GROWTH Project GRD1-1999-10752 "AEROSHAPE"

APPLICATION OF A REVERSE SYMBOLIC DIFFERENTIATOR TO THE LOW-STORAGE ASSEMBLY OF AN ADJOINT STATE EQUATION

Contribution of partner INRIA to Task 3.1

Contribution to Deliverable D3-18

F. Courty, A. Dervieux, B. Koobus, L. Hascoet
Inria Sophia-Antipolis BP 93
06902 Valbonne Cédex, France

June 4th, 2001

Summary

This report contributes to the discrete sensitivity approach. The (exact) discrete adjoint system is assembled with the help of Automated Differentiation. The so-called reverse mode of differentiation allows the assembly of the adjoint residual and of the chain rule final functional gradient by calling only once the tangent routines. The main difficulty in the proposed approach is the useless storage of intermediate variables inside assembly loops. This is solved today by an a priori adaptation of the software to differentiate and some a posteriori manipulation of the resulting differentiated software. The second part of the report concentrates on different ways in solving the adjoint systems when a simplified Jacobian is available, and on obtaining an exact or simplified gradient of the objective functional.

1 Introduction

Task 3.1 of Work Package 3 in Aeroshape is dedicated to **"Sensitivity computations based on discrete equations"**. More precisely, it is defined as follows:

"The technique to be studied is based on the discrete adjoint and flow equations which are solved in order to compute the gradient of the cost function and constraints. In order to compute derivative terms in the gradient expression as well as in the flow and adjoint equations, automatic differentiation (AD) might be used. The present technique will be extended to deal with the Navier-Stokes equations using AD and different accuracy levels for the adjoint equation in order to find a good compromise between accuracy and computational time and complexity."

INRIA contribution will cover all the aspects of the above definition.

The use of an optimization loop relying on an *exact discrete gradient* in optimal design in aerodynamics sets the problem of the complex calculation of the gradient of the considered CFD software.

A first way explored by Mohammadi ([7]) relies on the symbolic differentiation of the complete software suite, including flow solver and objective functional evaluation.
This way introduces many extra problems related with solution algorithm complexity and massive useless storage that have to be solved by hand manipulations.

Another approach, applicable in the short term, is the symbolic differentiation of the smaller part of the software which assembles the state equation.

This report aims at showing how the different terms of the adjoint equation and of chain rule functional derivative can be obtained by applying the reverse mode of symbolic differentiation. The differentiation tool ODYSSEE is the one considered for applications.
Then we focus in the solution algorithm for solving the adjoint system.
We then conclude by comparing several strategies using both exact and simplified adjoint systems in order to produce exact or approximated functional gradients.

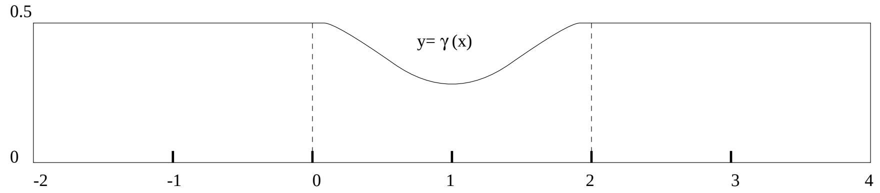

FIG. 1 – *The nozzle model problem*

2 An optimization problem

2.1 A model problem

We start with a model problem. It consists in solving an shape inverse/optimization problem for a family of 2D nozzles in which the flow is modelled by the Euler equations. By inverse/optimization we mean that the formulation is that of an optimization problem, but the precise description will be, for assessment purpose, that of an inverse problem. The flow model is moderately complex and mesh will be a variable one. The geometry is depicted in Fig.1, inlet and outlet are of constant section, and the median part can vary through the ordinates of the upper boundary. Variations around this formulation have been studied in an European BRITE-ECARP project ([4][5]).

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

Let us introduce:

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

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

The optimization problem to be solved is written:

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

with

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

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

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

where

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

in which:

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

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

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

where κ is 1.4.

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

$$\left\{ \begin{array}{rcl} \Psi(\gamma, W) & = & 0 \\ & & \text{(state)} \\ (\frac{\partial \Psi}{\partial W}(\gamma, W))^* \Pi & = & \frac{\partial J}{\partial W}(\gamma, W) \\ & & \text{(adjoint state)} \\ j'(\gamma) = \frac{\partial J}{\partial \gamma}(\gamma, W) - < \Pi, \frac{\partial \Psi}{\partial \gamma}(\gamma, W) > & = & 0 \\ & & \text{(optimality)} \end{array} \right. \tag{8}$$

2.2 Discrete standpoint

The mesh is made of triangles. It starts from an orthogonal triangulation with coordinates (x^1_{ij}, y^1_{ij}) of the constant-section nozzle geometry i.e. defined by $\gamma = 1$.

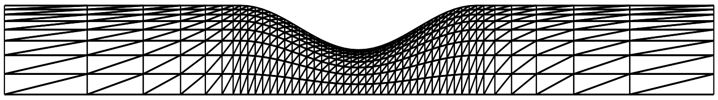

FIG. 2 – *Nozzle mesh with 423 vertices*

For any given control $(\gamma_1, i = 1, m)$, we derive a mesh on a accordion-like mode (Fig.2), that is by sliding vertices along vertical lines:

$$\forall i \in \{1, \ldots, m\},\ y^{\gamma}_{im'} = \gamma_i\ \forall j \in \{1, \ldots, m\},\ y^{\gamma}_{ij} = y^{\gamma}_{im'}\ y^{1}_{ij} \qquad (9)$$

m is the number of points in x direction in the interval $[0, 2]$ and m' is the number of points in the y direction.
In the sequel, the coordinates of the mesh will be shortly represented by the vector X of dimension M $(M = mm')$.
The mesh coordinates X are thus defined by (9) as an explicit function of the control parameter γ.

For discretizing the Euler equations, and more precisely the flux balance Ψ, we apply a vertex centered upwind finite-volume formulation (relying on median dual cells) ([6]) in which first order upwinding is applied for simplicity.

$$\Psi(X, W^n_j) = \sum_{k \in V(j)} \Phi(W^n_j, W^n_k, \vec{\eta}_{jk}(X)) + \text{ boundary conditions} \qquad (10)$$

where $V(j)$ is the set of nodes (=vertices) that are neighbors of j, ∂C_j the cell boundary associated with node j, $\partial C_{jk} = \partial C_j \bigcap \partial C_k$, $\vec{\eta}_{jk} = \int_{\partial C_{jk}} \vec{n}\, d\sigma$ and $\vec{n}$ the outward pointing normal vector to ∂C_j.
The function Ψ depends on state variable W through the two first variables of the Riemann solver Φ and depends on the mesh coordinate X through the elementary fluxes $\vec{\eta}_{jk}(X)$.

The upwinding in elementary flux Φ is the van Leer flux splitting (which is differentiable). It is useful to know that state solution will be iteratively obtained from a linearised implicit time advancing:

$$\left(\frac{M}{\Delta t^n} + \Psi'(W^n)\right) \delta W^{n+1} = -\Psi(W^n) \qquad (11)$$

M is the diagonal mass matrix, and when the time step Δt^n grows infinitely, Algorithm (11) becomes a Newton iteration for solving $\Psi(W) = 0$.

We are now considering an optimization with m real parameters:

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

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

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

The discrete variant of (2) writes:

$$
\text{Find } \gamma_0 \text{ in } I\!\!R^m \text{ such that } j(\gamma_0) = \min_{\gamma \in I\!\!R^m} j(\gamma). \tag{12}
$$

We obtain an adjoint state and optimality conditions by a way similar to that of the previous section.

2.3 optimization with gradient

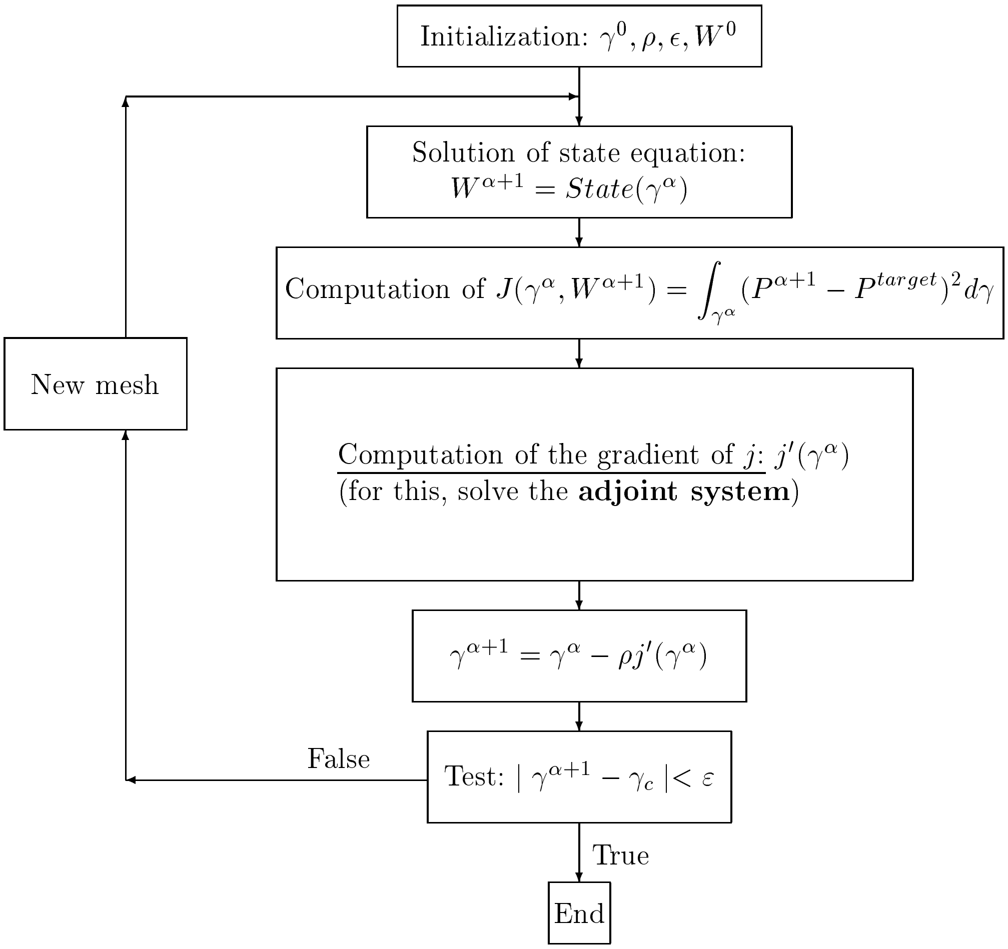

FIG. 3 – *Description of the "adjoint algorithm"*

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

3 Reverse mode by Automatic Differentiation

Automatic or Algorithmic differentiation is a topic with fast expansion, and we refer the interested reader to the bibliography list [3] and to the recent monography [2].

We focus on the *program differentiation* approach. By program differentiation, we mean that we shall apply a tool that take as input data the source code (typically in FORTRAN) of a routine that compute the result $f(x)$ as a function of the data x:

$$f : \mathbb{R}^m \longrightarrow \mathbb{R}^n$$

and will generate, as output result, a new source code (also in FORTRAN) that computes from any data x, the product of the Jacobian Df by a given vector.

3.1 Direct differentiation mode

The *direct differentiation mode* produces a directional derivative, that is the routine that computes, given an argument x and a direction δx, the increment $f'(x)\delta x$.

From the `FORTRAN` point of view, the function f is the composition of a set of elementary instructions, that represent also functions:

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

From the chain rule, we get :

$$\begin{aligned} Df(x_0) \quad = \quad & (Df_p \circ f_{p-1} \circ ... \circ f_1(x_0)) \\ & .\ (Df_{p-1} \circ f_{p-2} \circ ... \circ f_1(x_0)) \\ & \quad ... \\ & .\ (Df_1(x_0)). \end{aligned} \tag{13}$$

In the direct mode, for a given direction δx the "tangent linear program" follows the order of the initial code instructions.
It first evaluates both:
$f_1(x_0)$ and the increment $< (Df_1)(x_0), \delta x >$.

Then it computes both :
$f_2 \circ f_1(x_0)$ and the increment $< (Df_2) \circ f_1(x_0), \delta x >$.

And then:
$f_3 \circ f_2 \circ f_1(x_0)$ and the increment $< (Df_3) \circ f_2 \circ f_1(x_0), \delta x >$.

At this step, for example, we observe that the composite value of the current variable $f_2 \circ f_1(x_0)$ is available for both calculation from the immediate previous step, which means that no special saving has to be done.
Then the direct mode follows the initial program algorithm and do not require special savings.

However, the differentiated code produces as a result an array of dimension n . If we wish to obtain the complete $m \times n$ Jacobian matrix defined in (13), we need to call the "tangent linear program" exactly m times, viz. with successively as input increment δx the m basis vectors of $I\!R^m$.

3.2 The reverse mode

The *reverse differentiation mode* produces $f'^*(x)\delta f$, result of the action of the transposed Jacobian $f'^*(x)$ on a given vector δf .

From the chain rule and after transposition, we have:

$$\begin{array}{rcl} Df(x_0)^T & = & \quad (Df_1(x_0))^T \\ & & .\ (Df_2 \circ f_1(x_0))^T \\ & & .\ (Df_3 \circ f_2 \circ f_1(x_0))^T \\ & & \quad \ldots \\ & & .\ (Df_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x_0))^T \\ & & .\ (Df_p \circ f_{p-1} \circ \ldots \circ f_1(x_0))^T. \end{array} \tag{14}$$

Let δy be an array of $I\!R^m$.
The computation of $< ((Df)(x_0))^T, \delta y >$ starts with the calculation of:

$$\delta y_p \;=\; (Df_p \circ f_{p-1} \circ \ldots \circ f_1(x_0))^T.\delta y$$

and then:

$$\delta y_{p-1} \;=\; (Df_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x_0))^T.\delta y_p$$

and so on.
We observe that these operations are done in the inverse order with respect to the initial algorithm, and also that each of the above step need the knowledge of the intermediate result $f_k \circ f_{k-1} \ldots \circ f_1(x_0)$.
A natural way to do this without multiple recomputations is to have first a computation of $f_p \circ f_{p-1} \ldots \circ f_1(x_0)$ (*forward sweep*) involving the *storage* of the whole set of intermediate results, followed by a backward computation (*reverse sweep*) of the successive derivatives.

This is the principle of the reverse mode of ODYSSEE, the software developed by INRIA and applied to this project.

Let us call "cotangent linear" program, the FORTRAN software generated by the reverse mode. By comparison with the similar software generated by the direct mode, we observe first that since we have forward and reverse phases, this results in an larger complexity (in term of extra arithmetics).

But the main problem of the reverse mode is that a vast amount of storage may be necessary between the forward sweep and the reverse sweep. The management of this storage is done by special instructions, which are automatically inserted into the generated code. Notice furthermore that, in practical implementation of the reverse differentiation algorithm, some superfluous values may be stored and retrieved. An improved A.D. tool should avoid this unnecessary storage.

If now we consider the result of a "cotangent linear" program, we observe that it returns an array of dimension m. This allows elaborating a strategy for choosing between both modes:

- for n larger than m, it is more interesting to use the direct mode,

- for m larger than n, the reverse mode can be more interesting,

- for both case, for n and m larger than one, we have to compute several times for obtaining the complete Jacobian.

The particular case of a scalar-valued functional, $n = 1$, is much more efficiently treated by the reverse mode since it provides m results while the direct one provides only *one* result. We shall in the sequel illustrate the interest of the reverse mode for computing adjoints in Optimum Control.

3.3 Application of reverse mode to optimum design

The sequel concentrates on reverse differentiation. In order to convince the reader that the reverse mode is applicable to our context, we propose a new formulation of the above optimality conditions involving only tranposes:

$$
\left\{
\begin{array}{rcl}
\Psi(\gamma, W) & = & 0 \\
\text{(vanishing of state residual)} & & \\
(\dfrac{\partial \Psi}{\partial W}(\gamma, W))^{*}\Pi & - & (\dfrac{\partial J}{\partial W}(\gamma, W))^{**} \ = \ 0 \\
\text{(vanishing of adjoint residual)} & & \\
j'(\gamma) = (\dfrac{\partial J}{\partial \gamma}(\gamma, W))^{**} - (\dfrac{\partial \Psi}{\partial \gamma}(\gamma, W))^{*}\Pi & = & 0 \\
\text{(optimality)} & &
\end{array}
\right.
\tag{15}
$$

Since the functional J is scalar-valued, the above remark concerning the direct obtaining of a gradient by using the *reverse mode* applies. We indicate this by $**$. The other applications of the reverse mode for obtaining tranposes of Jacobians of Ψ times vectors is explained in details in the sequel.

The main point to remember now is that the proposed strategy is to apply the reverse mode four times, by differentiating two functions, J and Ψ, with respect to two variables, γ and W.

3.4 Low storage by iteration-wise adjoining

The data storage problem in reverse A.D. is a challenging difficulty. We concentrate now on dealing with it for the above type of target routines, that are, essentially *assembly loops*, according to the finite element vocabulary.
We remark that these loops are *parallel loops*, i.e. loops which can be computed in any order of the index without changing the result of the computation. Several authors such as Griewank ([2]), Hovland, Bischof and Mohammadi ([8]), have proposed strategies for managing with the memory consumption problem of parallel loops.
We propose to apply a new one: the **"iteration-wise adjoining"** described in paper [1] accessible on the **web**.
In short, for parallel-reduction loops the following contraction is possible:

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

The notation $\overline{\textit{body}}$ represents the reverse sweep corresponding to *body*.

This manipulation reduces drastically the amount of memory used (by a factor N, size of the loop).

So far, ODYSSEE does not *yet* implement the automated application of this refined technique; such a development is planned, but will not be done in 2001. In the meantime, we propose a semi-automatic strategy, that combines application of the existing ODYSSEE tool with *hand manipulations* of the source file and the generated file. Our goal is simply to minimize the work to do by hand, and leave as much as possible to the automatic tool. It is also to minimize the risk of manipulation errors.

3.5 Step-to-step checklist for Iterationwise adjoining

Step 0: Building the FORTRAN file : an important *a priori* rule is that the differentiated FORTRAN program should be a a complete subtree of the FORTRAN program, in the sense that if a routine is loaded into ODYSSEE, any subroutine of it should be also loaded into ODYSSEE. If you do not respect this rule, ODYSSEE will still produce an accurate derivative, but, since its algorithm cannot detect arguments that are not used or changed in missing subroutines, it will save them unnecessarily.

We concentrate now on the loop problem. Let us call `WFLO` the routine to differentiate, sketched on the next figure:

(Initial) $\Longrightarrow$

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

Step 1: The process starts with a *preparation of the original program*, that extracts parallel loops and parallel loop bodies in separate subroutines. For each "parallel-reduction" loop, extract it from its subroutine, replace it by a " `CALL WFLOLOOPn(...)`", define `WFLOLOOPn(...)` as a subroutine that does the loop, and inside the loop, just do a " `CALL WFLOBODYn(...)`", and finally define `WFLOBODYn(...)` as a subroutine that does the work. This step leads to the program illustrated by the following figure.

(by hand)
$\Longrightarrow$

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

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

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

Step 2: Then ODYSSEE in reverse mode is applied to the resulting set of routines (calling routine, `WFLOLOOPn` routines, and `WFLOBODYn` routines in a unique file). Let us say that we differentiate with respect to variable `XLFO`.

```
>odyssee
odytop>load wflo.f
odytop>diff -cl -h wflo.f -vars XFLO
odytop>get_program wflocl wflocl.f
```

this will generate a set of new FORTRAN subroutines `WFLOCL`, `WFLOLOOPnCL` and `WFLOBODYnCL`, sketched by the next figure. The original routines `WFLOLOOPn` and `WFLOBODYn`, even if they might be called by the differentiated program, are not shown because they are identical to their definitions in the previous figure.

(ODYSSEE)
$\Longrightarrow$

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

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

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

Step 3: The file generated by ODYSSEE must be modified by hand. To this end, first copy the generated file (`wflocl.f`) into a new file (`wflocl_opt.f`), which we are going to modify.

Step 4:

Try to compile the differentiated program.

It often happens that ODYSSEE creates new arrays (for snapshots), but is not able to guess the size of these arrays at compile time. These sizes are then denoted by variables named `ody***`, and the user is requested for the definition of these. Then put these definitions in a file named `odyparam.inc`

Due to an ODYSSEE bug (to be solved in the next version), it may happen in the generated program that some uses of a PARAMETER occur *before* this parameter is declared. If this problem arises, one must fix it by hand, by moving the PARAMETER declaration upwards.

Step 5:

Examine the differentiated program `wflocl_opt.f`:

Two kinds of storage are implemented (with a slightly different format):
- variables that are modified are pre-stored,
- before `CALL` instruction, a *snapshot* of arguments is stored.

Inside each `WFLOLOOPnCL` routine, erase the forward sweep with its snapshots. In the reverse sweep, erase the *Restore* operations. =>only one loop remains, containing only `CALL WFLOLOOPnCL(...)` remains.

Step 6: ODYSSEE implements the *Save/Restore* mechanism through the use of variables, named `SAVE*`. Since there is no such *Save/Restore* in the `WFLOLOOPnCL` routines, the declarations of the associated `SAVE*` variables can, and must be removed.

These two last steps of the process lead to an optimized program, illustrated by the following figure.

(by hand) $\Longrightarrow$

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

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






  call WFLOBODY3CL(i,...)
EndDo
End

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

Some comments :

Several extra optimizations may be done:
– as we said earlier, there are many other useless `SAVE*` variables. But this is independent from the optimization presented here.
– unused values should not be saved; this also holds for values that are not used in the derivatives; variables that are initialized should not be saved before initialization.
– in principle unmodified values should not be saved in the snapshot.
– at the end of a routine: if the last executable instruction is a call to a routine G, the sequence in the derivative program:

$$\texttt{snapshot} \;\; ; \;\; G \;\; ; \;\; \texttt{restore} \;\; ; \;\; \bar{G}$$

can be replaced by:

$$\bar{G}\,.$$

– The routines `WFLOLOOPn(...)`, `WFLOBODYn(...)`, and the differentiated routines `WFLOLOOPnCL(...)`, `WFLOBODYnCL(...)`, could be probably inlined back into their calling sites. Our experiences show this gives a rather small gain.

4 Low-storage gradient assembly with AD

By low-storage we mean that Jacobians are never directly applied, but, instead, their product with vector, yielding results that are vectors of yet tractable storage. We show now that it is possible for two terms arising in the optimality conditions (8).

4.1 Assembly of functional J gradients

We consider the building of the $**$ derivatives in (15):

$$\left\{ \begin{array}{lcl} DJ_W & = & (\dfrac{\partial J}{\partial W}(\gamma, W))^{**} \\ \\ DJ_\gamma & = & (\dfrac{\partial J}{\partial \gamma}(\gamma, W))^{**} \end{array} \right. \tag{16}$$

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

For this, we shall differentiate the routine `SUBROUTINE JCOST` that has as input data the mesh coordinates X_M in FORTRAN : `XMESH(1:M)`, and flow variables W_N, in FORTRAN : `WFLO(1:N)`, and the output result is the scalar `CVALUE`.

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

DJ_W **:** The application of ODYSSEE in *reverse or cotangent linear mode*, option `-cl`, to this context by:

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

will generate a new FORTRAN routine "`SUBROUTINE JCOSTCL`" with same in/out data except:

- an extra input scalar data, `CVALUECL`, which should be set at running time equal to 1,

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

DJ_γ **:** The application of ODYSSEE in *reverse or cotangent linear mode*, option `-cl`, to the *same* context by:

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

will generate a new FORTRAN routine "`SUBROUTINE JCOSTCL`".

N.B. this means that you should have saved the previous `SUBROUTINE JCOSTCL` obtained in previous section with another name.

The new routine has the same in/out data as `SUBROUTINE JCOST` except:

- an extra input scalar data, `CVALUECL`, which should be set at running time equal to 1,

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

4.2 Assembly of the Adjoint equation residual

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

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

$$(\Psi(X_M, W_N))_N \;=\; 0. \tag{17}$$

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

$$S_N \;=\; (\;(\frac{\partial \Psi}{\partial W}(X,W))^t_{N\times N}\;.\;\Pi_N\;)_M \;\;-\;\; (\frac{\partial J}{\partial W}(\gamma, W))_N\;. \tag{18}$$

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

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

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

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

The application of ODYSSEE to this context

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

will generate a new FORTRAN routine "`SUBROUTINE FLUXCL`" with same in/out data except:

- an extra input data, `PSIFLUCL(1:N)`, which should be set at running time equal to Π_N, that is the successive iterates of the adjoint state solution,

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

$$(\frac{\partial \Psi}{\partial W}(X,W))^t_{N\times N}\;.\;\Pi_N\;. \tag{19}$$

Therefore, the adequate call of `subroutine fluxcl` will produce the expected calculation.

4.3 Assembly of the sensitivity of state residual

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

$$T_n \; = \; (\frac{\partial \Psi}{\partial \gamma}(\gamma, W))_{N\times m} \; . \; \Pi_N. \tag{20}$$

Where Π is assumed to be given. In practise, we shall apply first a differentiation with respect to the mesh coordinates X:

$$T_n \; = \; (\frac{\partial \Psi}{\partial X}(X, W))_{N\times M} \; . \; (\frac{\partial X}{\partial \gamma}(\gamma))_{M\times n} \; . \; \Pi_N. \tag{21}$$

It is then possible to introduce the following transposition:

$$T_n \; = \; (\frac{\partial X}{\partial \gamma}(\gamma))_{M\times n} \; . \; (\,(\frac{\partial \Psi}{\partial X}(X, W))^t_{M\times N} \, . \, \Pi_N \,)_M \tag{22}$$

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

$$F(X)_1 \; = \; \Psi(X, W)_N . \Pi_N \; . \tag{23}$$

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

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

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

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

The application of ODYSSEE to this context will generate a new FORTRAN routine "`SUBROUTINE FLUXCL`", different from the one built in the previous section (*also saved beforehand with another name*). It has same in/out data as `SUBROUTINE FLUX` except:

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

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

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

It remains to multiply by $\frac{\partial X}{\partial\gamma}(\gamma))_{M\times n}$.

4.4 Large arrays

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

One word about the arrays *inside the routines to differentiate.*
It should be emphasized that intermediate large arrays such as:
- transformation of flow variables (from conservative to primitive,...),
- storage of metrics (normal vector to interfaces,...),
are susceptible to induce the differentiator in building *inside the differentiated routine* large matrices (such as the derivative of primitive flow variables with respect to conservative ones,...).

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

All arrays are vectors except the possible preconditioner of state and adjoint solution algorithms.

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

4.4.1 Adjoint computation

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

$$D_{\Delta t}\Pi\ +\ (\frac{\partial\Psi}{\partial W}(X,W))^*\Pi\ -\ (\frac{\partial J}{\partial W}(\gamma,W))^{**}\ =\ 0\ . \tag{25}$$

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

$$A^*(\Pi^{iter+1} - \Pi^{iter}) \;=\; -(\frac{\partial\Psi}{\partial W}(X,W))^*\Pi^{iter} \;+\; (\frac{\partial J}{\partial W}(\gamma,W))^{**} \; . \tag{26}$$

The only matrix stored is the preconditioner.

4.4.2 Final derivatives

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

$$j'(\gamma) = (\frac{\partial J}{\partial \gamma}(\gamma,W))^{**} - (\frac{\partial X}{\partial \gamma}(\gamma))^*(\frac{\partial\Psi}{\partial X}(X,W))^*\Pi \; . \tag{27}$$

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

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

4.5 Performances

The codes obtained with the above strategy have been measures as efficient and storage saving, since their CPU costs have been measures as less that 3 times the routine to differentiate, and the storage required has been measured as not larger that 2 times the routine to differentiate.

5 State and co-state solution

The above differentiation of state residual produces an adjoint state residual. In order to solve the equality of this residual with zero, we need a matrix-free method, that is a method not needing the exact matrix of the state equation.

Two answers are possible:
- (a) since the adjoint residual evaluation is not much more expansive than the state equation, we can consider solving it with an explicit time advancing iteration.

- (b) in the case where the initial software involves already a simplified preconditioner for the state equation, we can try to use it for solving the adjoint state equation.

In the initial code we use, a simplified "spatially first-order" Jacobian matrix of the state equation is available and applied to the state equation solution by means of a linearized implicit time advancing:

$$A_1 = (\Psi_1)'(W^n) \tag{28}$$

$$\bar{W}^{n+1} \;=\; W^n - \left(\frac{M}{\Delta t^n} + A_1\right)^{-1} \Psi_2(\gamma, W^n). \tag{29}$$

It is then very natural to solve the adjoint state with the transpose of the state preconditioner:

$$\left\{\begin{array}{lcl} (A_1)^*\Pi^{(0)} & = & \dfrac{\partial J}{\partial W}(\gamma, W) \\ & & \\ (A_1)^*(\Pi^{(n+1)} - \Pi^{(n)}) & = & -(\frac{\partial \Psi_2}{\partial W})^* \prod^{(n)} + \dfrac{\partial J}{\partial W}(\gamma, W). \end{array}\right. \tag{30}$$

Here we have not considered the addition of the mass-matrix $\dfrac{M}{\Delta t^n}$ term. It can be useful to re-introduce it when the linear system with A_1 reveals to be difficult to solve.

6 Gradient validation

We describe in the sequel how the adjoint system is solved, producing an exact adjoint used in the final assembly of the objective functional gradient.

We want to emphasize that that the gradient obtained in this manner is the exact gradient of the discrete objective functional. We can then *check* it by direct comparison with an evaluation by divided differences, component-by component of the functional.
Perturbations as small as 10^{-7} are applied to the control components, state equation is solved to machine zero (10^{-14}), and second-order divided differences are applied. This dœs not produce the best accuracy for all components, see Fig. 4, but for any of these components, an adequate choice of the perturbation size give at least 6 exact digits.

7 Convergence : numerical experiments

We examine now the efficiency of several options in the use of first-order and second order jacobians.

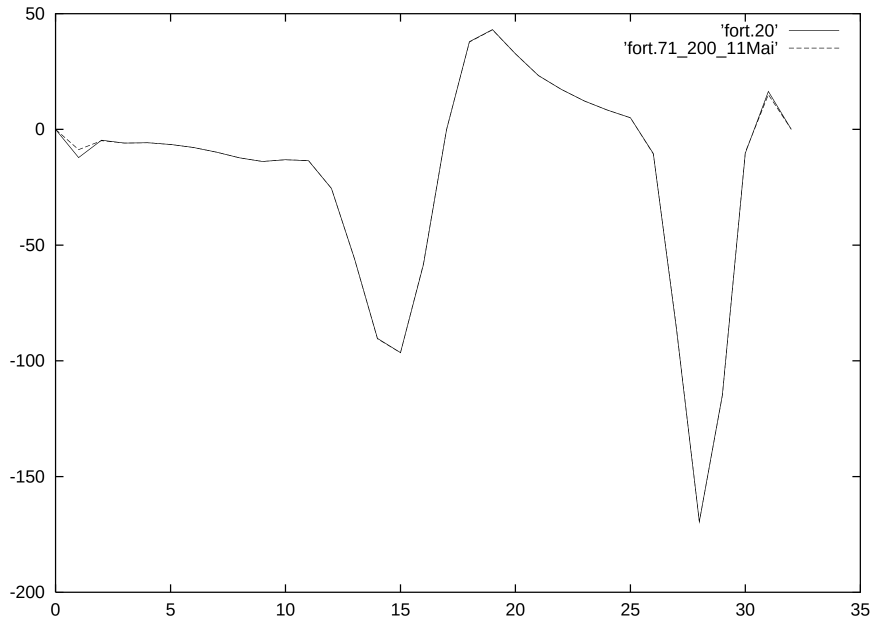

FIG. 4 – *Gradient validation : comparison between the exact gradient and second-order divided differences*

7.1 Test case description

We consider the above nozzle geometry, with a section restricted by a factor of 10%, a transonic flow obtained with a Mach number (inlet and outlet) of 0.74.
The mesh involve 400 vertices. The flow presents a mean strong shock, see Fig. 5.

7.2 Efficient convergence of the adjoint state

The adjoint state system should be more easy to solve than the state system: it is linear, and its operator is obtained as the exact transpose of the state Jacobian. We focus the discussion on the inviscid hyperbolic case.

7.2.1 Solving an hyperbolic system

During the last 20 years, CFD scientists have concentrated a large effort on the solution of Euler equations, under the usual nonlinear formulation as well as linearizations, for building linearised implicit and Newton iterative solvers.

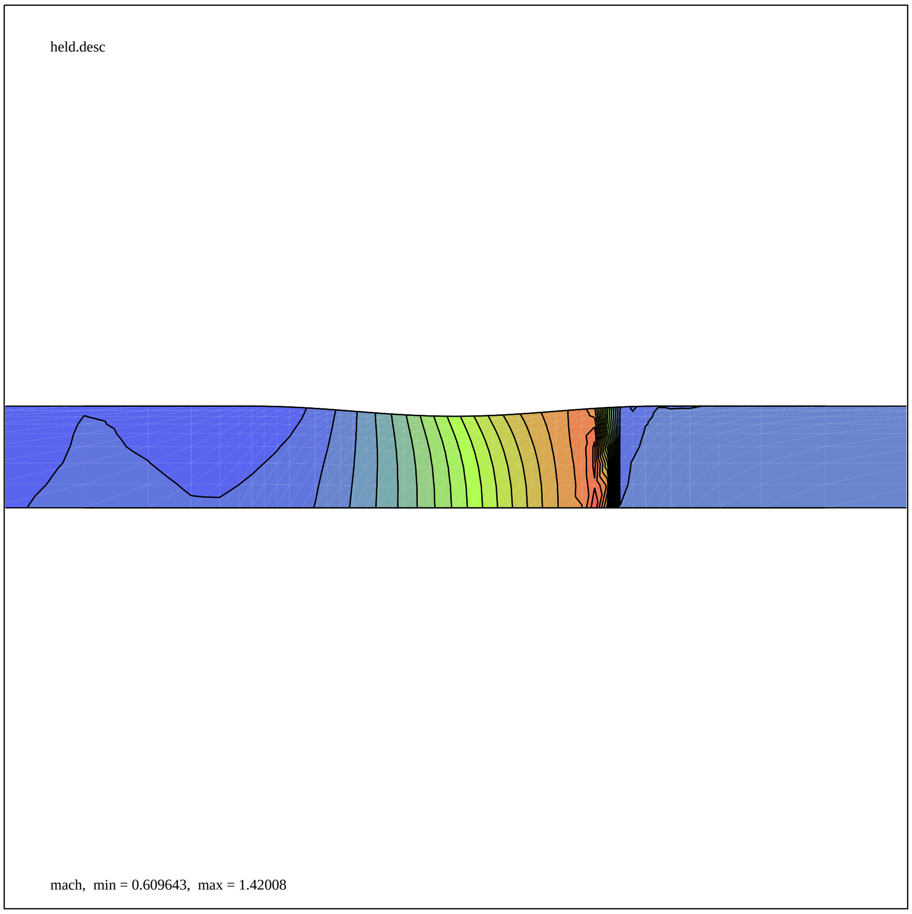

FIG. 5 – *Nozzle test case: Mach contours for the state equation, with inlet and outlet Mach specified to 0.74, minimum value is 0.609, and maximum is 1.420*

In some case, that we shall call (**"Type 1"**) these systems are discretized in such a manner that the diagonal is nearly dominant (if not strictly dominant), so that Jacobi (J) or Gauss-Seidel (GS) sweeps (and extensions à la SSOR) can be applied. This is obtained generally by introducing first-order Godunov spatial approximations, or in some case fully upwind second-order, or also compact multi-dimensional schemes, or even Lax-Wendroff-type elliptic terms. These usual iterations (J, GS,...) have a not so bad efficiency when only a few sweeps are applied at a time, inside external loops such as time advancing ones. They are also efficient as smoothers inside multi-grids schemes. More recent Krylov methods (GMRES,...) are also efficient, but do

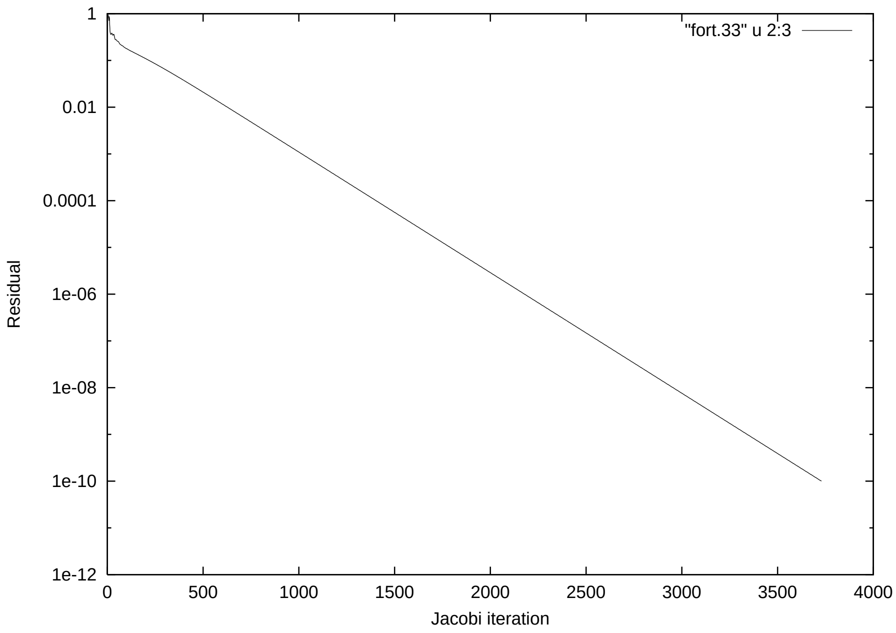

FIG. 6 – *Nozzel test case: solving the adjoint ; solution of first-order system, convergence history with residual as a function of number of Jacobi sweeps.*

not supersede the above sweeps, except when fully converged solutions are sought. We note finally that un-complete factorization methods have met also some success in combination with nearly diagonally dominant systems.

In most other cases, let us call them (**"Type 2"**), diagonals are not large enough for Jacobi or GS, explicit time stepping is built with multi-stage schemes. When GMRES is applied without preconditioning, convergence is disappointing.

We concentrate now on methods with quasi-dominant diagonals and more precisely on formulations in which the iteration is preconditioned with a spatially-first-order Jacobian, that is often refered as Defect-Correction (DeC) iteration.

7.2.2 Convergence of the first-order system

The first-order spatial system is solved by a block 4×4 Jacobi iteration, with relaxation parameter set to 1. Diagonal blocks are inverted with Gauss elimination. In Fig. 6, the convergence of 10 decades is obtained in 3500 iterations (14 decades in 5000 iterations). If the adjoint residual is also set to first-order spatial accuracy, then in(30), the simplified Jacobian is the exact

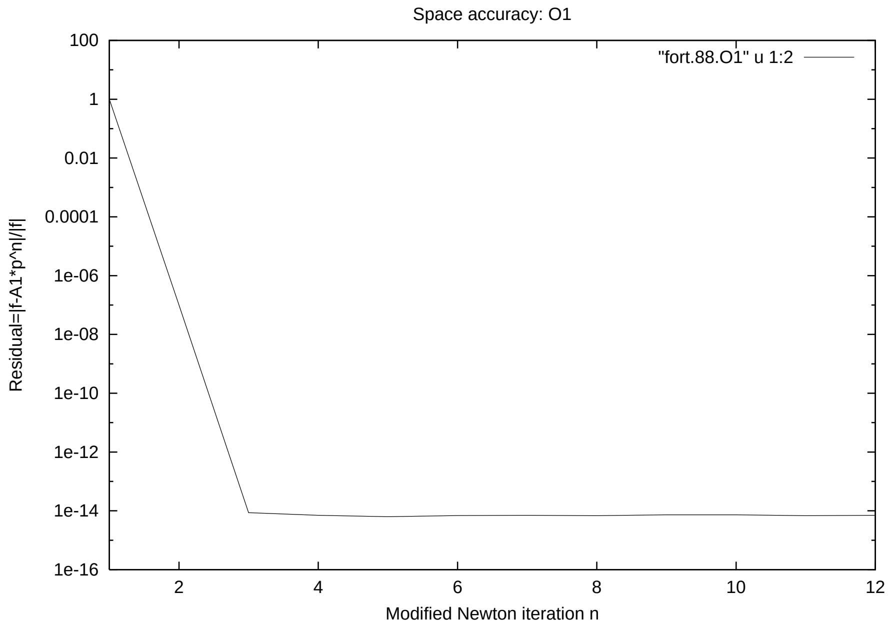

FIG. 7 – *Nozzle test case: solving the adjoint ; checking that the pseudo Newton with o1.o1 becomes indeed a Newton iteration (Jacobi phases are solved by 7 decades*

Jacobian and convergence is instantaneous, see Fig. 7.

7.2.3 Convergence of second-order system

The analysis of the Defect Correction iteration (see [4] predicts that its convergence ratio is mesh independent and equal to 0.5, that is giving 3 decades in 10 iterations.
In the experiment depicted in Fig. 8, any first order Jacobi phase is converged to 8 decades, which need between 1000 and 3000 Jacobi sweeps.
This does not allow (and converging Jacobi further not either) to find the theoretical DeC convergence, since the DeC residual need 50 iteration for a decrease of 3.5 decades (4.3 times more sweeps).
This can be explained by the centered differencing applied to the slip boundary conditions in our scheme (the centered scheme does not give a good convergence in combination with DeC, according to [4].

The rather extraordinary effort, of a total of 500000 Jacobi sweeps for this convergence is, fortunately superfluous. We obtain about the same DeC convergence by limiting the Jacobi phase to 2 decades, which turns to a total

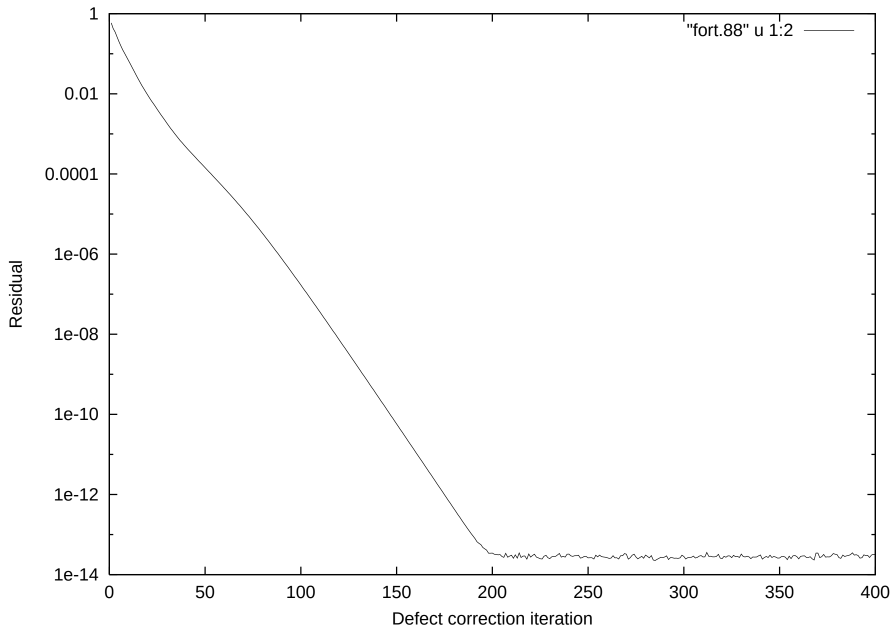

FIG. 8 – *Nozzle test case: solving the adjoint; convergence of the DeC iteration, when the first-order system is perfectly solved*

four times better but still large of 125000 sweeps, see Fig. 9.

In the same figure, we depict a quasi optimal compromise which consists in limiting to 20 sweeps the Jacobi phase, so that, with a total of 250 DeC iteration, we consume only 5000 Jacobi sweeps.
This means is the same figure as for first-order solution, which means that the extra cost for second-order accuracy is limited to the 250 second-order residual evaluation (in practise less than 50% of total cost).

In practise, i.e. in a gradient optimization loop, some complexity savings will be obtained since the adjoint iteration should not be converged that much and since good initial conditions for the adjoint will be inherited from previous gradient iterations.

7.3 Exact and simplified adjoint states

The understanding of the continuous adjoint system of Partial Differential Equations is a delicate question addressed already by several authors, see [7], [5], [6], [4]. One can remark indeed, that for the Euler equations, the adjoint system is first-order hyperbolic nature, linear, but non-conservative.

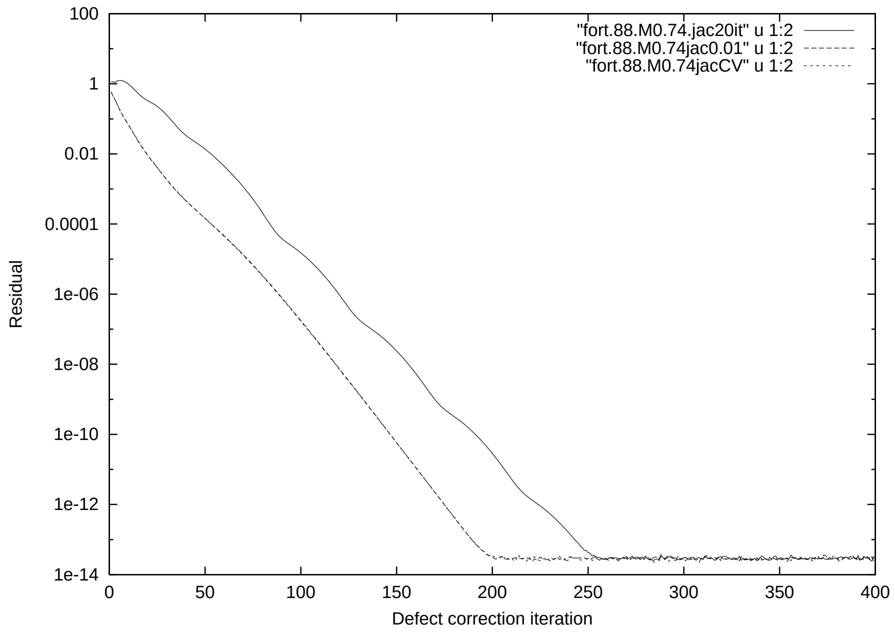

FIG. 9 – *Efficient convergence: two curves are almost identical, they show a convergence in 200 DeC iterations. One corresponds to a Jacobi phase that is fully converged, while in the second , convergence of Jacobi phase is pushed only for a two decade convergence. The third curve converges with 250 iterations, but with only 20 sweeps in each Jacobi phase.*

It presents diverging characteristics on the state variable shocks, and converging characteristics on state variable sonic lines.

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

At the contrary, sonic lines should turn to produce on the adjoint a kind of linear shock.

We propose a short comparison of the *exact* adjoint, depicted in Fig.s 15 to 11, with two different *simplified* adjoints, one obtained with solving with only the first order Jacobian, depicted in Fig. 10 , and one obtained by two iterations of DeC, depicted in Fig. 16.

Let us first examine the exact adjoint. We note that, as expected, the

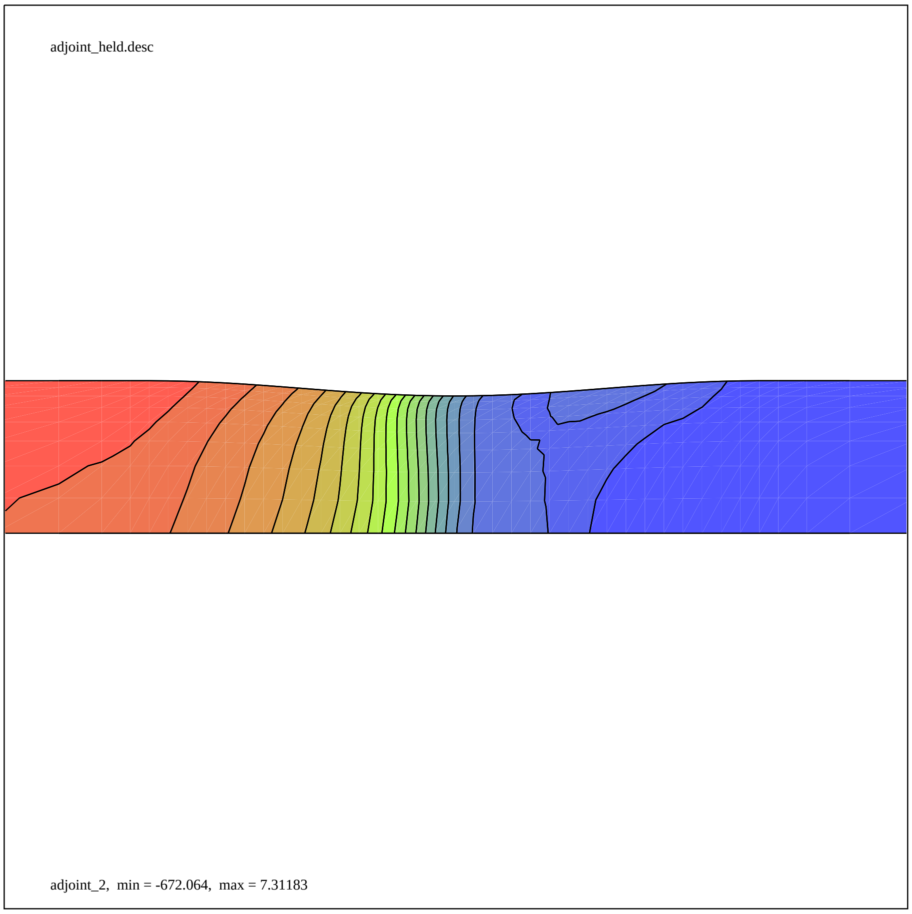

FIG. 10 – *First-order simplified of a second-order state. Second component, to be compared with figures 12 and 16.*

adjoint is smooth in the region where state variables show the discontinuity associated with the shock. This proves that the discrete derivative of the TVD scheme is viscous enough for giving the good continuous matching in the adjoint solution.
In Fig. 13 we show a horizontal cut of the second component. The matching is good at shock. At sonic location, we do not see really a discontinuity, but, from left to right a faster and faster increasing followed by a kind of small Dirac.

The comparison between simplified and exact adjoints show that the first order adjoint has lost the steep gradients at sonic lines (compare Figs. 10 and 12, while the DeC adjoint (Fig. 16) is a rather good approximation, but

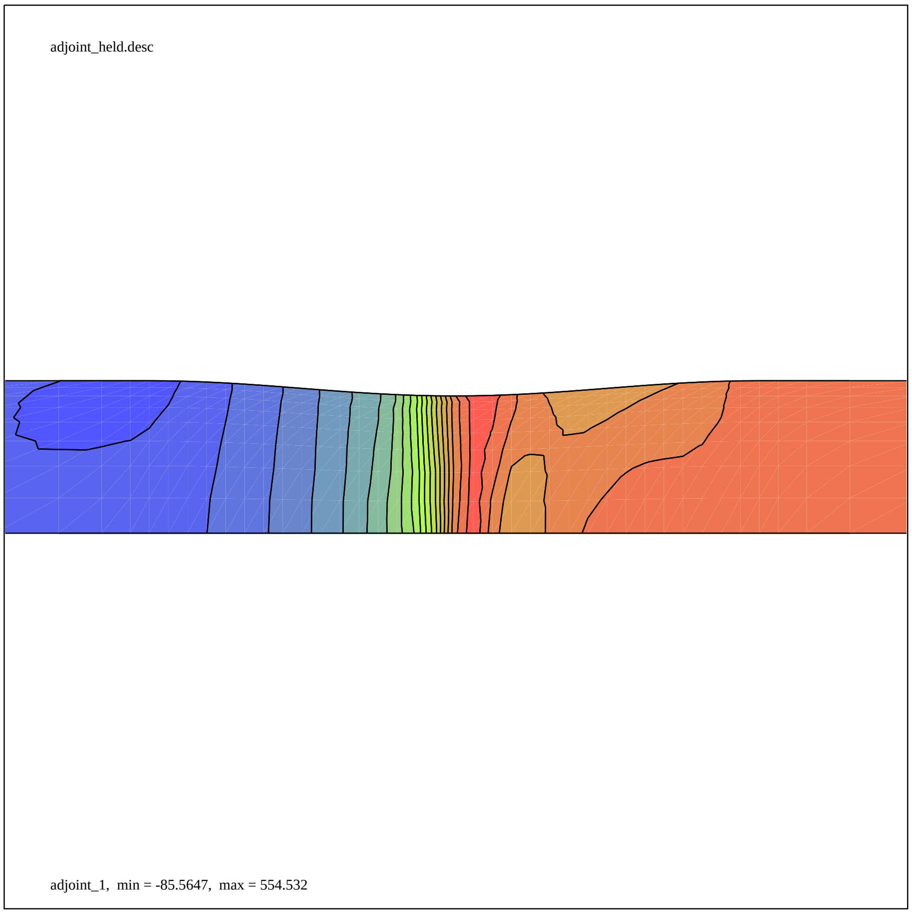

FIG. 11 – *Exact adjoint (second-order). First component.*

unfortunately shows some bad behavior near the upper boundary, at which traces are taken for assembling the gradient. Now the saving in using the DeC adjoint is in our experiences not evident (it is a few residual evaluations). In these conditions, we do not recommend to use the DeC adjoint except if the residual are very expensive to evaluate. This conclusion can also be influenced if a much faster solver is used for first-order system solution.

In order to complete the comparison, we depict the associate gradients of the functional in Fig. 17.

We observe a good correlation. With finer meshes, the deviation between DeC and exact gradient should reduce much faster than the deviation with first-order, since in the former case the difference is an $O(h^2$, according to

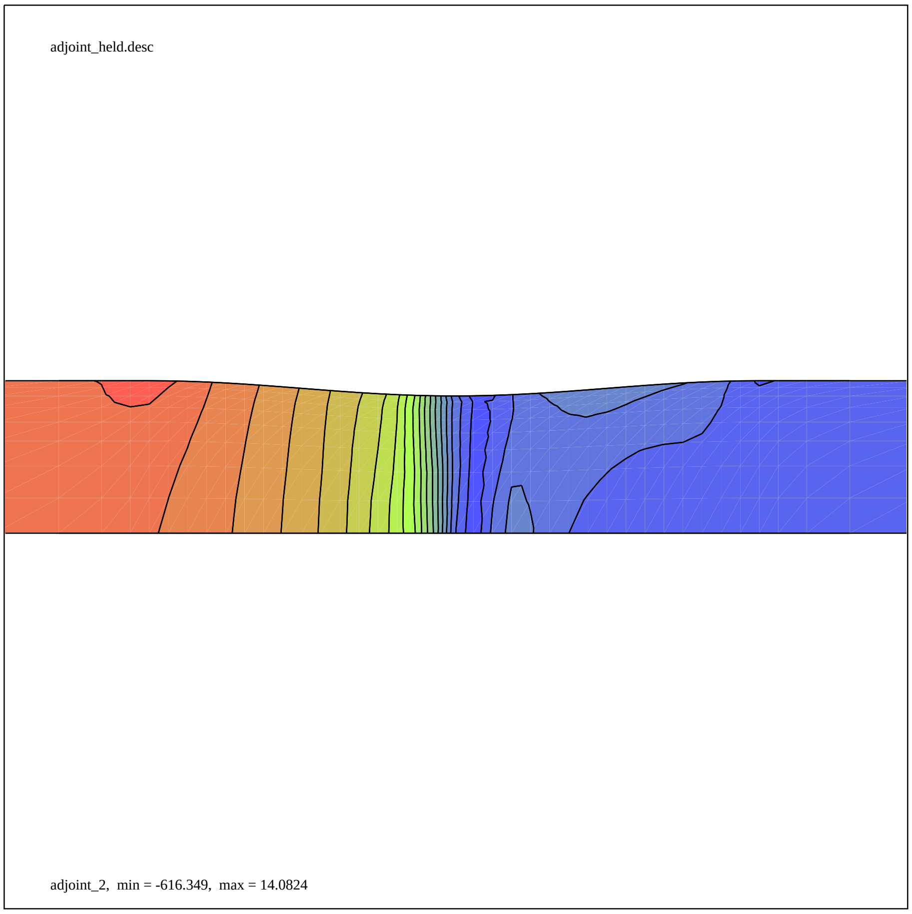

FIG. 12 – *Exact adjoint (second-order). Second component. To compare with figures 10 and 16.*

the theory of Defect Correction.

We remark that the exact gradient shows large variations on both sonic line (coordinate 27) and shock (coordinate 17). At shock location, the deviation between DEC and exact gradient is small as expected, since adjoints are smooth.

8 Conclusion

Optimal design is being renewed by adjoint methods. The INRIA team is one of the teams having made pioneering development of inverse mode Automated Differentiation (**IMAD**) Our purposes in Aeroshape Task 3.1 are

(a) to design at best an IMAD for adjoint development and (b) to promote adjoint development with this kind of tool.

Our strategy for reaching these purposes rely on a demonstrative work described in this report, and on interactions with Aeroshape partners interested by adjoint development using IMAD. Visits at INRIA have been organised with Partners Alenia and Dassault, one is planned with HCSA, and a date is discussed with BAE.

In the present report, the application of IMAD to the assembly of the different derivatives arising in the gradient computation is described. Some manual pre- and post-processing are still necessary. WE describe the complete process leading to the adjoint residual assembly, and to all other necessary derivatives assembly for computing the gradient. The resulting differentiated software is of satisfying performances.
We then discuss some options for solving the exact adjoint system or some simplified ones, without handling the exact Adjoint matrix. In the options that we compare, computing the exact adjoint with a simplified matrix is the best choice. The resulting exact gradient is validated and compared with two simplified strategies.

In future work, viscous extensions are planned. The sensitivity software we have obtained now will also be applied to the study of new families of optimization algorithms, in the context of Task 3.4.

Références

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

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

[3] Tropics Project. – On-line bibliography, in: `http://www-sop.inria.fr/safir/SAM/Odyssee/node10.html`

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

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

[6] Fezoui (L.) et Stoufflet (B.). – *A class of implicit upwind schemes for Euler simulations with unstructured meshes.* – Research report 517, INRIA, 1986.

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

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

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

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

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

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

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

[8] Giles (M.B.) et Pierce (N.A.) On the properties of solutions of the adjoint Euler equations *prépublication*(after 1996)

[9] Soto (O.) et Loehner (R.) CFD optimization using an incomplete-gradient adjoint approach *AIAA paper 00-0666, 38th Aerospace Sciences Meeting and Exhibit*, 10-13 janvier 2000, Reno, NV

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

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

[12] Lallemand M.-H., Steve H. , Dervieux A., Unstructured multigrid by volume agglomeration: current status, *Computer and Fluids*, 21, **3**, 397-433, 1992

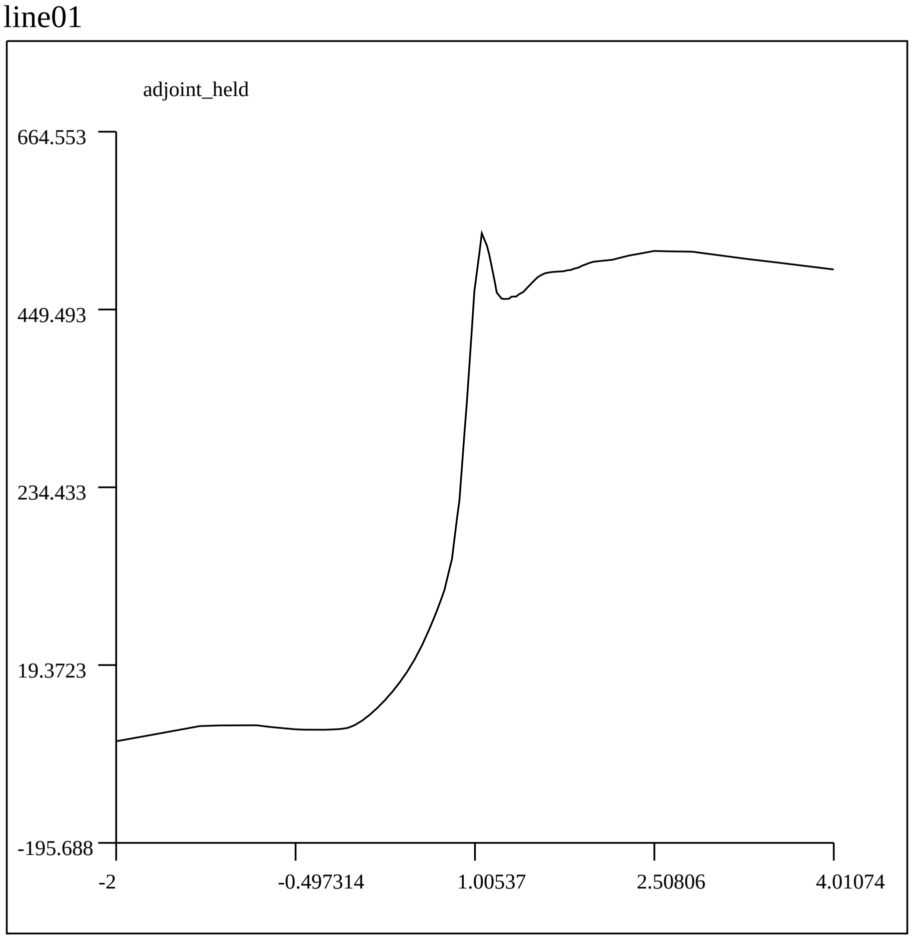

FIG. 13 – *Exact adjoint (second-order). Second component. Horizontal cut at mid-height. The pseudo-shock is located on the sonic line, where adjoint characteristics are convergent. The adjoint is smooth near the shock of state variable.*

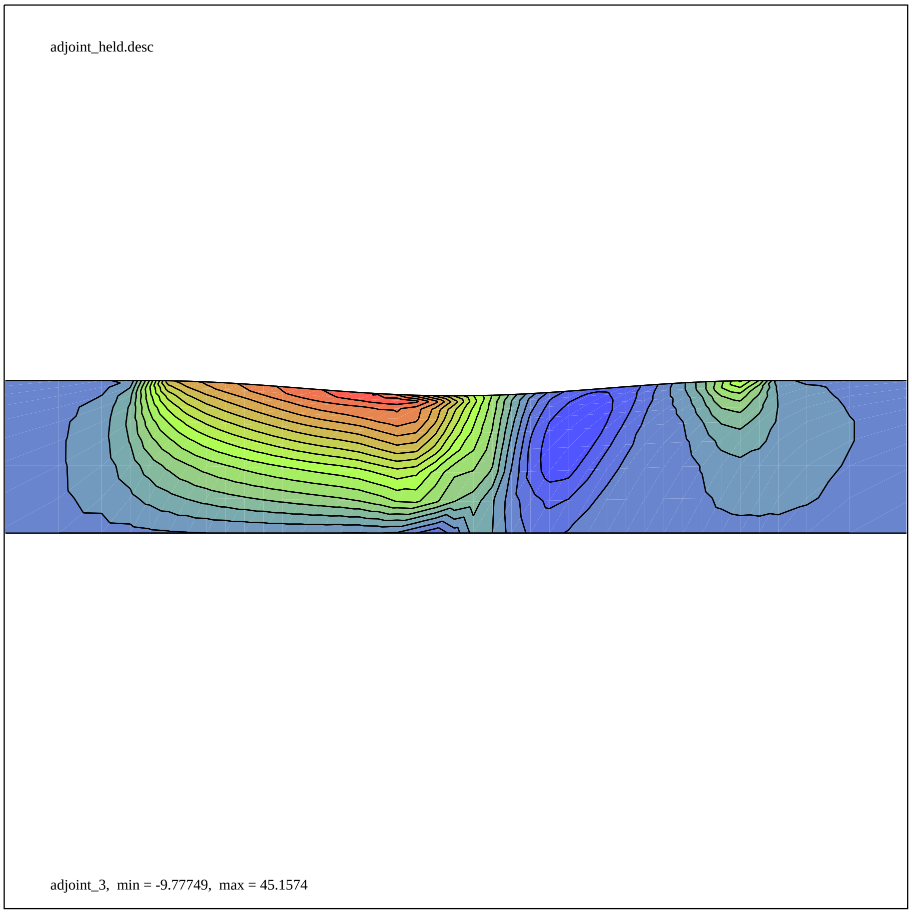

FIG. 14 – *Exact adjoint (second-order). Third component.*

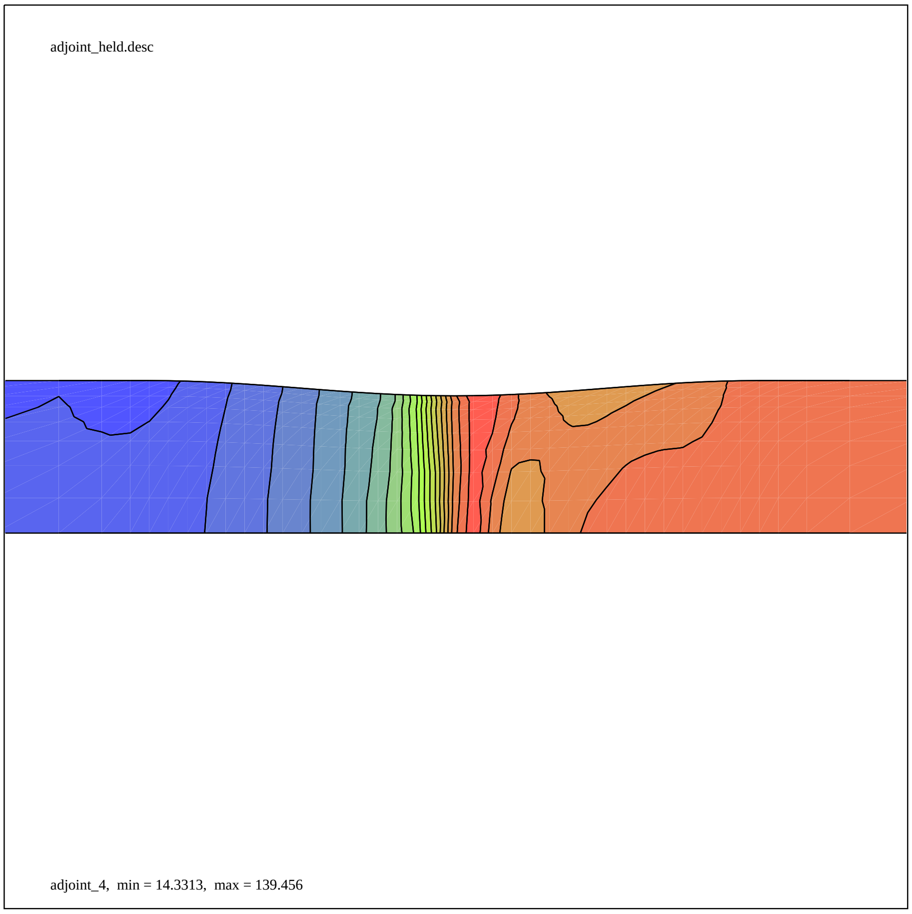

FIG. 15 – *Exact adjoint (second-order). Fourth component.*

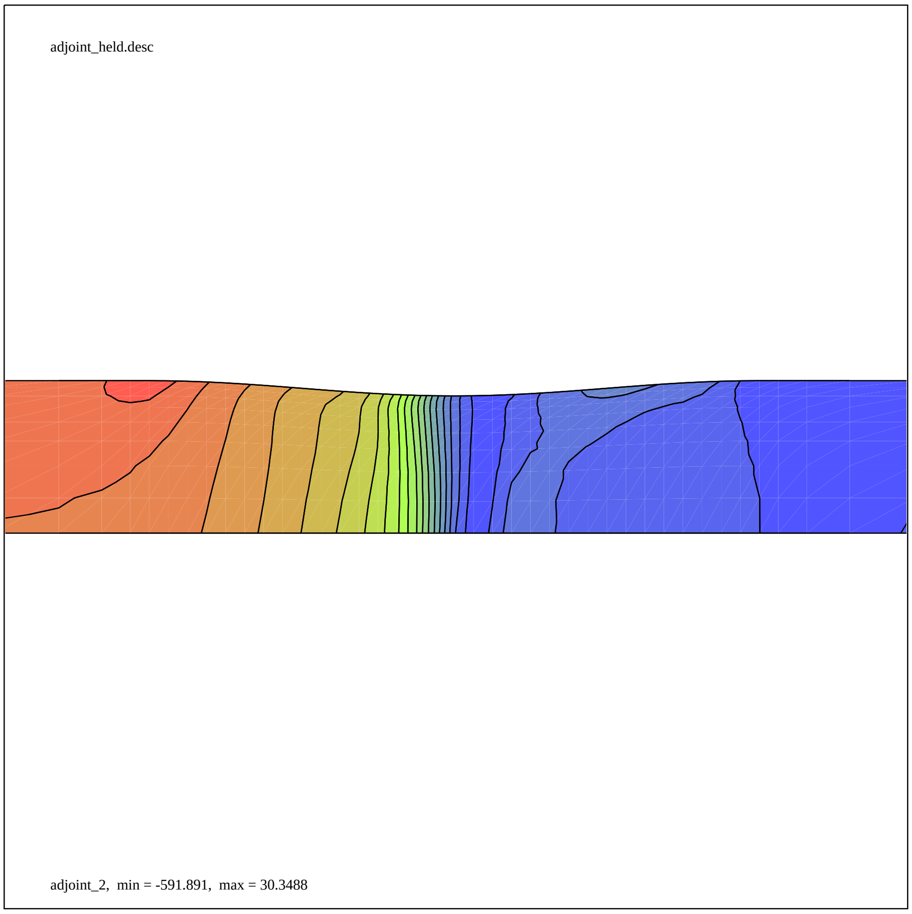

FIG. 16 – *Defect Correction simplified adjoint. Second component, To be compared with figures 10 and 12.*

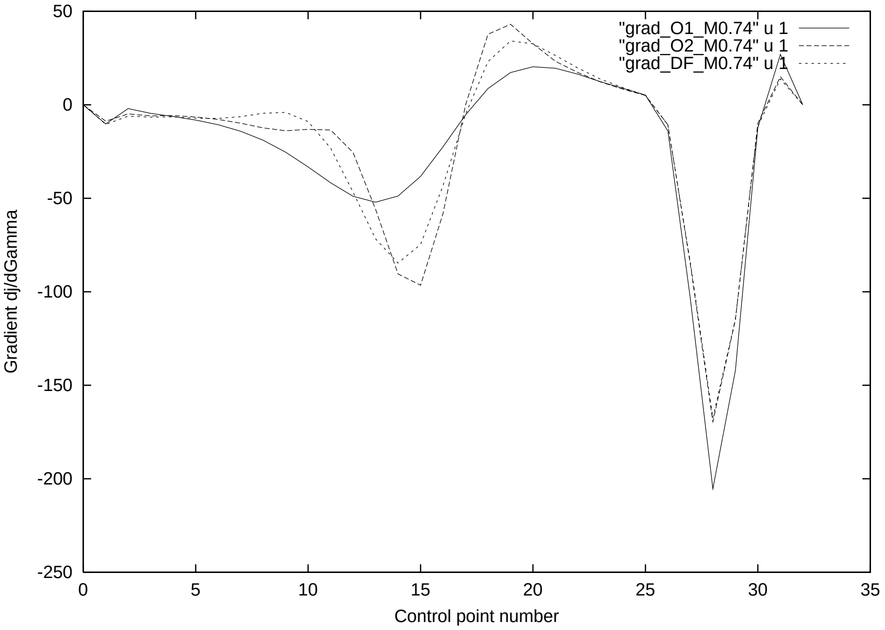

FIG. 17 – *Comparison of the different gradients. (1), continuous line, simplified gradient with the first order simplified adjoint depicted in Fig. 10. (2), dashes, exact gradient obtained with the exact adjoint depicted in Fig. 12. (3), dots, simplified gradient obtained with the Defect Correction simplified adjoint depicted in Fig. 16. For components 25 to 30, the two last gradients are almost identical.*