo-Newton iteration with larger time steps. At each time step, solution advancement is stable thanks to a simplified Jacobian, for which storage is not too large and iterative resolution rather easy. We refer to [10] for details on the numerical scheme.

There are several methods to evaluate the gradients that will drive the optimization process.

- One can compute them with finite differences: this requires many evaluations of the flow solver program `P`, for slightly different values of the shape parameters. This is costly and imprecise and therefore should be avoided.
- Alternatively, one can use *Automatic Differentiation* (AD) in its so-called *reverse mode*: given a program `P` that discretizes and computes a function F, AD in the reverse mode creates a new program $\overline{\mathtt{P}}$ that computes the transposed Jacobian of F multiplied by a given vector. In the case of an optimization process, F has a single output (the cost), and therefore the program $\overline{\mathtt{P}}$ computes exactly the gradient of F. The advantage of AD is that we obtain exact, analytical gradient of the discretized F, at a computational cost which is a small multiple of the cost of `P` itself. Section 5 gives more details. The drawback is that brute force *reverse AD* requires a lot of memory

space, grossly proportional to the execution cost of `P`. This is far too much for our large application. Therefore we cannot use AD as a black box tool on the whole program that goes from the shape parameters to the cost.

– We shall use a third method, which starts back from the initial mathematical equations to derive a so-called *adjoint system*, whose solution is the gradient. This method comes from control theory, and was made popular in aerodynamical design by Jameson in the classical reference [11]. We then write a new program that solves this adjoint system. However we notice that parts of this program can actually be generated by *reverse AD* on parts of the flow solver program `P`. This method thus amounts to a tradeoff where reverse AD is applied to large *parts* of `P`, requiring a manageable memory space, while the higher level of the algorithm, namely the iterative resolution of a linear system, is written by hand using knowledge of the mathematical problem. The rest of this section gives more details on this third method.

Basically, we are considering a minimization problem under a particular additional constraint: the flow equation $\Psi(\gamma, W(\gamma)) = 0$, which expresses the dependence of the flow field $W(\gamma)$ on the shape γ. We thus want to find the γ_0 that minimizes the cost functional $j(\gamma) = J(\gamma, W(\gamma))$ (*cf* equation (1)), under the constraint $\Psi(\gamma, W(\gamma)) = 0$. Here, this constraint is the compressible Euler equations, solved in a domain Ω_γ parametrized by γ. This minimization problem is solved using Lagrange multipliers. The problem's Lagrangian is

$$L(W, \gamma, \Pi) \;=\; J(\gamma, W) + \langle \Psi(\gamma, W), \Pi \rangle, \tag{2}$$

where Π is a generalized Lagrange multiplier, and $\langle\ ,\ \rangle$ is a suitable scalar product. Then the gradient $j'(\gamma)$ is found by solving

$$\begin{aligned} \Psi(\gamma, W(\gamma)) \;&=\; 0 \\ \nabla_W J(\gamma, W(\gamma)) \;-\; (\nabla_W \Psi(\gamma, W(\gamma)))^* \; \Pi(\gamma) \;&=\; 0 \\ j'(\gamma) \;&=\; \nabla_\gamma J(\gamma, W(\gamma)) \;-\; (\nabla_\gamma \Psi(\gamma, W(\gamma)))^* \; \Pi(\gamma). \end{aligned} \tag{3}$$

The first line gives $W(\gamma)$. The second line is the so-called *adjoint flow equation*, and gives $\Pi(\gamma)$. The last line gives the *gradient* $j'(\gamma)$ using $\Pi(\gamma)$ and $W(\gamma)$. This gradient will be used to update iteratively the former γ.

We then remark that, if we isolate the subprogram `Psi` of the flow solver program `P` that computes $\Psi(\gamma, W(\gamma))$, the reverse mode of AD can build automatically a new subprogram $\overline{\texttt{Psi}}_W$. This new subprogram, given any vector Π, returns the product $(\nabla_W \Psi(\gamma, W(\gamma)))^* \; \Pi$. Differentiation with respect to γ instead of W gives another subprogram $\overline{\texttt{Psi}}_\gamma$ that for any Π returns $(\nabla_\gamma \Psi(\gamma, W(\gamma)))^* \; \Pi$. Similarly, if we isolate the subprogram `J` that computes the cost function $J(\gamma, W(\gamma))$, the reverse mode of AD automatically builds subprograms $\overline{\texttt{J}}_W$ and $\overline{\texttt{J}}_\gamma$ that respectively compute $\nabla_W J(\gamma, W(\gamma))$ and $\nabla_\gamma J(\gamma, W(\gamma))$.

With these subroutines generated, what remains to be done by hand to get $j'(\gamma)$ is the solver that solves the *adjoint flow equation* for Π. Notice that AD does not give us the matrix $(\nabla_W \Psi(\gamma, W(\gamma)))^*$ explicitly. Anyway this (2^{nd}-order) Jacobian matrix, although sparse, is too large for efficient storage and

use. Therefore, we must build a *matrix-free* linear solver. Fortunately, we just need to modify the algorithm developed for the flow solver P. P uses a simplified (1^{st}-order) Jacobian for preconditioning the pseudo-Newton time advancing. This matrix is stored. We just transpose this simplified Jacobian and reuse its Gauss-Seidel solver to build a preconditioned fixed-point iteration, that solves the adjoint flow equation. This method is detailed for a 2D application in [12]. We validated the resulting gradient by direct comparison with divided differences of the cost function. The relative error is about 10^{-6}.

The overall optimization process is made of two nested loops. The outer loop evaluates the gradient, using an adjoint state as described above, then calls the inner loop which does a 1D search to get the steepest descent parameter, and finally updates the control parameters, i.e. the transpiration parameters. Algorithmic aspects of this application are presented in the companion paper [5].

4 Results on a Supersonic Business Jet

We applied our optimization program on the shape of a Supersonic Business Jet, currently under development at Dassault Aviation. The mesh consists of 173526 nodes and 981822 tetrahedra (for half of the aircraft). The inflow Mach number is 1.8 and the angle of attack is 3°. We target optimization of the wings of the aircraft only. Even then, the flow, the adjoint state, and the gradient $j'(\gamma)$ are computed taking into account the complete aircraft geometry. Figure 2 shows the gradient of our "sonic boom" cost functional (1) on the skin on the complete aircraft. Darker colors indicate places where modifying the shape strongly improves the sonic boom.

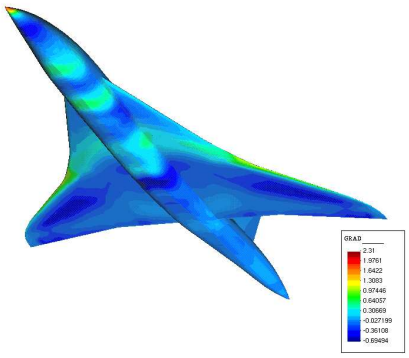

Fig. 2. Supersonic Business Jet: Gradient of the Cost Functional on the skin

The simplified wings provided by the constructor are horizontally symmetrical, with two successive sweep angles (17° and 38°). At the considered speed, the Mach cone is 34° wide. Therefore, only the outboard part of the wings cuts through this cone, and the sharpest pressure gradient will be produced ahead of the outboard portion of the wing. This can be checked on Fig. 2.

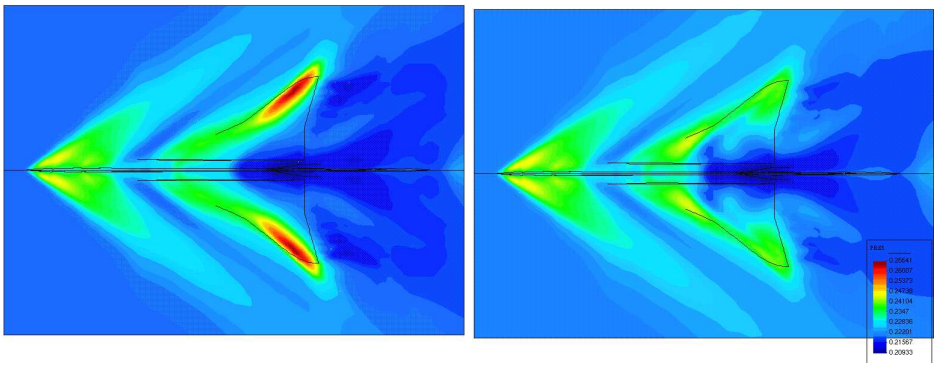

Fig. 3. Supersonic Business Jet: Pressure distribution in a plane below the aircraft. *Left: original. Right: optimized.*

Figure 3 shows the evolution of the pressure on the near field, after 8 iterations. We observe that the shock produced by the outboard part of the wings is dampened. However, within the Mach cone, close to the fuselage, the pressure peak has slightly increased after the optimization. This increase is tolerable, compared to the reduction obtained on the end of the wings; see [5] describing how this shape optimization improves the actual sonic boom on the ground.

5 A Focus on Application of Automatic Differentiation

In this section, we shall focus on the actual utilization of an Automatic Differentiation (AD) tool. Section 3 left us with the need for the four derivative subprograms $\overline{\texttt{Psi}}_W$, $\overline{\texttt{Psi}}_\gamma$, $\overline{\texttt{J}}_W$, and $\overline{\texttt{J}}_\gamma$, required during the process of computing the complete gradient $j'(\gamma)$. For example subprogram $\overline{\texttt{Psi}}_W$, given any γ and any vector $\overline{\Psi}$, must compute the product of the transposed Jacobian $(\nabla_W \Psi(\gamma, W(\gamma)))^*$ by $\overline{\Psi}$.

We build these four subprograms using the TAPENADE AD tool, developed by our team [4]. An AD tool uses the source of the program that computes $\Psi(\gamma, W(\gamma))$, and identifies this program with a composition of mathematical functions, one per run-time instruction. The derivatives are then computed *analytically*. Calling $I_k, k \in [1..p]$ the sequence of instructions executed at run-time, each of them implementing an elementary function f_k, the function f computed by the program is:

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

Calling x and y, respectively, the (multi-dimensional) input and output of f, the chain rule gives us the Jacobian of f:

$$\begin{aligned} f'(x) = \ & (f'_p \circ f_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x)) \\ & . \ (f'_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x)) \\ & . \ \ldots \\ & . \ (f'_1(x)) \ . \end{aligned} \tag{4}$$

Let us call for short $x_0 = x$ and $x_k = f_k(x_{k-1})$. The so-called *reverse* mode of AD aims at computing the product of the transposition of f' by a given weight vector $\overline{y}$. Therefore, we transpose equation (4) to get:

$$f'^*(x).\overline{y} = f_1'^*(x_0).f_2'^*(x_1).\ldots.f_p'^*(x_{p-1}).\overline{y}$$

Because matrix×vector products are so much cheaper than matrix×matrix products, the program that computes $f'^*(x).\overline{y}$ runs in the following order:

$$\begin{array}{l}\overline{y}_{p-1} = f_p'^*(x_{p-1}).\overline{y}\\ \ldots\\ \overline{y}_{k-1} = f_k'^*(x_{k-1}).\overline{y}_k\\ \ldots\\ \texttt{return } \overline{y}_0\end{array}$$

This turns out to make a very efficient program, at least theoretically. However, we observe that the x_k are required in the *inverse* of their normal computation order. TAPENADE handles this by storing the needed x_k into a stack, during an instrumented execution of the original program, called the *forward sweep*. Then the reverse-differentiated program computes $\overline{y}_0$ as shown above (*reverse sweep*), progressively popping the x_k from the stack.

This stack is the bottleneck of reverse AD. On large programs, we will need storage/recomputation tradeoffs to keep it small enough. In particular, this stack is one reason to reject brute force reverse AD to get the gradient $j'(\gamma)$: this stack would contain all intermediate values for all pseudo-time steps! In contrast, our strategy uses subprogram $\overline{\texttt{Psi}}_W$ which runs for just *one* pseudo-time step, therefore dramatically reducing the stack.

Now that we know how TAPENADE produces the four subprograms $\overline{\texttt{Psi}}_W$, $\overline{\texttt{Psi}}_\gamma$, $\overline{\texttt{J}}_W$, and $\overline{\texttt{J}}_\gamma$, we want to validate the derivatives. The usual process is to validate the tangent derivatives (Jacobian times vector) with respect to divided differences, and to validate the reverse derivatives with respect to the tangent derivatives, using the "dot-product" test. More precisely, given a validated tangent derivative code, which computes $\dot{y} = f'(x).\dot{x}$ for a given $\dot{x}$, we call the reverse mode to compute $\overline{x} = f'(x)^*.\dot{y}$. The dot-product test just checks that

$$\langle \overline{x}\ ,\ \dot{x}\rangle = \langle f'(x)^*.\dot{y}\ ,\ \dot{x}\rangle = \langle \dot{y}\ ,\ f'(x).\dot{x}\rangle = \langle \dot{y}\ ,\ \dot{y}\rangle\ .$$

Unfortunately, these are "yes-no" tests, that do not help much when things go wrong. To improve things, we developed two test libraries: (i) a library that runs the divided differences test at any selected place in the program (helping to find the place where the tangent code goes wrong) and (ii) a library that runs the dot-product test between any two given points in the program (helping to find the place where the reverse code goes wrong). These libraries proved useful, for there was actually a problem with one reverse-differentiated program, that we could find and fix in a matter of days. Then the validation tests for a representative case gave excellent agreement:

Squared Divided Differences	$\langle \dot{y}\ ,\ \dot{y}\rangle$	$\langle \overline{x}\ ,\ \dot{x}\rangle$
3301.12804	3301.12817	3301.12817

After validation of the derivatives there is the question of efficiency. Again, the bottleneck is the memory size of the stack that keeps track of intermediate values. This size is now proportional to the mesh size. It is worth noting that this stack grows and shrinks repeatedly during any single execution of, say, $\overline{\texttt{Psi}}_W$. This is because of the *checkpointing* mechanism, that trades duplicate execution for stack size. Precisely, checkpointing a fragment of the program means the following: (i) during the forward sweep, do *not* push intermediate values onto the stack when inside the fragment. (ii) during the reverse sweep, just before reaching the reverse of the fragment, run it forwards *again*, this time pushing the values, so that the reverse sweep can use them.

Of course, this requires that all variables needed to run the fragment be stored on the stack before the first execution, and popped before the second execution. These variables form what we call a *snapshot*. In TAPENADE, by default, these fragments are subroutine calls. This is far from optimal, but easy to implement. For example `Psi` calls subroutines `A`, `B`, and `C`, and Fig. 4 shows the call tree of $\overline{\texttt{Psi}}$. The stack grows when a snapshot is taken, and during forward sweeps. It shrinks when a snapshot is used, and during reverse sweeps. Therefore the size of the stack reaches four local maxima, shown on Fig. 4.

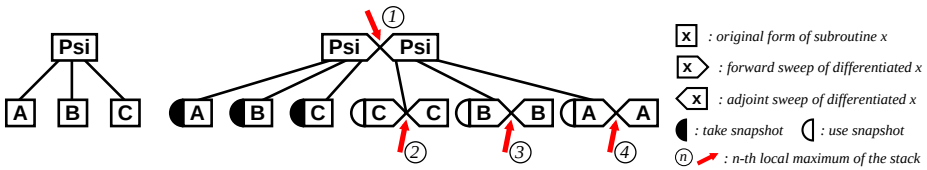

Fig. 4. Call Trees of original and reverse-differentiated `Psi`

We identified two improvements to reduce the stack. Their automation inside TAPENADE is in progress. One uses *in-out* information to reduce the size of snapshots, taking only variables that not only are used in the checkpoint, but *also* are possibly modified before the second execution of the checkpoint. The other observes that many loops in subroutines `A`, `B`, and `C`, are gather-scatter loops operating on mesh elements. These loops have *independent iterations*. In this case, thanks to data-flow considerations described in [3], we can improve the reverse-differentiated program as shown in Fig. 5, so that each *forward sweep* of

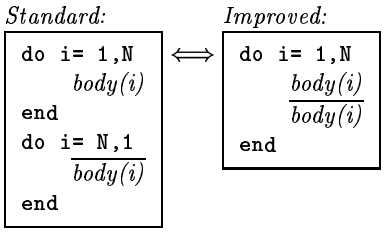

Fig. 5. Equivalent transformation of a reverse-differentiated gather-scatter Loop

an iteration of the loop *body* is immediately followed by the corresponding *reverse sweep* $\overline{body}$. This reduces the size of the stack by a factor N, size of the loop and of the mesh.

Table 1 shows the four local maxima of the stack size, at the locations defined in Fig. 4, for various combinations of the two improvements above. Notice that *both* improvements must be applied, otherwise the global maximum remains too high. This test was done on a reduced mesh of 2200 nodes. In proportion, each node in the mesh consumes 58 `REAL*8`'s in stack size, which seems acceptable.

Stack local maximum #	①	②	③	④
No modification:	12.40	12.37	13.60	9.66
Only snapshot reduction:	1.02	0.85	9.70	9.33
Only loops improvement:	12.38	7.98	4.10	0.02
Both improvements:	1.02	0.61	0.22	0.02

Table 1. Influence of improvements on the Maximal Size (Mbytes) of the Stack

6 Conclusion

We have presented a new methodology for application of reverse Automatic Differentiation to Optimal Control, and more precisely to the adjoint-based optimization of aerodynamic supersonic shapes. We emphasize two aspects of the utilization of AD. First, we advocate using AD on carefully selected parts of the program, namely the assembly of the flow residual and of the cost function, but not on the "upper level" resolution algorithms. In any case, differentiation of a resolution algorithm makes little sense, and is dangerously expensive for the reverse mode of AD, which we are using. The other aspect is local to the subroutines we are differentiating with AD: we describe a number of improvements to further reduce the memory needed by reverse-differentiated programs. Eventually, this memory grows in an acceptable proportion with the problem size, i.e. the number of mesh nodes. This opens the way for application of AD to very large real problems.

There have been previous experiments made with ODYSSEE, on a 2D problem. They showed the hard problems in reducing memory usage, which are addressed by the present work. In particular we give experimental figures that demonstrate the relative impact of the necessary AD improvements. These improvements are based on formal dataflow properties of the program, and therefore can be (and are being) automated in TAPENADE. The application problem is now a non-trivial 3D optimization of the sonic boom produced under a supersonic business jet, which confirms the soundness of this approach.

This work shows that powerful and accurate dataflow analyses can make the difference for the reverse mode of AD. In particular *in-out* and *data dependence* are a key to reducing the memory consumption. Also, we need versatile tools to debug and validate the differentiated programs. Finally, for approaches such as the present one, which combine AD and "manual" differentiation, an AD tool should provide the user with dependency information between differentiable

variables, even *outside* of the part of the code that is differentiated by AD. Otherwise, manual differentiation may go wrong because the developer overlooked some dependencies that contribute to the final derivatives.

Acknowledgments

This work was partly supported by the Comité d'Orientation Supersonique of the French Ministry for Research. We thank Dassault Aviation for making available the geometry of their Supersonic Business Jet. We thank the CINES for providing the computational facilities to run the largest examples.

References

1. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM (2000)
2. Corliss, G., Faure, C., Griewank, A., Hascoet, L., Naumann(editors), U.: Automatic Differentiation of Algorithms, from Simulation to Optimization. Springer (2002) Selected proceedings of AD2000, Nice, France.
3. Hascoet, L., Fidanova, S., Held, C.: Adjoining independent computations. in [2] (2001) 185–190
4. Hascoet, L., Greborio, R.M., Pascual, V.: Computing adjoints by Automatic Differentiation with TAPENADE. Research report *to appear*, INRIA (2003) On-line documentation on www-sop.inria.fr/tropics/tapenade.html.
5. Vázquez, M., Dervieux, A., Koobus, B.: Aerodynamical and sonic boom optimization of a supersonic aircraft. Research report 4520, INRIA (2002)
6. Huffman, W., Melvin, D., Young, D., Johnson, F., Bussoletti, J., Bieterman, M., Hilmes, C.: Practical design and optimization in computational fluid dynamics. AIAA paper 93-3111 (1993)
7. Mortchelewicz, G.: Résolution des équations d'Euler tridimensionnelles instationnaires en maillages non structurés. La Recherche Aérospatiale **6** (1991) 17–25
8. Seebas, R., Argrow, B.: Sonic boom minimization revisited. AIAA paper 98-2956 (1998)
9. Maglieri, D., Plotkin, K.: Aeroacoustics of flight vehicles: theory and practice. Acoustical Society of America publications (1991)
10. Stoufflet, B., Periaux, J., Fezoui, L., Dervieux, A.: 3-D hypersonic Euler numerical simulation around space vehicles using adapted finite elements. AIAA paper 86-0560 (1987) 25th AIAA Aerospace Meeting, Reno, Nevada.
11. Jameson, A.: Aerodynamic design via control theory. Journal of Scientific Computing **3** (1988) 233–260
12. Courty, F., Dervieux, A., Koobus, B., Hascoet, L.: Reverse Automatic Differentiation for optimum design: from adjoint state assembly to gradient computation. Research report 4363, INRIA (2002)