The 8th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2022
5-9 June 2022, Oslo, Norway

ALGORITHMIC DIFFERENTIATION FOR AN EFFICIENT CFD SOLVER

BRUNO MAUGARS[1], SÉBASTIEN BOURASSEAU[1], CÉDRIC CONTENT[1], BERTRAND MICHEL[1], BÉRENGER BERTHOUL[1], JORGE NUNEZ RAMIREZ[1], PASCAL RAUD[1] AND LAURENT HASCOËT[2]

[1] DAAA, ONERA, Université Paris Saclay, F-92322 Châtillon, France
e-mail: cedric.content@onera.fr

[2] INRIA Sophia Antipolis-Méditerranée, BP 93 F-06902 Sophia Antipolis, France

Key words: Algorithmic Differentiation (AD), High Performance Computing (HPC), Computational Fluid Dynamics (CFD)

Abstract. We illustrate the benefits of Algorithmic Differentiation (AD) for the development of aerodynamic flow simulation software. In refining the architecture of the elsA CFD solver, developed jointly by ONERA and Safran, we consider AD as a key technology to cut development costs of some derivatives of interest, namely the tangent, adjoint, and Jacobian. We first recall the mathematical background of CFD applications which involve these derivatives. Then, we briefly present the software architecture of elsA (Cambier et al. [12]) and the design choices which give it its HPC capability while highlighting how these choices strongly constrain the applicability of AD. To meet our efficiency requirements, we select the Source-Transformation approach to AD through the Tapenade tool which is justified by a series of experiments and benchmarks. Finally, we present results on large scale configurations.

1 INTRODUCTION

During the past decades, in the hope to prorogate Moore's law (Moore [28]), computer architecture has evolved from higher processor frequency to increasing number of cores with lower frequency and lower amount of memory. In the context of computational fluid dynamics (CFD) in general and more specifically for the elsA software, this implies to reimplement algorithms in order to make the most of these new architectures and to achieve the speedup that they enable.

To this end, an effort has been made to improve efficiency both in terms of CPU and memory costs of the explicit assembly part of the code. For instance, the work of I.Mary[27] shows the need to reduce memory access and to vectorize loops. In the following, we refer to this improvement as the High Performance Computing behavior.

Most fluid dynamics studies (such as computation of steady states, optimization, stability analysis, mesh adaptation, flow control,...) are based on the inversion of either the linearized Navier-Stokes (or RANS) equations or their adjoint. As those systems are sparse and extremely large (up to 100 billons degrees of freedom) we select algorithms which do not require the storage of the entire Jacobian matrix. Due to the stiffness of systems to solve, the iterative GMRES linear algebra solver (Saad [32]) appears to be a good candidate. This method only requires the

Jacobian-vector product to be evaluated. On the one hand, the left Jacobian-vector product (in use to solve the linearized system) can be easily approximated by means of a first order Taylor series expansion, as in the JFNK method (Knoll and Keyes [24]). This naturally inherits the good HPC behavior of the primal simulation code. Also, extensions to several numerical schemes are straightfoward. There are two main drawbacks of this method. The first one is the difficulty to obtain the left Jacobian-vector product with machine precision, which is key to the good convergence of Newton algorithm and stability analysis. Secondly it does not provide the discrete adjoint of the primal operator. On the other hand, right and left Jacobian-vector products can be both obtained by means of Algorithmic Differentiation (AD) (Griewank and Walther [20]) with machine precision, directly providing the discrete adjoint (see Kenway *et al.* [23]). It also propagates HPC performance of the primal code on the left Jacobian-vector product. However, the HPC behavior preservation in the adjoint is delicate, as adjoint AD deeply modifies the structure of the code. This paper will discuss our approach to address this problem with a special care on its efficiency.

In section 2, we present the mathematical formulation of various important CFD problems, highlighting the central role of the Jacobian, and more accurately the use of right and left Jacobian-vector products. Then section 3 presents the different approaches to Algorithmic Differentiation. Section 4 motivates the choice of Tapenade (Hascoët and Pascual [21]) for CFD needs. Then, the architectural implementation is discussed in section 5, followed in section 6 by a focus on the method we used to preserve the HPC behavior. Finally, we show a few experimental results in section 7 before concluding.

2 MATHEMATICAL FORMULATION OF SOME CFD PROBLEMS

Fluid dynamics, like other parts of classical physics, deals with conservation laws, which can be written as:

$$\partial_t Q + \nabla \cdot \boldsymbol{F} = S \tag{1}$$

with Q an m-vector of conservative quantities, $\partial_t Q$ the m-vector of its time derivative, $\boldsymbol{F}$ the tensor of conservative fluxes, $\nabla \cdot \boldsymbol{F}$ its divergence and S an m-vector of source terms independent on Q.

For convenience, we introduce the residual of the space operator as:

$$\mathcal{R} = \nabla \cdot \boldsymbol{F} - S. \tag{2}$$

Then (1) becomes:

$$\partial_t Q + \mathcal{R}(Q) = 0 \tag{3}$$

In the following, we will show that many useful analyses of these equations boil down to the resolution of what we will call either *direct equations* or *adjoint equations*, involving Jacobian-vector products. Moreover, an approximation of the full Jacobian matrix may be required for preconditioning.

Direct equations

Nonlinear dynamical systems tend to admit a number of different equilibria which can take

the form of fixed points, periodic orbits, torus or strange attractors (Berger *et al.* [9]). The first equilibria that one would like to access to form the phase space of a system (i.e. to characterize and thus to understand it) are its fixed points. The method used to compute it is also the milestone of the other equilibria detection and the first step of branch continuation techniques (Tuckerman and Barkley [37]).
Obtaining a fixed point of equations which describe evolution laws is equivalent to computing a steady state. Hence, equation (1) becomes:

$$\partial_t Q = 0 \Leftrightarrow \mathcal{R}(Q) = 0 \tag{4}$$

A general procedure to compute a fixed point of this kind of system is to use the first order Newton method which consists in iterating over a sequence of linear systems:

$$\mathcal{J}\Delta Q = -\mathcal{R} \tag{5}$$

where $\Delta Q^n = Q^{n+1} - Q^n$ is the unknown and $\mathcal{J}$ the Jacobian matrix of the space operator $\mathcal{R}$:

$$\mathcal{J} = \frac{\partial \mathcal{R}}{\partial Q} \tag{6}$$

As an initial guess of the flow field is generally far from solution, the Newton algorithm is not guaranteed to show a fast convergence behavior, or even to converge at all. A classical way used to overcome this difficulty is to solve an implicit Euler scheme:

$$(\frac{1}{\Delta t}I + \mathcal{J})\Delta Q = -\mathcal{R} \tag{7}$$

where $\Delta \mathrm{t} = \mathrm{t}^{\mathrm{n+1}} - \mathrm{t}^{\mathrm{n}}$. This scheme used in conjunction with a local time step and a pseudo-transient continuation technique as described in (Crivellini and Bassi [13]) can be seen as a relaxed-Newton algorithm and leads to the same result.

Another key to understand a flow configuration is to well capture flow motion. Analysis of unsteady features requires discretizing the time operator. While explicit schemes provide accurate temporal resolution for PDE like equation (1), the time step size is dictated by stability constraints of the algorithm rather than by the frequency content of the physical phenomenum under consideration. A way of relaxing stability constraints consists in using an implicit time integration method. The discrete time scheme for equation (3) reads:

$$\mathcal{N}^{n+1} = \frac{DQ^{n+1}}{\Delta t} + \mathcal{R}(Q^{n+1}) = 0 \tag{8}$$

In the large family of implicit Runge-Kutta methods, the class of linearly implicit Rosenbrock-type Runge-Kutta schemes (Rosenbrock [31]) is of special interest because such methods, being linearly implicit, require to solve only linear systems in the stages within each time step, i.e., the Jacobian matrix needs to be assembled and factorized only once per time step. The performance of all the aforementioned time integration schemes have been recently investigated, and, according to the numerical comparison presented in Bassi *et al.* [8]. Rosenbrock schemes turn out to be an appealing choice both in terms of accuracy and efficiency.

Despite of the good behavior of the above method, multistep Backward Differentiation Formulae (BDF) (Curtiss and Hirschfelder [14]) are often used as implicit time integrators, due to ease of implementation and robustness. At each time step, these schemes require to solve several non-linear systems of equations, a task that can be efficiently performed, for example, by means of the (quasi-)Newton method. Linearization of the residual leads to the following implicit scheme

$$\frac{d\mathcal{N}}{dQ}\Delta Q = -\mathcal{R} \tag{9}$$

Considering the time discretization expressed for second order as:

$$\frac{DQ^{n+1}}{\Delta t} = \frac{3\Delta Q^{n} - \Delta Q^{n-1}}{2\Delta t} \tag{10}$$

one can write (9) as:

$$(\frac{3}{2\Delta t}I + \mathcal{J})\Delta Q = -\mathcal{R} \tag{11}$$

To wrap up about *direct equations*, equations (5) and (11) are just two emblematic cases that require inversion of a linear equation of type $AV = b$ at each iteration, where A is basically the Jacobian of some known operator. Matrix-free inversion methods will only require the Jacobian-vector product ($\mathcal{J}\ V$).

Adjoint equations

Powerful tools that one might access to optimize a flow configuration either for its understanding (Luchini and Bottaro [26]) or for reducing its loss (Dwight [16]) are based on adjoint equations. As recalled by Luchini and Bottaro [26], the *"adjoint formulation is useful when one is seeking to obtain one or a few outputs of a system for a wide range of possible inputs. There are several such cases in fluid mechanics (and other disciplines),[...] but the greatest advantage is obtained in optimization. In fact, the typical optimization problem has a single objective function (possibly combining multiple objectives through suitable weights) that has to be minimized or maximized with respect to a large number, or even a continuum, of input variables: a perfect application for adjoints!"* A detailled definition and some uses of the adjoint equation can be found in [26, 16, 25].

Any type of optimization process corresponds to the minimization of a scalar objective function $\mathcal{I}_n$, subject to the constraint that the discrete flow equations and boundary conditions are satisfied (Giles and Pierce [19]). The adjoint vector Λ is the solution of

$$\mathcal{J}^T|_{\bar{Q}}\ \Lambda = \frac{\partial \mathcal{I}_n}{\partial Q} I \tag{12}$$

Matrix-free solution methods will only require the transposed Jacobian-vector product ($\mathcal{J}^T\ \Lambda$).

Resolution

Solving *direct* and *adjoint* equations when applied to CFD involves sparse and extremely large matrices in a parallel distributed environment. Thus to access to their inversions without assemblying and storing the entire exact Jacobian matrix, iterative linear algebra methods (as

GMRES [32]), which only require the Jacobian-vector product, are considered. Unfortunately, to ensure the good convergence of those matrix-free solvers, a preconditionner is required. Many preconditionners need at least an approximate, possibly without cross-processor coupling, of the first order Jacobian matrix $\tilde{\mathcal{J}}$.

To sum up, we have described a number of situations in our CFD applications where one must have access to derivatives of selected operators implemented/present in the code. More specifically, for some function $F : X \in \mathbb{R}^n \mapsto Y \in \mathbb{R}^m$ that we can identify with a portion of the code `P`, and calling $\mathcal{J}$ the Jacobian of F, we may want to obtain the exact Jacobian-vector product $(\mathcal{J}\ V)$, the exact transposed Jacobian-vector product $(\mathcal{J}^T\ \Lambda)$, and an approximated Jacobian matrix $(\tilde{\mathcal{J}})$, possibly without its cross-processor components.

3 ALGORITHMIC DIFFERENTIATION

There are several approaches at hand to obtain these derivatives. Finite Differences (as the one described for JFNK [24]) are maybe the simplest approach, but it introduces truncation approximation into $\mathcal{J}$, which we want to avoid. Manual discretization-then-implementation of the differentiated equations gives derivatives that are obviously more accurate, at a huge development cost that we want to avoid too. Moreover, the discretization errors in computing $\mathcal{J}$ may be inconsistent with those in computing F, leading to possible convergence problems. Algorithmic Differentiation (AD) is yet another approach that takes as input the implementation `P` of the discretized F, which we call the "primal" code, and turns it into another program/code that computes the derivative $\mathcal{J}$ of F, or one of its projections $(\mathcal{J}V)$ or $(\mathcal{J}^T V)$. AD introduces no approximation other than the one already present in `P`. Other advantages of AD are that it can be automated to a large extent, and that with due care, the efficiency of the differentiated code compares very well with that of more manual approaches.

AD identifies the primal code `P` with a sequence of assignment instructions $\{I_1; I_2; \dots I_p;\}$, each I_k implementing a simple function f_k. Thus, `P` computes $F = f_p \circ f_{p-1} \circ \dots \circ f_1$. Call for short $v_0 = X$ and $v_k = f_k(v_{k-1})$ for each k.

The simplest form of AD, *Tangent* AD, computes $\dot{Y} = F'(X) \times \dot{X}$ for a given input direction $\dot{X}$, i.e.

$$\dot{Y} = f'_p(v_{p-1}) \ \times \ f'_{p-1}(v_{p-2}) \ \times \ \dots \ \times \ f'_1(v_0) \ \times \ \dot{X} \ .$$

Since $\dot{X}$ is a vector, efficient evaluation must be done from right to left, i.e. in the same order as the primal code. We define $\dot{v}_0 = \dot{X}$, then each primal instruction I_k is immediately accompanied by a tangent instruction I'_k that computes $\dot{v}_k = f'_k(v_{k-1}) \times \dot{v}_{k-1}$. The second column of Fig. 1 shows the tangent code for one particular I_k.

Another form of AD, *Adjoint* AD, computes $\overline{X} = \overline{Y} \times F'(X)$ (or equivalently $F'(X)^T \times \overline{Y}^T$) for a given output weighting $\overline{Y}$, i.e.

$$\overline{X} = \overline{Y} \ \times \ f'_p(v_{p-1}) \ \times \ f'_{p-1}(v_{p-2}) \ \times \ \dots \ \times \ f'_1(v_0) \ .$$

Efficient evaluation must be done from left to right, i.e. in the reverse of the primal computation order. We define $\overline{v}_p = \overline{Y}$ and then for all k, $\overline{v}_{k-1} = \overline{v}_k \times f'_k(v_{k-1})$. The full primal sequence must be executed before the backward sequence of derivatives computations. The third column of Fig. 1 illustrates the adjoint code for one particular I_k. Notice that I_k and the two assignments

of $\overline{I}_k$ are not consecutive any more, and the value of `b` used in $\overline{I}_k$ is its value *before* evaluation of I_k.

Primal assignment	Tangent AD	Adjoint AD
(I_k) `b = 2*sin(a)*b+1;`	($\dot{I}_k$) $\dot{\texttt{b}}$ `= 2*cos(a)*b*`$\dot{\texttt{a}}$ `+ 2*sin(a)*`$\dot{\texttt{b}}$`;` (I_k) `b = 2*sin(a)*b+1;`	(I_k) `b = 2*sin(a)*b+1;` ... ($\overline{I}_k$) $\overline{\texttt{a}}$ `=` $\overline{\texttt{a}}$ `+` `2*cos(a)*b*`$\overline{\texttt{b}}$`;` ($\overline{I}_k$) $\overline{\texttt{b}}$ `= 2*sin(a)*`$\overline{\texttt{b}}$`;`

Figure 1: Tangent AD and Adjoint AD of an individual assignment

It is worth noting that the most important form of derivatives that we use in section 2, and the one likely most expensive to compute, is the "transposed Jacobian vector product" $\mathcal{J}^T\ \Lambda$, where $\mathcal{J}$ is the Jacobian of F. Most often, the F's we are dealing with have very large input dimension n whereas result dimension m is 1 or just a few. In that case, it is well known that efficient computation of $\mathcal{J}^T\ \Lambda$ must be done backwards with respect to evaluation order of F, which in our context means using adjoint AD. The strength of adjoint AD is that it produces a code that computes $\mathcal{J}^T\ \Lambda$ at a cost that is proportional to m, but that is *independent* of the input dimension n. However, control-flow reversal and more importantly data-flow reversal have a cost, generally in memory. The art of adjoint AD is about implementing control-flow and data-flow reversal on every possible construct of the application language, with a reasonable memory cost for data-flow reversal.

We motivate the existence of two main families of AD tools in the light of this sophisticated adjoint AD. In all cases, everything starts with a so-called *forward sweep*, which is mostly an evaluation of the primal code for F, that sets up the reference control-flow and data-flow that must be played in reverse by the coming derivative computation (the *backward sweep*).

- One strategy is to instrument the forward sweep so that it writes a tape of the full sequence of arithmetic operations performed, complete with operand addresses and values, arithmetic operation, and result address. The following backward sweep is a special code that reads the tape backwards to propagate the derivatives. As this is most easily done on application languages that provide overloading, this is generally called Operator-Overloading AD (OO-AD). OO-AD easily handles "exotic" constructs of the application language, as they all boil down to a uniform structure on the tape. The code of the backward sweep is written once and for all, independent from the primal code. It can be quite sophisticated. On the other hand, the tape grows rapidly, as it stores not only values but addresses and operations.

- Another strategy is to produce a specialized backward sweep for each primal code, reproducing its control structure, only in the reverse direction. The forward sweep must still be instrumented, but storing only (a subset of) the intermediate values, yielding a much smaller tape. This is called Source-Transformation AD (ST-AD). ST-AD requires complex tools, able to take in the whole primal code and generate a backward sweep code in which

control-flow is reversed. It must also fill this backward sweep with the adjoint counterpart of each individual assignment of the primal code. Finally, it must orchestrate the storage of intermediate values by the forward sweep and their retrieval in reverse order by the backward sweep. Consequently ST-AD tools are often lagging behind OO-AD when it comes to handling the latest and newest language constructs. For instance, there is still no real ST-AD tool for C++. On the other hand, the resulting backward sweep is exposed to the compiler, yielding better performance, and the tape is smaller. ST-AD tools, having to build an internal representation of the full primal code, are able to run global software analysis that lead eventually to a more efficient adjoint code.

To sum up, OO-AD and ST-AD are just two different implementations of the same model of derivative computation. In particular, they will return the same derivatives in the end, with the same approximation errors. Differences lay in run-time and memory consumption, and sometimes in limitations of the accepted language for the primal code. In section 4 we will compare performance of OO-AD and ST-AD for the kind of primal codes that we consider.

From a software engineering point of view, the *automatic* aspect of AD is appealing, reducing the development and maintenance time. It is therefore recommended that the generated derivative code is not modified by hand afterwards. In particular, efficiency adaptations of the derivative code, specific to the HPC behavior and the target parallel architecture, should not be applied after AD, but rather before or during AD. Two cases arize:

- if the primal code for F already embeds performance features, such as parallelization directives or calls, the AD tool should preserve these features and use them consistently in the generated code for F'. Research on AD models has made this possible in several cases, like tangent and adjoint AD of point-to-point and global communication primitives of parallel distributed environnements such as MPI [38].
- if the AD process has introduced new features that were not present in the primal code, such as loops on differentiation directions in the so-called *vector* mode, then the AD tool must produce code adapted to the target architecture, specified e.g. with directives in the primal code.

At the time of design of our proposed architecture of elsA, it is essential to discuss with AD tools developers, to choose an HPC approach that the AD tool can preserve in the adjoint code. As we will show in section 5, this sometimes leads to additional developments in the AD tool itself.

4 CHOICE OF ST-AD TOOL

As discussed in section 2, a CFD solver needs to provide a tangent and an adjoint modes to treat, for examples, implicit resolution, shape optimization, goal-oriented mesh adaptation, sensibility or stability. Furthermore, software architecture must provide an HPC behavior adapted to new hardware architecture. In particular, domain decomposition, vectorization directive or pragma in the differentiated code must be consistent with the primal implementation.

As experience shows that it is hard to maintain a handmade differentiated code, we will rely on AD tools to rather generate it. A first question is to determine which of OO-AD and ST-AD

best fits our needs and constraints. To answer this question, two of the most used AD tools software in CFD community are compared: CoDiPack (OO-AD) [1] and Tapenade (ST-AD) [21].

In order to compare the two strategies, two toy problems of increasing complexity are considered: 2nd order finite differences which is one of the simplest operator that includes stencil combinations, and Roe flux that involves non-linear operations. The mesh has $256 \times 256 \times 24$ cells and 200 iterations are made. Our comparison criteria are CPU and memory costs. Direct (13), tangent (14) and adjoint (15) computations are made for each AD-tool.

$$x \rightarrow \Psi(x) \tag{13}$$

$$(x, \delta x) \rightarrow \left(\frac{\partial \Psi(x)}{\partial x}\right) \delta x \tag{14}$$

$$(x, \delta x) \rightarrow \left(\frac{\partial \Psi(x)}{\partial x}\right)^T \delta x \tag{15}$$

To fairly compare Fortran/Tapenade and C++/CoDiPack, the same code optimization has been performed to ensure that the comparison in terms of CPU and memory costs is relevant.

Table 1 gives the CPU cost and the memory consumtion for the direct computation of 2nd order finite differences (FDO2) and Roe flux (Roe) in each implementation. These results (T_D and M_D) will be used as the baseline to form CPU slowdown and memory increase for tangent (T_T/T_D and M_T/M_D) and adjoint (T_A/T_D and M_A/M_D) computations. Table 2 shows no significant differences between OO-AD and ST-AD on tangent AD, both in terms of CPU and memory. On the contrary, table 3 shows factors up to 5 (in CPU) and 10 (in memory) on adjoint AD.

	Direct (C++)		Direct (Fortran)	
	T_D (s)	M_D (MB)	T_D (s)	M_D (MB)
FDO2	0.41	16	0.39	16
Roe	5.38	120	5.37	120

Table 1: CPU and memory cost comparison for direct computation

	Tangent (C++)		Tangent (Fortran)	
	T_T/T_D	M_L/M_D	T_T/T_D	M_L/M_D
FDO2	1.20	2.00	1.28	2.00
Roe	0.96	1.53	1.15	1.53

Table 2: CPU and memory cost ratios comparison for tangent computation

Those spectacular results on adjoint mode computation must be confirmed on a full CFD application. Indeed, these initial comparisons are limited to the flux divergence computation

	Adjoint (C++)		Adjoint (Fortran)	
	T_A/T_D	M_A/M_D	T_A/T_D	M_A/M_D
FDO2	7.53	6.75	1.49	2.00
Roe	7.42	16.08	1.75	1.53

Table 3: CPU and memory cost ratios comparison for adjoint computation

and do not take into account the other CFD software components, such as boundary conditions, initialization, pre-processing, post-processing, etc.

To experiment further, on full codes, we choose SU2 [2], an open-sources CFD solver using CodiPack (OO-AD) for differentiation and we retain the elsA solver with Tapenade (ST-AD).

In order to investigate the memory footprint, we run both direct and adjoint simulations of ONERA M6 wing (Schmitt and Charpin [33]) with about 580 000 nodes. In agreement with Gauger communication (Albring *et al.* [5]), there is a memory factor about 7 between direct (3.3 GB) and adjoint (21.7 GB) with SU2/CodiPack whereas this factor is only 1.5 (2.7 GB vs 4.1 GB) for elsA/Tapenade.

Based on these results and the fact that modern scalar hardware architectures provide a memory amount up to 4 GB per core, the option that best suits our needs for CFD adjoint computation appears to be ST-AD, e.g. with Tapenade. From now on, as we will concentrate on the ST-AD tool, we will refer to it as the “AD tool”.

5 RESULTING IMPLEMENTATION

As a reminder, elsA [12] is a large CFD simulation software developed in collaboration between ONERA and Safran. elsA deals with internal and external aerodynamics from the low subsonic to the high supersonic flow regime. In addition, elsA interoperates with multidisciplinary simulation platforms in order to integrate CFD simulation capabilities into industrial workflows. In the last years, an effort based on the work of I. Mary [27] has been made to challenge the two limiting factors of CFD applications, namely their CPU and memory costs. To guarantee this improvement on any types of computation (direct, tangent and adjoint) and to cope with a major limitation of ST-AD tools (they work at computation level, e.g. Fortran code but not at the driver level, e.g. C++ or Python code), we organize the solver as an chain of operators specified by user choices. Each operator (see Fig. 2) is described by its inputs, its outputs and a low-level implementation. As the coarse-grain part of our HPC model is based on domain decomposition, each operator must also describe the support of inputs and outputs (i.e. on vertices, edges, facets or cells) and how computed values may become inconsistent on overlapping halos, therefore triggering communication.

In CFD solvers, the main variables are the conservatives Q and the mesh X. We use the AD tool to generate the code for the derivative operators with respect to Q, X and any other parameter P. For each operator under consideration, Tapenade was able to generate the tangent (”computeLin”), adjoint (”computeAdj”), and approximated Jacobian (”computeJac”) code, from the primal code of the operator. Consequently, each operator now comes with its differentiated functions of these three kinds (see Fig. 3). The Jacobian was produced using the so-called *vector tangent* mode, that computes the exact Jacobian of the first order spatial operator $\mathcal{R}$ for

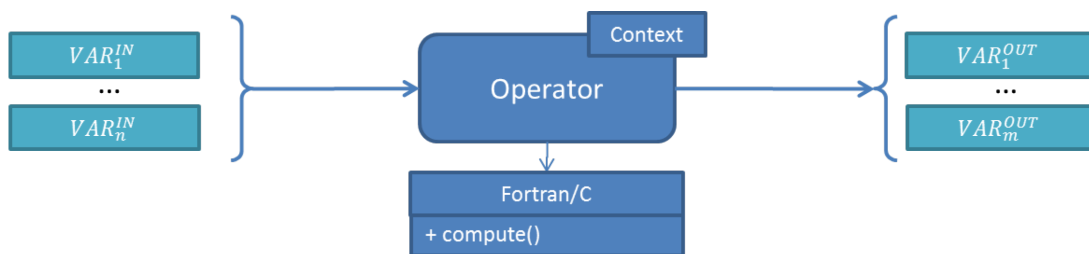


Figure 2: Operator description

each processor in a parallel distributed environnement, without its cross-processor components. The resulting overall implementation has the form of a chain of operators that has to be inversed for the adjoint mode.

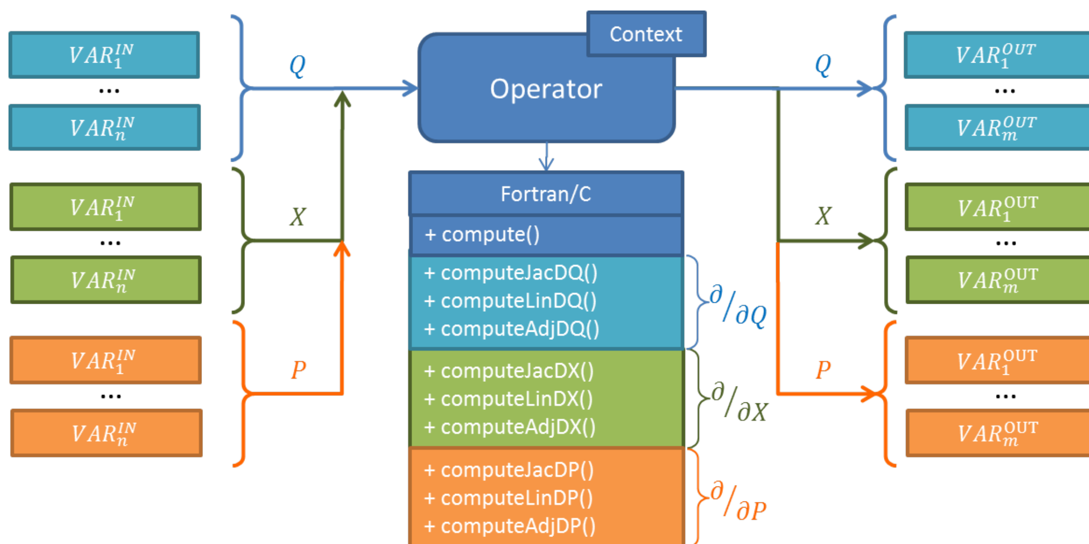


Figure 3: Operator description with code generated by ST-AD

6 PRESERVING THE HPC BEHAVIOR

Recalling section 3, one main requirement for our applications is that the AD tool should preserve the HPC behavior. In particular, we will now discuss the efficiency question related to loop vectorization of the generated adjoint operator.

The AD tool defines a pragma (`II-LOOP` for "Independent Iterations") that the user may set before a loop to indicate that the loop's iterations order can be changed freely. This pragma can improve the adjoint code dramatically, as illustrated in Fig. 4. Instead of a *forward sweep* containing essentially the source loop, followed by a *backward sweep* containing the reversed

loop, the adjoint code features only one loop in which each source iteration is immediately followed by its own adjoint iteration. In this unique loop of the adjoint code, each iteration starts with its own *forward sweep* immediately followed by its own *backward sweep*. The main benefit is that the peak memory size used to store intermediate numerical values is reduced, as each iteration pushes only its own temporaries (e.g. variable `right`) and then immedialely pops them in its *backward sweep*. An additional benefit is that we get one loop instead of two, reducing loop overhead and probably improving locality. It is thus recommended to place an

source loop	**adjoint loop**	**static taping**
`!$AD II-LOOP`	`!DIR$ IVDEP`	
`!DIR$ IVDEP`	`DO i=2,n`	
`DO i=2,n`	`left = a(i-1)`	
`left = A(i-1)`	`right = a(i+1)`	
`right = A(i+1)`	`CALL PUSHREAL8(right)` ....*or*⟶	`ad_save = right`
`right = left*SIN(right)`	`right = left*SIN(right)`	
`mid = (left*right)/2.0`	$\overline{mid}$ = $\overline{b}$(i-1) + $\overline{b}$(i+1)	
`B(i-1) = B(i-1) + mid`	$\overline{left}$ = right*$\overline{mid}$/2.0	
`B(i+1) = B(i+1) + mid`	$\overline{right}$ = left*$\overline{mid}$/2.0	
`ENDDO`	`CALL POPREAL8(right)` ....*or*⟶	`right = ad_save`
	$\overline{left}$ = $\overline{left}$ + SIN(right)*$\overline{right}$	
	$\overline{right}$ = left*COS(right)*$\overline{right}$	
	$\overline{a}$(i+1) = $\overline{a}$(i+1) + $\overline{right}$	
	$\overline{a}$(i-1) = $\overline{a}$(i-1) + $\overline{left}$	
	`ENDDO`	

Figure 4: A primal loop with pragma `II-LOOP` (left), its adjoint loop (middle), and a modified adjoint loop with static taping of intermediate values (right)

`II-LOOP` on every loop of the primal operator that deserves it. This is obviously the case for loops with no loop-carried dependencies, but we may go further: the duality that is at the heart of adjoint AD is such that a variable *read* in the primal code will cause an *increment* of its adjoint variable in the adjoint backward sweep. Symmetrically a primal *increment* of a variable will trigger a single *read* of its adjoint variable. Because successive increments can be exchanged (provided they are atomic in case of parallel execution), we can extend the `II-LOOP` pragma to loops whose *only* loop-carried dependencies are between *increments* of variables. Consequently, classical gather-scatter loops can receive the `II-LOOP` pragma.

On its side, the primal code of operators is already equipped with `IVDEP` pragmas, that tell the compiler about vectorizable loops. Unfortunately, there is no clear relation between `IVDEP` and `II-LOOP` pragmas. We can build `IVDEP` loops that contain loop-carried *anti*-dependencies and therefore are not `II-LOOP`. Conversely we can build a gather-scatter loop, deserving an `II-LOOP`, but that cannot be marked `IVDEP` because of loop-carried dependencies between increments. All we can recommend at that stage is that an `IVDEP` loop is very likely to also be an `II-LOOP`, and should be examined for that in priority.

Now taking a closer look at the adjoint loop itself, we notice we cannot in general mark as `IVDEP` the adjoint of a loop, be it `IVDEP`, `II-LOOP`, or both. In many cases we can (see for

instance the adjoint loop of Fig. 4), because the dependency distance is explicit (`i+1` and `i-1` versus `i`). However when the dependency is more intricate, e.g. using indirection arrays, the `IVDEP` is forbidden.

The issue exists already on the primal code. elsA generally deals with it by coloring, as shown on the left of Fig. 5. The loop has been rescheduled as two nested loops, such that the inner loop guarantees that two different `i` never get the same value of `q`. The inner loop is therefore `IVDEP`, while the outer loop is not. In passing, notice that both loops are `II-LOOP`, yielding the adjoint loop nest on the right. Which leads to the following question: can we place an `IVDEP` on the inner adjoint loop? The answer is no in general because we have no guarantee that two different `i` always get different `p`. This leads us to our recommendation that the coloring scheme initially devized to guarantee independence of the write operations (i.e. `Y(q)=...`) be refined to also guarantee independence of the read operations (i.e. `X(p)`). This refinement is indeed useless for the primal loop, and it may even slightly degrade performance, but it allows us to place a `IVDEP` on the adjoint inner loop, improving its performance significantly.

source loop	**adjoint loop**
`!$AD II-LOOP`	`DO pp=1,nPack`
`DO pp=1,nPack`	`first = packIdx(pp)`
`first = packIdx(pp)`	`last = packIdx(pp+1)`
`last = packIdx(pp+1)`	`!DIR$ IVDEP`
`!$AD II-LOOP`	`DO i=first,last-1`
`!DIR$ IVDEP`	`p = indGather(i)`
`DO i=first,last-1`	`q = indScatter(i)`
`p = indGather(i)`	`temp = X(p)`
`q = indScatter(i)`	`Y(q) = Y(q) + temp*4`
`temp = X(p)`	$\overline{\texttt{temp}}$= 4*$\overline{\texttt{Y}}$(q)
`Y(q) = Y(q) + temp*4`	$\overline{\texttt{X}}$(p) = $\overline{\texttt{X}}$(p) + $\overline{\texttt{temp}}$
`ENDDO`	`ENDDO`
`ENDDO`	`ENDDO`

Figure 5: Adjoint loop when using coloring

We must not neglect the last source of dependencies in the adjoint loop i.e. those coming from the PUSH/POP mechanism on intermediate values of primal variables. Actually, the compiler refuses to vectorize the loop in the 2nd column of fig. 4, and rightly so since `PUSHREAL8` and `POPREAL8` are external procedures that use an underlying global stack. We do know that the related dependencies are indeed local to each loop, but we need to make this obvious to the compiler. To this end we developed an alternative save/restore through local temporary variables that needs no stack nor subroutine calls. It is triggered by a command-line option of the AD tool (`staticTape`). In the future, it will be refined to apply selectively to chosen loops.

With the (`staticTape`) option, the AD tool tries whenever possible to save intermediate values into additional local variables of the enclosing differentiated procedure, as shown in the 3rd column of fig. 4. This mechanism has limitations: each intermediate value of possible inner loops requires a storage array dimensioned after the iteration space of the loop. Inner loops with dynamic iteration length (e.g. `while` loops) forbid this. Also, this mechanism may end up

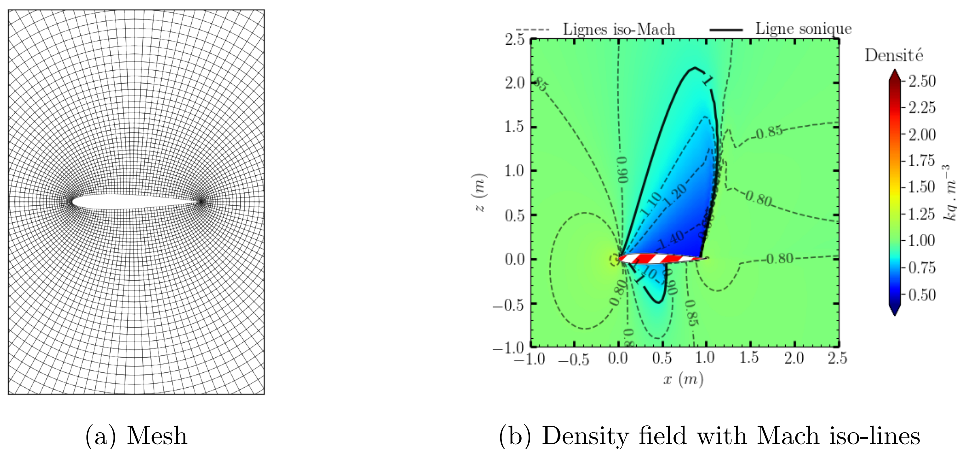
reserving more storage space than actually needed, in case of conditionals. Those limitations are similar to those of the classical compiler transformation known as *Static Single Assignment* (SSA).

Performance-wise, adjoint code with the `staticTape` option behaves quite well. Applied to elsA, the additional memory consumption (on the execution stack) is marginally larger than the memory used by the previously existing `PUSH/POP` calls (on the heap). As management and access for heap memory can be slower than for stack memory, the `staticTape` option slightly improves speed. More importantly, this replaces calls to procedures `PUSH/POP` with simple assignments that are significantly faster. In our context, using the `staticTape` option is the last brick that allows the vectorizing compiler to use the `IVDEP` pragma on the adjoint loop, improving speed dramatically as will be shown in the next section.

7 RESULTS

Duality tests on a simple CFD test case : the NACA0012 case

The NACA0012 airfoil [33] is a well-known simple CFD case that allows to test new functionalities (like duality for example) of a solver without spending lots of time to prepare and solve it. This configuration is well documented in the litterature and permits to compare results to the other already published.



(a) Mesh

(b) Density field with Mach iso-lines

Figure 6: NACA0012 Euler transonic configuration (M=0.85/AoA=2°)

The principle of this test is to make sure that the generated tangent and adjoint modes and their operators chain are consistent and can be written as:

$$\forall(\lambda, \delta Q): \underbrace{\left(\lambda \frac{\partial \mathcal{R}}{\partial Q}\right)}_{\text{AdjCompute}} \underbrace{\delta Q}_{\text{TestVector}} = \underbrace{\lambda}_{\text{TestVector}} \underbrace{\left(\frac{\partial \mathcal{R}}{\partial Q} \delta Q\right)}_{\text{LinCompute}} \tag{16}$$

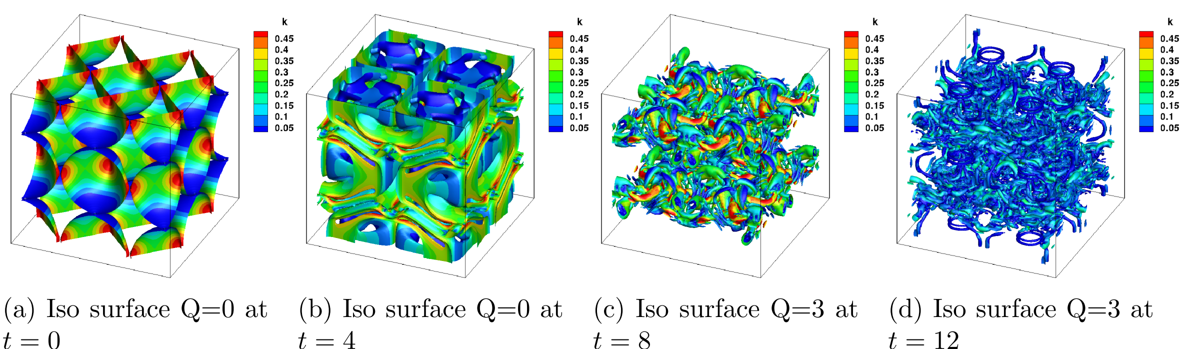
where λ is an adjoint vector and δQ a tangent vector.

To verify the precision obtained on this equality, the NACA0012 airfoil is tested for an Euler transonic flow with an angle of attack of 2° and a Mach number of 0.85 on an unstructured mesh

(see Fig.6). Sequential and parallel computations with first and second order spatial schemes have been tested. All the resulted duality errors are about $1e^{-15}$.

Performance (CPU elapsed) on a full periodic problem : the Taylor Green Vortex

Initialy introduced by Taylor and Green [36], the viscous Taylor-Green Vortex (TGV) has been extensively studied by Brachet et al [11, 10]. This flow is one of the simplest system in which one can study the generation of small scales and the resulting turbulence. A three-dimensional vortex is set as an initial condition for 3D-computation. Because of vortex-stretching and vortex-tilting mechanisms, the vortex breaks up, giving birth to smaller and smaller structures. At finite Reynolds number, the kinetic energy is transferred from larger to smaller scales and dissipated by the smallest one; the test case gives thereby a simple model of the energy cascade (Fig. 7) and is often used as a good milestone to assess the applicability of numerical schemes to Large Eddy Simulation [3, 18, 17, 22].



(a) Iso surface Q=0 at $t = 0$

(b) Iso surface Q=0 at $t = 4$

(c) Iso surface Q=3 at $t = 8$

(d) Iso surface Q=3 at $t = 12$

Figure 7: TGV: Iso surface of the Q criterion colored by k. The figure shows phases of the vortex break-up.

In this study we verify the ability of our solver to find the trivial fixed point in which all of the kinectic energy has been dissipated.

This test case has been run on a 14 cores/2 sockets Intel Broadwell processor. Figure 8 shows the CPU elapsed time by cell (noted τ herafter) to evaluate $\mathcal{R}$ (green), or $\mathcal{R}$ and $\mathcal{J}\mathcal{V}$ (orange) or $\mathcal{R}$ and $\mathcal{J}^T\mathcal{V}$ (purple). To check the strong scalability of each computational modes on a single processor, TGV test case is run from 1 to 14 cores. The nearly constant magnitude of τ confirms the good parallelization of the solver for each computation mode, i.e. the code generated by Tapenade preserve the HPC behavior of the primal code. The factor between direct and tangent modes can be explained by the increase of computational fields (In tangent mode, solvers must also compute linearized fields). The discrepancy between tangent and `staticTape` adjoint modes comes from the overhead computations required for control-flow and data-flow reversal, as reported in [20] . Using `staticTape` instead of `dynamicTape` mode (removal of push/pop, cf. part 6) yields a factor about 1.5 in terms of CPU.

Industrial test-case : the NASA rotor 37

The NASA 37 transonic rotor (Moore and Reid [29]) is a well known turbomachinery test

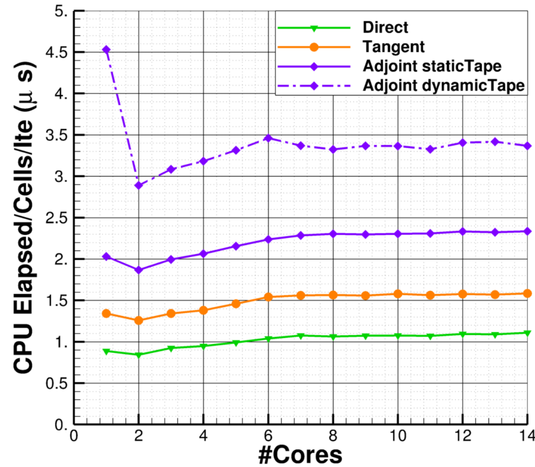


Figure 8: TGV : performances for direct, tangent and adjoint computations

case. This test case has been computed by numerous authors [4]. Experimental data were obtained at various measurement planes using both Laser Doppler Velocimetry and classical rake measurements of pitchwise-averaged total pressure and pitchwise-averaged total temperature (Strazisar [34]). The rotor has 36 blades, a nominal speed of 17188 rpm, the nominal tip-clearance gap taken into account for all computations is equal to 0.4 mm. The upstream stagnation pressure and temperature are respectively equal to 101325 Pa and 288.2 K. The measurement uncertainties are reported by Suder [35]. This test case presents a shock-wave/boundary-layer interaction leading to a flow separation.

To solve this 1.4 million cells test case, the Roe [30] scheme is used at orders 1 and 2 by linear reconstruction (Van Albada limiter [39]) with the negative version of Spalart-Allmaras model (Allmaras *et al.* [6]). The use of this version of the Spalart-Allmaras model ensure the regularization of the turbulent equation that is determinant in the fixed point resolution. All the walls (blade, hub and pan) are considered adiabatic.

Results obtained on this test case show the robustness of the chosen architecture. Figure 9 shows the fixed point residual convergence of the direct equations. Thanks to the exact differentiation of $\mathcal{R}$ in conjunction with the pseudo-transient continuation technique [13], the Newton-Raphson quadratic convergence is retrieved after 1000 iterations and then the machine precision is reached after less than 10 iterations. The Mach number field is plotted for first (see Fig. 10a) and second (see Fig. 10b) order spatial schemes. The magnitude of local Mach numbers are in agreement with Ameri [7] and Denton [15]. As expected, shocks are better captured and their wakes are better described with second order. The compression ratio is 2.106 and the nominal rate flow obtained is 20.19 kg/s, value found in the litterature.
Figure 11 shows the GMRES residual convergence of the adjoint system. Also due to the exactness of the differentiate operator, the linear algebra iterative method converges to machine precision. Figures 12a and 12b present the first component of the adjoint vector, at respectively

first and second orders, for the entropy flux as objective function. As expected, the adjoint vector highlights zones, upstream shocks, which start at their feet. As for the direct computation, the second order allows for a finest solution.

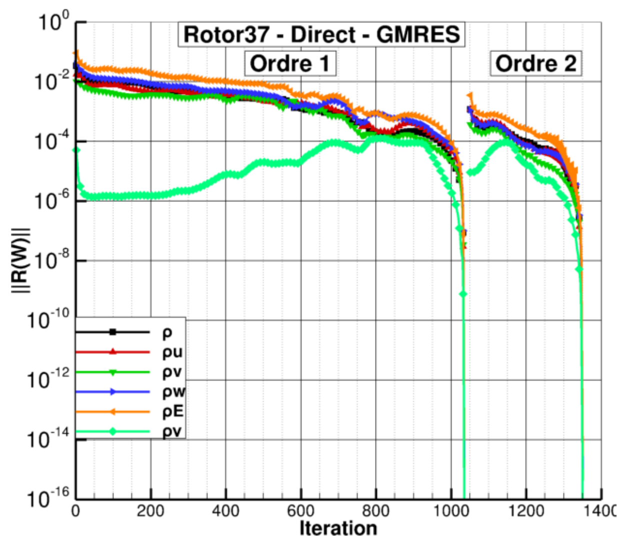
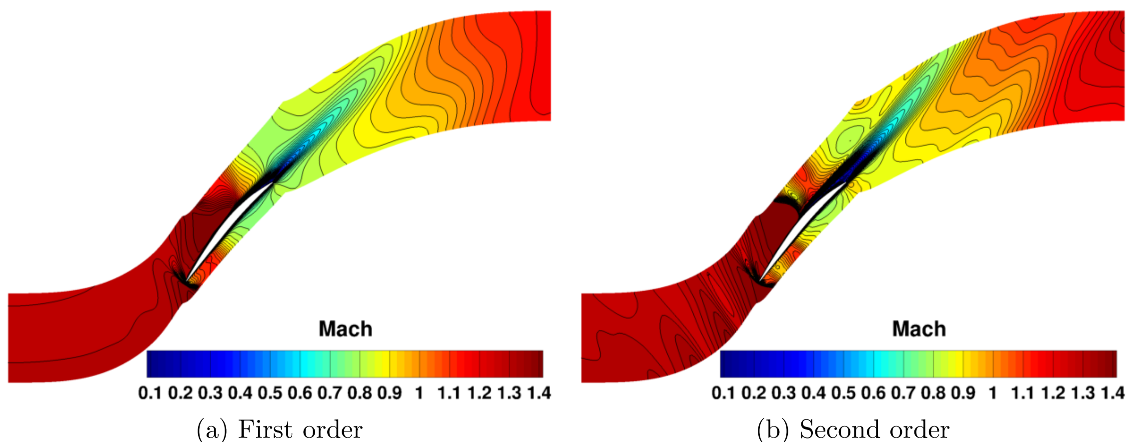


Figure 9: Rotor37: fixed point residual convergence of the direct equations



(a) First order

(b) Second order

Figure 10: Rotor37: Mach fields

8 CONCLUSION

In the present work, we demonstrate the relevance of Source Transformation Algorithm Differentiation (ST-AD) to obtain a maintainable, efficient, industrial solver which covers a wide range of CFD applications in a parallel distributed environnement. We highlight the benefits

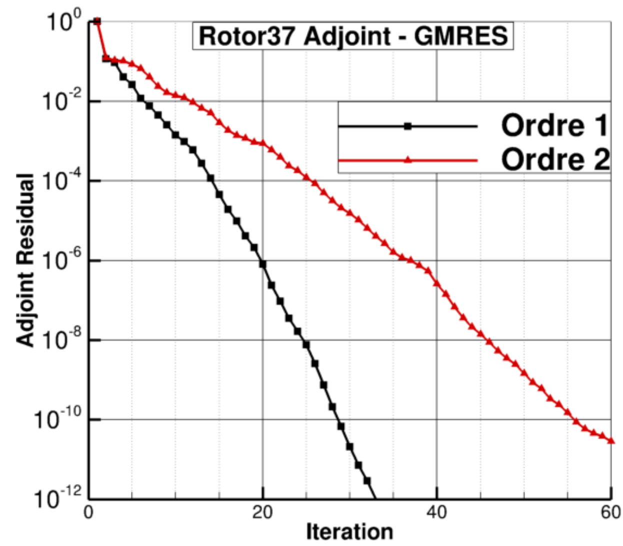
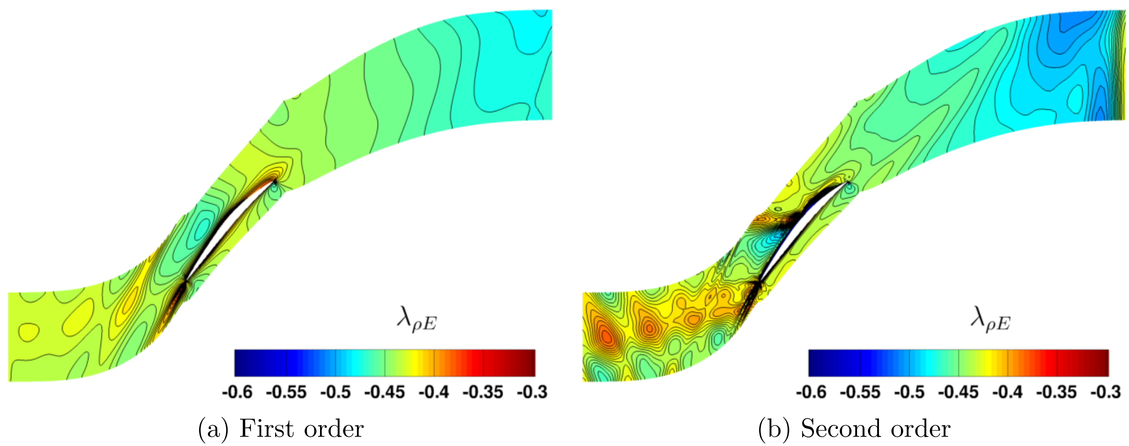


Figure 11: Rotor37: GMRES residual convergence of the adjoint system inversion



(a) First order (b) Second order

Figure 12: Rotor37: First component of adjoint fields

of Source Transformation-AD compared to Operator-Overloading-AD. Especially for adjoint mode, ST-AD is far more efficient, both in terms of CPU and memory costs. We emphasized the improvements made on the Tapenade ST-AD tool, in order to preserve the HPC behavior of each operator in tangent and adjoint modes. Specifically, we discuss how to automatically preserve vectorization and low memory consumption.

The ST-AD tool is able to generate the exact tangent, adjoint and first order Jacobian code for each operator under consideration, preserving the efficiency qualities of the direct operator. To implement the entire solver, operators are organized in chains of operators which have to be reversed for the adjoint mode. Additional research is needed to reverse these chains automatically.

The obtained solver has been validated on several test cases from academic to industrial configurations without any major drawback. Thanks to the exactness of the differentiated operator, the linear algebra iterative method converges to machine precision on all investigated applications.

Acknowledgment

Many thanks to the elsA Team and to all elsA developpers.

The study presented in this article makes use of the elsA CFD software, whose co-owners are Safran and ONERA, and of the TAPENADE software developed at INRIA Sophia-Antipolis.

Following these preliminary results, we continue these activities in collaboration with Safran in the SONICE project funded by DGAC (French Government).

References

REFERENCES

[1] Codipack. https://www.scicomp.uni-kl.de/codi/.

[2] Su2. https://su2code.github.io.

[3] Third international workshop on high-order cfd methods. https://www.grc.nasa.gov/hiocfd, 2015.

[4] Agard-AR-355. CFD Validation for Propulsion System Components. Technical report, May 1998.

[5] T. Albring, N. Gauger, M. Sagebaum, and B. Zhou. AD-based discrete adjoints in SU2. https://su2code.github.io/documents/su2_dev_gauger.pdf.

[6] S. Allmaras, F. Johnson, and P. Spalart. Modifications and clarifications for the implementation of the spalart-allmaras turbulence model. In *Seventh international conference on computational fluid dynamics (ICCFD7)*, pages 1–11, 2012.

[7] A. Ameri. Nasa rotor 37 cfd code validation glenn-ht code. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, page 1060, 2009.

[8] F. Bassi, L. Botti, A. Colombo, A. Ghidoni, and F. Massa. Linearly implicit Rosenbrock-type Runge-Kutta schemes applied to the Discontinuous Galerkin solution of compressible and incompressible unsteady flows. *Computers and Fluids*, 118:305–320, June 2015.

[9] Berger, Pommeau, and Vidal. *Order Within Chaos.* John Wiley, 1984.

[10] M. E. Brachet, D. Meiron, S. Orszag, B. Nickel, R. Morf, and U. Frisch. The Taylor-Green vortex and fully developed turbulence. *Journal of Statistical Physics*, 34(5-6):1049–1063, 1984.

[11] Marc E. Brachet, Daniel I. Meiron, Steven A. Orszag, B. G. Nickel, Rudolf H. Morf, and Uriel Frisch. Small-scale structure of the taylorgreen vortex. *Journal of Fluid Mechanics*, 130:411452, 1983.

[12] L. Cambier, S. Heib, and S. Plot. The ONERA elsA CFD software : input from research and feedback from industry. *Mech. Ind.*, page ., 2013.

[13] A. Crivellini and F. Bassi. An implicit matrix-free discontinuous Galerkin solver for viscous and turbulent aerodynamic simulations. *Computers and Fluids*, 50:81–93, 2011.

[14] C.F. Curtiss and J.O. Hirschfelder. Integration of stiff equation. *Proc. Natl. Acad. Sci. USA*, 43:235, 1952.

[15] JD Denton. Lessons from rotor 37. *Journal of Thermal Science*, 6(1):1–13, 1997.

[16] R. Dwight. *Efficiency improvements of RANS-based analysis and optimization using implicit and adjoint methods on unstructured grid.* PhD thesis, University of Manchester, 2006.

[17] D. Fauconnier, C. Bogey, and E. Dick. On the performance of relaxation filtering for large-eddy simulation. *Journal of Turbulence*, 14(1):22–49, 2013.

[18] Dieter Fauconnier, Chris De Langhe, and Erik Dick. Construction of explicit and implicit dynamic finite difference schemes and application to the large-eddy simulation of the Taylor-Green vortex. *Journal of Computational Physics*, 228:8053–8084, 2009.

[19] M.B. Giles and N.A. Pierce. An introduction to the adjoint approach to design. *Flow Turbulence and Combustion*, 65:393–415, 2000.

[20] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 105 in Other Titles in Applied Mathematics. SIAM, 2nd edition, 2008.

[21] L. Hascoët and V. Pascual. The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, 2013.

[22] Stefan Hickel, Nikolaus A. Adams, and J. Andrzej Domaradzki. An adaptative local deconvolution method for implicit les. *Journal of Computational Physics*, 213:413–436, 2006.

[23] Gaetan KW Kenway, Charles A Mader, Ping He, and Joaquim RRA Martins. Effective adjoint approaches for computational fluid dynamics. *Progress in Aerospace Sciences*, page 100542, 2019.

[24] D.a. Knoll and D.E. Keyes. Jacobian-free NewtonKrylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357–397, January 2004.

[25] J.L. Lions. *Optimal Control of Systems Governed by Partial Differential Equations.* Springer - Verlag, 1971.

[26] P. Luchini and A. Bottaro. An Introduction to Adjoint Problems. *Annual Review of Fluid Mechanics*, 46:., 2014.

[27] I. Mary. Flexible aerodynamic solver technology in an hpc environment. `http://www.maisondelasimulation.fr/seminar/data/201611_poster.pdf`.

[28] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, page ., 1965.

[29] Royce D Moore and Lonnie Reid. Performance of single-stage axial-flow transonic compressor with rotor and stator aspect ratios of 1.19 and 1.26 respectively, and with design pressure ratio of 2.05. Technical report, NASA, 1980.

[30] P. Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics*, 43(2):357–372, 1981.

[31] H.H. Rosenbrock. Some general implicit processes for the numerical solution of differential equations. *The Computer Journal*, 5:329–330, January 1963.

[32] Youcef Saad and Martin H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, July 1986.

[33] V. Schmitt and F. Charpin. Pressure distributions on the ONERA M6-wing at transonic mach numbers, experimental data base for computer program assessment. Technical report, 1979.

[34] AJ Strazisar. Data report and data diskette for nasa transonic compressor rotor 37. *NASA Lewis Research Center, Cleveland, OH*, page ., 1994.

[35] Kenneth Lee Suder. Blockage development in a transonic, axial compressor rotor. *Journal of Turbomachinery*, 120(3):465–476, 1998.

[36] G.I. Taylor and A.E. Green. Mechanism of the Production of Small Eddies from Large Ones. *Proc. R. Soc. Lond. A*, 158:499–521, 1937.

[37] L. Tuckerman and D. Barkley. Numerical methods for bifurcation problems and large-scale dynamical systems – Bifurcation analysis for timesteppers. *Springer*, pages 453–466, 2000.

[38] J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland, and U. Naumann. Toward Adjoinable MPI. In *Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering, PDSEC'09*, page ., 2009.

[39] GD Van Albada, Bram Van Leer, and WWjun Roberts. A comparative study of computational methods in cosmic gas dynamics. In *Upwind and High-Resolution Schemes*, pages 95–103. Springer, 1997.