Automatic Differentiation applied to Edge in NFFP5 project MADEF

OLIVIER AMOIGNON

FOI-R--3689--SE
ISSN 1650-1942 May 2013

Olivier Amoignon

Automatic Differentiation applied to Edge in NFFP5 project MADEF

Titel	Automatisk Differentiering tillämpad i Edge inom NFFP5 projektet MADEF
Title	Automatic Differentiation applied to Edge in NFFP5 project MADEF
Rapportnr/Report no	FOI-R--3689--SE
Månad/Month	05
Utgivningsår/Year	2013
Antal sidor/Pages	29 p
ISSN	1650-1942
Kund/Customer	VINNOVA
Forskningsområde	3. Flygteknik och luftstridssimulering
FoT-område	Flygteknik
Projektnr/Project no	E28174
Godkänd av/Approved by	Peter E Eliasson
Ansvarig avdelning	320

Sammanfattning

Inom MADEF projektet tillämpas automatisk differentiering (AD) för att komplettera utvecklingen av den adjungerade strömningsekvationslösaren i Edge, vilket används inom aerodynamisk design. Vi använde programmet Tapenade från INRIA för att utveckla ”dforce”, ett program som räknar derivator av funktionen F vilka beror av strömningslösningen, till exempel aerodynamiska krafter och moment, eller DC60 kriteriet för interna strömningar. Derivator av F, till exempel med avseenden på primitiva eller konservativa variabler vid noderna, är partiella derivator vilka används vid olika steg när man räknar derivator av F med avseenden på formen. I vanliga designtillämpningar, där gradient-baserad optimering används, skulle F kunna vara antingen målfunktionen som ska minimeras eller ett av bivillkoren. Efter utvecklingen av dforce har adjunktlösaren i Edge anpassats för att kunna använda högerled beräknade med hjälp av AD.

Nyckelord: funktionsderivator, känsligheter, adjunkt metoden, automatisk differentiering, bakåt mod.

Summary

In the project MADEF we apply automatic differentiation (AD) as a complement to the adjoint flow solver already present in the CFD code Edge, in the framework of aerodynamic aircraft design. We used the code Tapenade from INRIA for the development of a program called “dforce” that calculates derivatives of a function F depending on the flow solution given by the CFD solver such as aerodynamic forces, moment or criteria like the DC60 for internal flows. The derivatives of F, with respect to the nodal values of the primitive or conservative variables for instance, are partial derivatives used at different stages when calculating the shape derivative of F. Our focus is thus on applications in design carried out by gradient-based optimization where F is either the cost function being minimized or a constraint. Following the development of the program dforce, the adjoint solver in Edge has been adapted in order to enable using right-hand sides calculated by AD.

Keywords: function derivatives, sensitivities, adjoint method, automatic differentiation, reverse mode.

1 Introduction

In introduction we describe the relation between the field of aerodynamic design optimization and techniques of automatic differentiation.

Aerodynamic shape optimization for aircraft design is a challenging for all numerical methods involved: parameterization of shape deformations, mesh deformation, flow simulations and gradients computation with respect to design parameters. From the perspective of optimization, aerodynamic design has thus the particularity to involve the flow equations (CFD) as constraints, and due to the challenging physics this commonly involves from a couple of thousands degrees of freedom (dof) , for simple flow models in two dimensions, to millions of dof, in three dimension. In this class of problems the *adjoint approach* emerged as an efficient method for calculating derivatives of the cost function, also called objective, and constraints, with respect to the design parameters including the constraints of the flow equations in the so called **reduced gradient**. Without the adjoint of the flow equations design by optimization involving fluid mechanics is limited to using a few parameters of design or to use simplified flow models, eventually leading to designs that will poorly perform in reality. Optimization based on derivatives (gradients) and adjoint flow solvers appears to be the best approach for single- and multi-disciplinary industrial design involving many parameters and computer intensive simulations.

Automatic differentiation (AD) appears to have a natural connection with the adjoint approach. AD designates various technologies aiming at calculating derivatives of functions computed by algorithms. Briefly described, AD tools implement the chain rule of differential calculus to source codes; it is not to be confused with symbolic differentiation or divided differences. Introductions on the subject can be found in references [1] and [6], a large source of references can be found on the AD community site [7]. AD software is most often dedicated to one or two programming languages: C/C++ (ADOL-C, ADIC, AMPL, OpenAD, Tapenade …), Fortran (ADIFOR, Tapenade, OpenAD …), Matlab (ADimat, AD, Adiff …). From the beginning, application of AD has covered all areas using software for computations: in product design, simulations or for data analysis. Classic applications are for instance error estimation on complicated algorithms such as solvers of differential equations [4] or in electrical engineering [3] where the adjoint analysis, the analogue of reverse mode in AD, has been early recognized for its ability to obtain accurate and cheap derivatives. In AD, the superiority of reverse programming was early recognized [2] but it did not gain a large audience of engineers and scientists until dedicated software named above started to be developed in the early eighties.

Much of the research in AD concerns the performance of the produced code in reverse mode, the major bottle-neck, and the readability of the produced code when codes need be further maintained. Developers of AD, among them the INRIA team developing TAPENADE, indicate that important developments are needed and require research for problems related to the adjoint (reverse) mode: MPI, dynamic memory management and object-oriented languages [5].

In general, application of AD requires important transformation of the source program before and after generating the differentiated code. As Griewank summarizes it, the problem of general purpose differentiation is of a computer science nature, while the mathematics is quite straight forward [1]. The general advice in the literature being that the use of AD should be limited to specific situations and needs; for example when finite differences derivatives are inaccurate, evaluating the function N times (number of parameters) is too expensive, or when the adjoint equation is impossible to

obtain with reasonable efforts. Another field of research is the application of AD on codes with several languages.

AD, unlike the adjoint method, intends to differentiate what is implemented; this is in contrast with the developer of an adjoint solver who will program what he thinks is the adjoint of the flow solver. The latter assumes that the code is continuously differentiable, assumption that must carefully be observed because programmers, by convenience or in order to circumvent difficulties; most CFD codes contain "if" statements on variables being computed in some of their numerical algorithms that need be differentiated (branching) or non-differentiable expressions like absolute value.

While differentiation of a CFD solver can require a tremendous amount of work, and may be totally impossible to do on already existing codes, numerical aerodynamic shape optimization requires several differentiations that need not to be performed by the same method. The picture below (Figure 1) illustrates a typical "loop" in the context of computational aerodynamic shape optimization using CFD. All algorithms, from the parameterization of the shape to the flow solver are differentiated in some way to obtain accurate gradients of the drag, lift and other figures of merit. In this particular example (a code called AESOP [10]) we use, apart from hand-differentiated adjoint flow solver and pre-processor: hand-differentiated Radial Basis Function interpolations (RBF for parameterization or mesh deformation) and elliptic solvers (such as discrete Laplace for mesh deformation), complex-trick for industrial-like parameterizations (twist distribution for ex.), symbolic differentiation for the efficient assembly of the gradient of non trivial cost functions (such as polynomial or rational expressions, power or logarithm functions, and combination of those). However, typical optimizations have a number of parameters ranging from a handful up to hundreds for pure aerodynamic applications. As we are progressing towards aeroelastic optimization this number will increase, eventually ending with sizing the structure of a wing, which can involve finding the values of thousands of parameters. More advanced projects could involve topology optimization of the structure coupled to aerodynamic design, further increasing the number of design parameters.

Scalability is thus an issue we consider in the development of experimental code as AESOP. The adjoint approach involves thus the solution of the adjoint flow equations in Edge, in order to reduce the computational cost of the partial derivatives of the drag or other performances related to the flow solution with respect to the design parameters.

The AD tool Tapenade is developed by INRIA [1] and is used in this work in order to differentiate some routines of Edge, our objectives being:

- Extend the use of the adjoint solver to the calculation of figures of performances that have not yet been "hand-differentiated" such as DC60.
- Provide an alternative method for the derivation of components of the adjoint solver such as boundary conditions.
- Identify possible applications of the technology in our activities on optimal design, for the development of adjoint turbulence models or as a mean of verification of codes under the development phase.

It is therefore not intended to replace the adjoint approach by AD because of the low performance of the code produced, but more to investigate potential benefits that it could bring to our activities.

The next section presents applications of automatic differentiation on some examples. We introduce some basic ideas about the adjoint method because it is used in the

derivation presented here of the reverse mode. The third section presents the program "dforce".

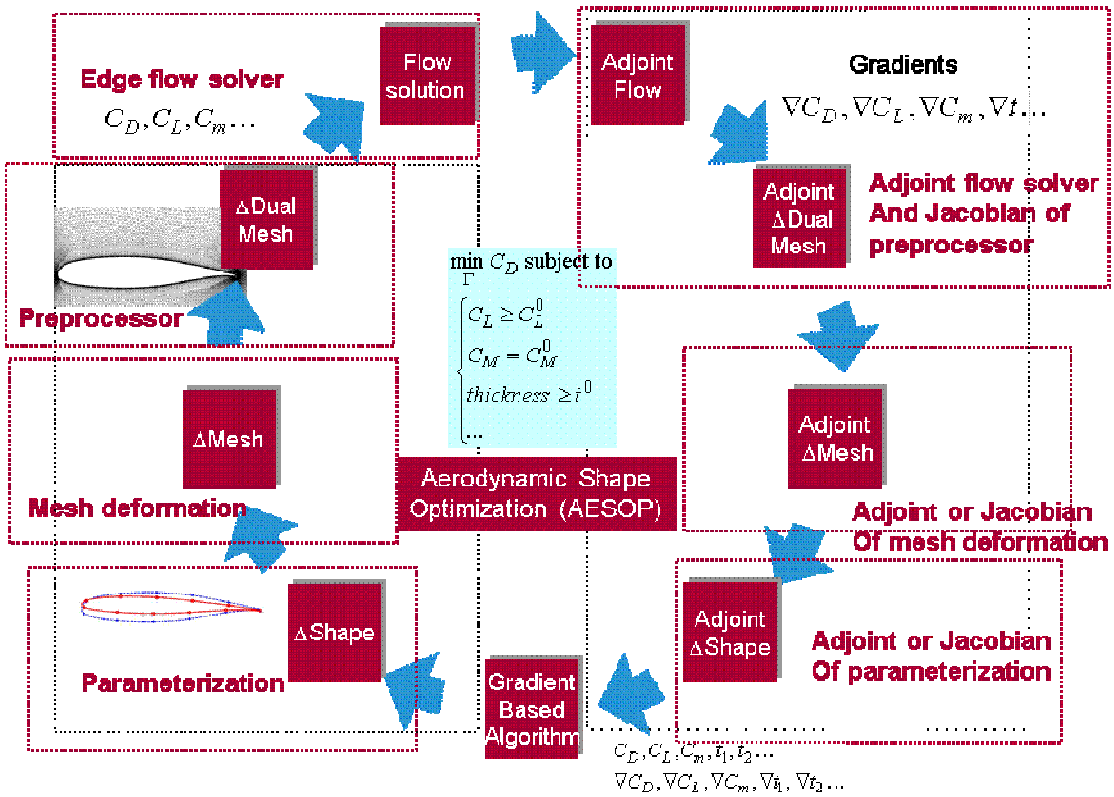

Figure 1 - Solution of aerodynamic optimization problems usually involves an iterative procedure as here in the code AESOP which integrates the flow solver EDGE, its adjoint and other algorithms.

2 Methods

2.1 Sensitivities versus Adjoint

This section exposes how in principle the adjoint method calculates the gradient of a function f used in an optimization:

f is a function of $\mathbf{w}$, solution of equation

$$\mathbf{A}\mathbf{w} = \mathbf{N}\mathbf{a} \quad \mathbf{a} \in \Re^n$$

Assuming that f is linear in $\mathbf{w}$:

$$f(\mathbf{w}) = \mathbf{g}^{\mathrm{T}}\mathbf{w}$$

Computing gradients with sensitivities can be described as

$$\forall \delta\mathbf{a}, \quad \delta f(\mathbf{w}) = \mathbf{g}^{\mathrm{T}}\delta\mathbf{w} \quad \text{where} \quad \mathbf{A}\delta\mathbf{w} = \mathbf{N}\delta\mathbf{a}$$

and by definition of the gradient :

$$\delta f(\mathbf{w}) = \nabla f^{T}\delta\mathbf{a}$$

which suggests the algorithm :

for k = 1 : n

$$\mathbf{A}\delta\mathbf{w}_k = \mathbf{N}\mathbf{e}_k \quad , \mathbf{e}_k = [0...0\underset{k}{1}0...0]$$

$$(\nabla f)_k = \mathbf{g}^{\mathrm{T}}\delta\mathbf{w}_k$$

end

The cost of a gradient computed by sensitivities is thus proportional to the number of parameters (n). The adjoint approach is now introduced via this example:

$$\forall \delta\mathbf{a}, \quad \delta f(\mathbf{w}) = \mathbf{g}^{\mathrm{T}}\delta\mathbf{w} \quad \text{where} \quad \mathbf{A}\delta\mathbf{w} = \mathbf{N}\delta\mathbf{a}$$

$$\Rightarrow \delta f(\mathbf{w}) = \mathbf{g}^{\mathrm{T}}\left(\mathbf{A}^{-1}\mathbf{N}\delta\mathbf{a}\right)$$

$$\Leftrightarrow \delta f(\mathbf{w}) = \left(\mathbf{A}^{-T}\mathbf{g}\right)^{T}\mathbf{N}\delta\mathbf{a}$$

therefore solving $\mathbf{A}^{\mathrm{T}}\mathbf{w}^{*} = \mathbf{g}$

gives $\nabla f = \mathbf{N}^{\mathrm{T}}\mathbf{w}^{*}$

It consists in identifying an equation (the adjoint of a real matrix is the transposed matrix) which solution enables giving an expression of the gradient at once instead of requiring n evaluations of the state equation.

2.2 Adjoint approach for coupled equations

The purpose is to show that even dealing with a complex system of equations the adjoint approach reveals the structure of the equations that must be solved in order to efficiently compute gradients of a performance J. The equations in the example below are representative of a shape optimization problem of an aeroelastic system solved with Edge where the structure model is represented by modes.

In order to optimize the shape a parameterization of the geometry is defined as follows: the vector of design parameters **a** determines displacements of the shape, called here the boundary, and those displacements at the boundary are propagated to the entire volume grid used for the flow computation (CFD):

$$\begin{cases} \text{Parameterization of the boundary:} \\ \mathbf{S}\mathbf{Y}_0^{\Gamma} = \mathbf{C}\mathbf{a} \quad \text{(fluid boundary displacement)} \\ \mathbf{A}\mathbf{Y}_0 = \mathbf{B}\mathbf{Y}_0^{\Gamma} \quad \text{(mesh deformation)} \end{cases}$$

The equations are thus solved for the statically coupled fluid-structure (modal) system, which in Edge can be described like this

$$\begin{cases} \mathbf{R}_h\left(\mathbf{U}_h, \mathbf{n}_h\left(\mathbf{X}_0 + \mathbf{Y}_0 + \mathbf{Y}_f\right), \mathbf{X}_0 + \mathbf{Y}_0 + \mathbf{Y}_f\right) = \mathbf{0} \\ (\mathbf{Y}_0 : \text{initial displacement}) \\ f = \mathbf{H}^T \mathbf{Q}\mathbf{U}_h \quad \text{(aerodynamic load)} \\ \Omega\eta = \mathbf{Z}^{\mathbf{T}} f \quad (\Omega : \text{diagonal}, \eta : \text{modal coordinates}) \\ \mathbf{Y}_{fs}^{\Gamma} = \mathbf{H}\mathbf{Z}\eta \quad \text{(fluid boundary displacement)} \\ \mathbf{A}\mathbf{Y}_f = \mathbf{B}\mathbf{Y}_{fs}^{\Gamma} \quad \text{(mesh deformation)} \end{cases}$$

Finally, we evaluate the performance J (drag, lift …) of the design:

$$J_h\left(\mathbf{U}_h, \mathbf{X}_0 + \mathbf{Y}_0 + \mathbf{Y}_f\right)$$

In order to optimize the performance J, the gradient of J with respect to the design parameters **a** will be calculated. We develop here symbolically the adjoint approach that can be used, showing which equations are involved.

We start by defining a global vector **V** for the various states and a global residual vector **T**:

$$V = \begin{bmatrix} \mathbf{Y}_0^{\Gamma} \\ \mathbf{Y}_0 \\ \mathbf{U}_h \\ \mathbf{Y}_f \\ \eta \end{bmatrix} \text{ and } \mathbf{T}(\mathbf{V},\mathbf{a}) = \begin{bmatrix} \mathbf{S}\mathbf{Y}_0^{\Gamma} - \mathbf{C}\mathbf{a} \\ \mathbf{A}\mathbf{Y}_0 - \mathbf{B}\mathbf{Y}_0^{\Gamma} \\ \mathbf{R}_h\left(\mathbf{U}_h, \mathbf{X}_0 + \mathbf{Y}_0 + \mathbf{Y}_f\right) \\ \mathbf{A}\mathbf{Y}_f - \mathbf{B}\mathbf{H}\mathbf{Z}\eta \\ \Omega\eta - \mathbf{Z}^T\mathbf{H}^T\mathbf{Q}\mathbf{U}_h \end{bmatrix}$$

We can thus define as usual an adjoint system of equations:

$$\left(\frac{\partial \mathbf{T}}{\partial \mathbf{V}}\right)^T \tilde{\mathbf{V}} = -\left(\frac{\partial J_h}{\partial \mathbf{V}}\right)^T$$

$$\text{with} \quad \left(\frac{\partial J_h}{\partial \mathbf{V}}\right) = \left[\mathbf{0}, \left(\frac{\partial J_h}{\partial \mathbf{Y}_0}\right), \left(\frac{\partial J_h}{\partial \mathbf{U}_h}\right), \left(\frac{\partial J_h}{\partial \mathbf{Y}_f}\right), \mathbf{0}\right]$$

$$\text{and} \quad \frac{\partial \mathbf{T}}{\partial \mathbf{V}} = \begin{bmatrix} \mathbf{S} & 0 & 0 & 0 & 0 \\ -\mathbf{B} & \mathbf{A} & \mathbf{0} & \mathbf{0} & 0 \\ \mathbf{0} & \frac{\partial \mathbf{R}_h}{\partial \mathbf{Y}_0} & \frac{\partial \mathbf{R}_h}{\partial \mathbf{U}_h} & \frac{\partial \mathbf{R}_h}{\partial \mathbf{Y}_f} & 0 \\ \mathbf{0} & 0 & 0 & \mathbf{A} & -\mathbf{BHZ} \\ \mathbf{0} & 0 & -\mathbf{Z}^T\mathbf{H}^T\mathbf{Q} & 0 & \mathbf{\Omega} \end{bmatrix}$$

Its solution allows calculating exactly the gradient (if all derivatives are computed exactly and all equations are solved down to machine precision):

$$\nabla J_h = \frac{dJ_h}{d\mathbf{a}} = \left(\frac{\partial \mathbf{T}}{\partial \mathbf{a}}\right)^T \tilde{\mathbf{V}} \quad \text{where} \quad \left(\frac{\partial \mathbf{T}}{\partial \mathbf{a}}\right)^T = \left[-\mathbf{C}^T, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}\right]$$

The large adjoint system above can be broken down into several equations

$$(3) \quad \mathbf{S}^T \tilde{\mathbf{Y}}_0^\Gamma = \mathbf{B}\tilde{\mathbf{Y}}_0$$

$$(2) \quad \mathbf{A}^T \tilde{\mathbf{Y}}_0 = -\left(\frac{\partial \mathbf{R}_h}{\partial \mathbf{Y}}\right)_{\mathbf{X_0}}^T \tilde{\mathbf{U}}_h - \left(\frac{\partial J_h}{\partial \mathbf{Y}}\right)_{\mathbf{X}_0}^T$$

$$(1) \quad \begin{cases} \left(\frac{\partial \mathbf{R}_h}{\partial \mathbf{U}_h}\right)_{\mathbf{X}_0}^T \tilde{\mathbf{U}}_h = \mathbf{Q}^T\mathbf{HZ}\tilde{\eta} - \left(\frac{\partial J_h}{\partial \mathbf{U}_h}\right)_{\mathbf{X}_0}^T & \text{(usual adjoint flow + 1 source term)} \\ \mathbf{A}^T \tilde{\mathbf{Y}}_f = -\left(\frac{\partial \mathbf{R}_h}{\partial \mathbf{Y}}\right)_{\mathbf{X_0}}^T \tilde{\mathbf{U}}_h - \left(\frac{\partial J_h}{\partial \mathbf{Y}}\right)_{\mathbf{X}_0}^T & \text{(usual adjoint mesh deformation)} \\ \Omega^T \tilde{\eta} = \mathbf{Z}^T\mathbf{H}^T\mathbf{B}^T \tilde{\mathbf{Y}}_f & \text{(adjoint modal structure equation)} \end{cases}$$

$$\Rightarrow \tilde{\mathbf{Y}}_0 = \tilde{\mathbf{Y}}_f \ \text{(gradients w.r.t. mesh coordinates need not be re - computed by (2))}$$

Solving (1) and (3) enables to solve the "usual" gradient expression:

$$\nabla J_h = \frac{dJ_h}{d\mathbf{a}} = -\mathbf{C}^T \tilde{\mathbf{Y}}_\mathbf{0}^\Gamma$$

Note that equation (2) can be skipped only in the special case where the same mesh deformation is used in order to deform the baseline grid due to changes in design

parameters and to deform the grid when solving the coupled fluid-structure system of equations.

2.3 Automatic differentiation

AD blends rule-based differentiation, for example for intrinsic functions like sin(u), u**2, and derivatives accumulation following the chain rule of calculus.

The forward mode starts with derivatives of the input variables and propagates them as for the calculation of the function. The backward mode starts with the derivatives of the output propagating them backward to the input variables.

In order to "visualize" both approaches we propose here to symbolically apply AD on a routine only implementing a real vector valued function calculated by explicit expressions, without branching.

2.3.1 Forward mode

It can be described as follows: for a vector **x** of input parameters, having n components, the vector valued function **f** is defined by a sequence of M assignments, like M lines of code, each possibly using all expressions already computed by the routine:

$$
\begin{aligned}
&\mathbf{y}_1 = \mathbf{f}_1(\mathbf{x}) \quad , \mathbf{x} \in \mathbf{R}^n, \mathbf{y}_j \in \mathbf{R}^m \\
&\mathbf{y}_2 = \mathbf{f}_2(\mathbf{x}, \mathbf{y}_1) \\
&\mathbf{y}_3 = \mathbf{f}_3(\mathbf{x}, \mathbf{y}_1, \mathbf{y}_2) \\
&\qquad \vdots \\
&\mathbf{y}_M = \mathbf{f}_M(\mathbf{x}, \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{M-1}) \\
&\mathbf{f} = \mathbf{y}_M
\end{aligned}
$$

We introduce the following notations:

For any **U** and **V** in $\mathbf{R}^{(\mathrm{M}\times\mathrm{m})}$,

$$
\text{where } \mathbf{U} = \begin{bmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_{M-1} \\ \mathbf{u}_M \end{bmatrix} \text{ and } \mathbf{u}_j = \begin{bmatrix} u_{j,1} \\ \vdots \\ u_{j,m} \end{bmatrix}
$$

$\mathbf{w} = \mathbf{U} \cdot \mathbf{V}$ is an element in $\mathbf{R}^m$ with components

$$
w_i = \sum_{j=1}^{M} u_{j,i} v_{j,i}
$$

The vector valued function is expressed using the above introduced inner product

$$
\begin{aligned}
&\mathbf{f} = \mathbf{C} \cdot \mathbf{Y} \\
&\text{with } \mathbf{C} = \begin{bmatrix} \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{1} \end{bmatrix}, \mathbf{1} \in \mathbf{R}^m \text{ and } \mathbf{Y} = \begin{bmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_{M-1} \\ \mathbf{y}_M \end{bmatrix}
\end{aligned}
$$

Via those notations the vector valued function can be more general, involving for instance the combination of intermediate results produced in the routine.

A straight-forward differentiation of the function assignments gives its first variation

$$\delta\mathbf{y}_1 = \left(\frac{\partial \mathbf{f}_1(\mathbf{x})}{\partial \mathbf{x}}\right)\delta\mathbf{x}$$

$$\delta\mathbf{y}_2 = \left(\frac{\partial \mathbf{f}_2(\mathbf{x},\mathbf{y}_1)}{\partial \mathbf{x}}\right)\delta\mathbf{x} + \left(\frac{\partial \mathbf{f}_2(\mathbf{x},\mathbf{y}_1)}{\partial \mathbf{y}_1}\right)\delta\mathbf{y}_1$$

$$\delta\mathbf{y}_3 = \left(\frac{\partial \mathbf{f}_3(\mathbf{x},\mathbf{y}_1,\mathbf{y}_2)}{\partial \mathbf{x}}\right)\delta\mathbf{x} + \sum_{j=1}^{2}\left(\frac{\partial \mathbf{f}_3(\mathbf{x},\mathbf{y}_1,\mathbf{y}_2)}{\partial \mathbf{y}_j}\right)\delta\mathbf{y}_j$$

$$\vdots$$

$$\delta\mathbf{y}_M = \left(\frac{\partial \mathbf{f}_M(\mathbf{x},\mathbf{y}_1,\mathbf{y}_2,\cdots,\mathbf{y}_{M-1})}{\partial \mathbf{x}}\right)\delta\mathbf{x} + \sum_{j=1}^{M-1}\left(\frac{\partial \mathbf{f}_M(\mathbf{x},\mathbf{y}_1,\mathbf{y}_2,\cdots,\mathbf{y}_{M-1})}{\partial \mathbf{y}_j}\right)\delta\mathbf{y}_j$$

where $\left(\partial\mathbf{f}_j/\partial\mathbf{x}\right)\in\mathbf{R}^{m\times n}$ and $\left(\partial\mathbf{f}_j/\partial\mathbf{y}_k\right)\in\mathbf{R}^{m\times m}$

$$\delta\mathbf{Y} = \mathbf{A}\delta\mathbf{x} + \mathbf{B}\delta\mathbf{Y} \quad \text{with } \mathbf{A}\in\mathbf{R}^{(M\times m)\times n} \text{ and } \mathbf{B}\in\mathbf{R}^{(M\times m)^2}$$

$$\delta\mathbf{f} = \mathbf{C}\cdot\delta\mathbf{Y} \text{ or}$$

$$\delta\mathbf{f} = \mathbf{C}\cdot\left(\left(\mathbf{I}-\mathbf{B}\right)^{-1}\mathbf{A}\delta\mathbf{x}\right)$$

All entries in matrices A and B in the expressions above are thus obtained using the rules of differentiation that all AD codes based on program transformation are implementing (see the routine F3_D below differentiated in forward mode with Tapenade).

$$\mathbf{A}_j = \left(\frac{\partial \mathbf{f}_j(\mathbf{x},\mathbf{y}_1,\mathbf{y}_2,\cdots,\mathbf{y}_{j-1})}{\partial \mathbf{x}}\right)\in\mathbf{R}^{m\times n}, \quad 1\le j\le M$$

$$\mathbf{A} = \begin{pmatrix}\mathbf{A}_1\\ \vdots \\ \mathbf{A}_M\end{pmatrix}$$

$$\mathbf{B}_{jk} = \left(\frac{\partial \mathbf{f}_j(\mathbf{x},\mathbf{y}_1,\mathbf{y}_2,\cdots,\mathbf{y}_{j-1})}{\partial \mathbf{y}_k}\right)\in\mathbf{R}^{m\times m}, \quad 2\le j\le M, \quad 1\le k\le j-1$$

$$\mathbf{B} = \begin{pmatrix}\mathbf{0} & \mathbf{0} & \cdots & \mathbf{0}\\ \mathbf{B}_{21} & \mathbf{0} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathbf{0}\\ \mathbf{B}_{M1} & \cdots & \mathbf{B}_{M(M-1)} & \mathbf{0}\end{pmatrix}$$

From those sensitivities we can “calculate” the forward mode derivatives:

$$\frac{\partial \mathbf{f}}{\partial x_i} = \mathbf{C} \cdot \left(\left(\mathbf{I} - \mathbf{B} \right)^{-1} \mathbf{A} \mathbf{x}_i^* \right) \quad 1 \le i \le n$$
$$\mathbf{x}_i^* = \left[0 \cdots 0 \underset{i}{1} 0 \cdots 0 \right]^T$$

The Jacobian[1] of the function is obtained by "seeding" its forward differentiated version for each of the component of its entry data with a vector **x*** having only one non-zero component at the i[th] position. The cost appears thus to be n, as expected.

2.3.2 Backward ("adjoint") mode

Now, transforming the final expression in the sensitivities above yields:

$$\delta \mathbf{f} = \mathbf{C} \cdot \left(\left(\mathbf{I} - \mathbf{B} \right)^{-1} \mathbf{A} \delta \mathbf{x} \right) \Leftrightarrow \delta \mathbf{f} = \left(\left(\mathbf{I} - \mathbf{B} \right)^{-T} \mathbf{C} \right) \cdot \mathbf{A} \delta \mathbf{x}$$

By analogy to the adjoint state equation presented in section 2.1 (Sensitivities versus Adjoint), we introduce here an "adjoint" vector $\tilde{\mathbf{Y}}$, the components being the results of code lines (assignments) that we will express further down.

$$\delta \mathbf{f} = \left(\left(\mathbf{I} - \mathbf{B} \right)^{-T} \mathbf{C} \right) \cdot \left(\mathbf{A} \delta \mathbf{x} \right)$$
$$\left(\mathbf{I} - \mathbf{B} \right)^T \tilde{\mathbf{Y}} = \mathbf{C}$$
$$\Rightarrow \forall \delta \mathbf{x}, \quad \delta \mathbf{f} = \tilde{\mathbf{Y}} \cdot \left(\mathbf{A} \delta \mathbf{x} \right)$$
$$\left(\delta \mathbf{f} \right)_i = \sum_{j=1}^{M} \tilde{y}_{(j-1)m+i} \sum_{k=1}^{M} \mathbf{A}_{(j-1)m+i,k} \delta \mathbf{x}_k$$
$$\Leftrightarrow \left(\delta \mathbf{f} \right)_i = \sum_{k=1}^{M} \left(\sum_{j=1}^{M} \tilde{y}_{(j-1)m+i} \mathbf{A}_{(j-1)m+i,k} \right) \delta \mathbf{x}_k$$

and defining the Jacobian matrix such that

$\delta \mathbf{f} = \mathbf{J} \delta \mathbf{x}$ for all $\delta \mathbf{x}$, then

$$\mathbf{J}_{ik} = \sum_{j=1}^{M} \tilde{y}_{(j-1)m+i} \mathbf{A}_{(j-1)m+i,k}$$ which is in some sense

the transposed of A times the vector of "adjoint" assignments :

$$\mathbf{J} = \mathbf{A}^T \tilde{\mathbf{Y}}$$

The definition of the transposed matrix of A above is related to the inner product defined further up. It is easier to figure out when considering the simpler case where m=1.

[1] Each column of the Jacobian (n components) is the gradient of one component of the vectorial function **f**. If m=1 the Jacobian is called the gradient of *f*.

The structure of the backward "mode" appears examining the structure of **B**: lower triangular with **0** (block) diagonal it reflects the sequential execution of assignments in function **f.**

$$\left(\mathbf{I}-\mathbf{B}\right)^T \widetilde{\mathbf{Y}} = \mathbf{C} \;\Leftrightarrow\; \widetilde{\mathbf{Y}} = \mathbf{C} + \mathbf{B}^T \widetilde{\mathbf{Y}}$$

which is equivalent to the M lines of code :

$$\widetilde{\mathbf{y}}_M = \mathbf{1} \in \mathbf{R}^m$$

$$\widetilde{\mathbf{y}}_{M-1} = \mathbf{B}_{M,M-1}\widetilde{\mathbf{y}}_M$$

$$\widetilde{\mathbf{y}}_{M-2} = \mathbf{B}_{M,M-2}\widetilde{\mathbf{y}}_M + \mathbf{B}_{M-1,M-2}\widetilde{\mathbf{y}}_{M-1}$$

$$\vdots$$

$$\widetilde{\mathbf{y}}_1 = \sum_{j=2}^{M} \mathbf{B}_{j,1}\widetilde{\mathbf{y}}_j$$

and finally

$$\mathbf{J} = \mathbf{A}^T \widetilde{\mathbf{Y}}$$

The additional lines of code $\tilde{\mathbf{Y}}$ are performed in the reverse order in comparison to the components of the intermediate variables **Y** when the function **f** is evaluated. This also shows that without further analysis of the M assignments, the storage and computational costs become real issues when m and M are large:

- "store-all" approach: the non-zero entries of the Jacobian matrices A and B need be computed only once but storage represents up to $Mmn+M(M-1)m^2/2$!
- "recompute all" approach: the non-zero entries of A and B can be computed. At the beginning of the reversed routine by computing first the M assignments of the original routine.

The strategy used in Tapenade blends both approaches using checkpoints.

For comparison, hand differentiation of codes is carried out at the algorithm level, not based on the implementation. It is thus possible to obtain an implementation that can in principle be as efficient as one evaluation of the linearized (sensitivities) function without little additional storage.

2.3.3 Another presentation of the reverse mode

From a presentation given by Pironneau and Dicésaré (UPMC) we get another understanding of AD in backward mode than the pure algebraic (adjoint) approach proposed above. It starts, as it was done above, identifying the assignments as intermediate functions:

$$J(u)$$

with

$$x = l_1(u) \equiv 2u(2u+1)$$

$$y = l_2(x,u) \equiv x + \sin(u)$$

$$l_3(x,y) \equiv x \times y$$

$$J = l_3(x,y)$$

A Lagrangian is then formed where intermediate assignments are considered as equality constraints:

$$L(u,x,y) = J(u) - l_3(x,y) + p_1(x - l_1(u)) + p_2(y - l_2(x,u))$$

Stationarity with respect to the Lagrange multiplier (p_1, p_2) would give back the program computing $J(u)$, whereas stationarity with respect to the intermediate variables, x and y, give expressions for the multipliers (the adjoint variables in this approach), but they must be computed in the reverse order:

$$\begin{aligned}
&1)\ \partial L/\partial y = 0 \Leftrightarrow p_2 - \partial l_3/\partial y = 0\\
&\Rightarrow p_2 = x = 2u(2u+1)\\
&2)\ \partial L/\partial x = 0 \Leftrightarrow p_1 - p_2\partial l_2/\partial x - \partial l_3/\partial x = 0\\
&\Rightarrow p_1 = x + y = 4u(2u+1) + \sin(u)
\end{aligned}$$

Finally, stationarity with respect to the input data (u) gives expressions that, together with the calculated multipliers, enable to compute the derivate of J:

$$\begin{aligned}
&3)\ \partial L/\partial u = 0 \Leftrightarrow dJ/du - p_1\partial l_1/\partial u - p_2\partial l_2/\partial u = 0\\
&\Rightarrow J'(u) = (4u(2u+1) + \sin(u))(4u+2) + 2u(2u+1)\cos(u)\\
&\text{or } J'(u) = (x+y)(4u+2) + x\cos(u)
\end{aligned}$$

We will use the same example (function) below in order to show how code differentiation of a scalar function is performed using TAPENADE.

2.3.4 Cost of gradient calculation in AD

In Forward mode the gradient is obtained at the cost of n (independent variables) function evaluations by executing the following loop:

```
 ! calculates gradient of f:
        ! using the differentiated function f2 in forward (tangent) mode:
        !
        do I=1,N
          xn=0.
          xn(I)=1;
          gf(I) = f3_d(x,xn,f)
        end do
```

In contrast, in backward mode (adjoint) the gradient is obtained at the cost of one function call:

```
    !
        ! calculates gradient of f:
        ! using the differentiated function f2 in reverse (adjoint) mode:
        !
        gfb=0.
        f3b  =1.
        call f3_b(x,gf,f2b)
```

2.4 Examples

2.4.1 Scalar case (m=1) with scalar variable (n=1)

We use the example from section 2.3.3, showing the Fortran and how the approach exposed in section 2.3.1 (Forward mode) is applied:

$\mathbf{x}$ has only one component u (n = 1)

and M = 3

$$y_1 = f_1(\mathbf{x}) = 2u(u+1)$$
$$y_2 = f_2(\mathbf{x}, y_1) = y_1 + \sin(u)$$
$$y_3 = f_3(\mathbf{x}, y_1, y_2) = y_1 y_2$$
$$f = \mathbf{c}^T \mathbf{y}$$

$$\mathbf{c} = [0,0,1]^T \text{ and } \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

```
FUNCTION F3(u)
          IMPLICIT NONE
        !
          INTEGER :: i
          REAL :: f3, u, x, y
          INTRINSIC SIN
        !
          f3 = 0.
          x = 2*u*(u+1)
          y = x + SIN(u)
          f3 = x*y
END FUNCTION F3
```

Figure 2 - Simple scalar function in Fortran

The first variation of each assignment reads:

$$\delta y_1 = \frac{\partial f_1(\mathbf{x})}{\partial u}\delta u = (4u+2)\delta u$$
$$\delta y_2 = \frac{\partial f_2(\mathbf{x}, y_1)}{\partial u}\delta u + \frac{\partial f_2(\mathbf{x}, y_1)}{\partial y_1}\delta y_1 = \cos(u)\delta u + 1\delta y_1$$
$$\delta y_3 = \frac{\partial f_3(\mathbf{x}, y_1, y_2)}{\partial u}\delta u + \sum_{j=1}^{2} \frac{\partial f_3(\mathbf{x}, y_1, y_2)}{\partial y_j}\delta y_j = 0\delta u + y_2\delta y_1 + y_1\delta y_2$$

And the code generated by Tapenade in forward mode:

$$\delta y_1 = (4u+2)\delta u$$
$$\delta y_2 = \cos(u)\delta u + 1\delta y_1$$
$$\delta y_3 = y_2\delta y_1 + y_1\delta y_2$$

Figure 3 - First variations of the three non-zero assignments, identified by variables with a suffix “d” in the forward differentiated routine generated with help of Tapenade on the right

```
FUNCTION F3_D(u, ud, f3)
          IMPLICIT NONE
        !
          INTEGER :: i
          REAL :: f3, u, x, y
          REAL :: f3_d, ud, xd, yd
          INTRINSIC SIN
        !
          f3 = 0.
          xd = 2*(ud*(u+1)+u*ud)
          x = 2*u*(u+1)
          yd = xd + ud*COS(u)
          y = x + SIN(u)
          f3_d = xd*y + x*yd
          f3 = x*y
END FUNCTION F3_D
```

Applying the reverse mode (see section 2.3.2) can create a routine calculating the gradient at once, here presented with the adjoint code generated by Tapenade:

$$y_3^* = 1 \ (\text{f3b} = 1)$$

$$\mathbf{g}_1 = \frac{\partial f_3(\mathbf{x}, y_1, y_2)}{\partial u} y_3^* = 0$$

$$y_2^* = \frac{\partial f_3(\mathbf{x}, y_1, y_2)}{\partial y_2} y_3^* = y_1$$

$$\mathbf{g}_2 = \mathbf{g}_1 + \frac{\partial f_2(\mathbf{x}, y_1)}{\partial u} y_2^* = 0 + \cos(u) y_1$$

$$y_1^* = \frac{\partial f_3(\mathbf{x}, y_1, y_2)}{\partial y_1} y_3^* + \frac{\partial f_2(\mathbf{x}, y_1)}{\partial y_1} y_2^* = y_2 + y_1$$

$$\mathbf{g}_3 = \mathbf{g}_2 + \frac{\partial f_1(\mathbf{x})}{\partial u} y_1^* = \cos(u) y_1 + (4u + 2)(y_2 + y_1)$$

$$\frac{df}{du} = \mathbf{g}_3$$

```
SUBROUTINE F3_B(u, ub, f3b)
        IMPLICIT NONE
      !
        INTEGER :: i
        REAL :: f3, u, x, y
        REAL :: f3b, ub, xb, yb
        INTRINSIC SIN
      !
        x = 2*u*(u+1)
        y = x + SIN(u)
        yb = x*f3b
        xb = yb + y*f3b
        ub = (4*u+2)*xb + COS(u)*yb

END SUBROUTINE F3_B
```

Figure 4 - Backward mode differentiated example by Tapenade. Replace f3b by 1 to obtain the expression df/du.

2.4.2 Scalar case (m=1) with vector variables (n>1): fcfoal in Edge

We show this example because it is one of the simplest among the functions used in aerodynamic shape optimization; we only display the computational loop and skip all variables declarations. The original routine (fcfoal) in Edge calculates the integrated inviscid forces and moments applied on one boundary. The original code is modified in order to be processed through TAPENADE. All data types not standard in Fortran 90 are removed, tree data structures for instance, and, instead, input parameters and output of the routine are explicitly declared as calling variables.

Example of Edge routine, after transformations necessary for being processed by Tapenade:

```
  SUBROUTINE FCFOAL_4D(pp, xx, bcsur, xm, df, dm, pref, ndim, nb, ibcn)
    IMPLICIT NONE
...
    rfdim = ndim - 2
    df = 0.
    dm = 0.
!     LOOP OVER THE BOUNDARY NODES
    DO in=1,nb
      dx = xx(ibcn(in), 1) - xm(1)
      dy = xx(ibcn(in), 2) - xm(2)
      dz = (xx(ibcn(in), ndim)-xm(3))*rfdim
      sx = bcsur(in, 1)
      sy = bcsur(in, 2)
      sz = bcsur(in, ndim)*rfdim
      pf = pp(ibcn(in)) - pref
      pm = pp(ibcn(in)) - pref
      df(1) = df(1) + pf*sx
      df(2) = df(2) + pf*sy
      df(3) = df(3) + pf*sz
      dm(1) = dm(1) + pm*(dy*sz-dz*sy)
      dm(2) = dm(2) + pm*(dz*sx-dx*sz)
      dm(3) = dm(3) + pm*(dx*sy-dy*sx)
    END DO
!
    RETURN
  END SUBROUTINE FCFOAL_4D
```

Forward-AD applied the routine FCFOAL:

```
 SUBROUTINE DFCFOAL(pp, ppd, xx, xxd, bcsur,  bcsurd, xm, df, dfd,  dm, dmd, pref,
           ndim, nb, ibcn)


              IMPLICIT NONE

….

      rfdim = ndim - 2
               df = 0.
               dm = 0.
               dfd = 0.0
               dmd = 0.0
            !    LOOP OVER THE BOUNDARY NODES
               DO in=1,nb
                 dxd = xxd(ibcn(in), 1)
                 dx = xx(ibcn(in), 1) - xm(1)
                 dyd = xxd(ibcn(in), 2)
                 dy = xx(ibcn(in), 2) - xm(2)
                 dzd = rfdim*xxd(ibcn(in), ndim)
                 dz = (xx(ibcn(in), ndim)-xm(3))*rfdim
                 sxd = bcsurd(in, 1)
                 sx = bcsur(in, 1)
                 syd = bcsurd(in, 2)
                 sy = bcsur(in, 2)
                 szd = rfdim*bcsurd(in, ndim)
                 sz = bcsur(in, ndim)*rfdim
                 pfd = ppd(ibcn(in))
                 pf = pp(ibcn(in)) - pref
                 pmd = ppd(ibcn(in))
                 pm = pp(ibcn(in)) – pref
              dfd(1) = dfd(1) + pfd*sx + pf*sxd
              df(1) = df(1) + pf*sx
              dfd(2) = dfd(2) + pfd*sy + pf*syd
              df(2) = df(2) + pf*sy
              dfd(3) = dfd(3) + pfd*sz + pf*szd
              df(3) = df(3) + pf*sz
              dmd(1) = dmd(1) + pmd*(dy*sz-dz*sy) +  pm*(dyd*sz+dy*szd-dzd*sy-dz*syd
              dm(1) = dm(1) + pm*(dy*sz-dz*sy)
              dmd(2) = dmd(2) + pmd*(dz*sx-dx*sz) + pm*(dzd*sx+dz*sxd-dxd*sz-dx*szd)
              dm(2) = dm(2) + pm*(dz*sx-dx*sz)
              dmd(3) = dmd(3) + pmd*(dx*sy-dy*sx) + pm*(dxd*sy+dx*syd-dyd*sx-dy*sxd)
              dm(3) = dm(3) + pm*(dx*sy-dy*sx)
            END DO
            RETURN
```

Backward-AD on routine FCFOAL in Edge:

```
SUBROUTINE BFCFOAL(pp, ppb, xx, xxb, bcsur, bcsurb, xm, df, dfb, &
                   dm, dmb, pref, ndim, nb, ibcn)
                  IMPLICIT NONE
                …
                  rfdim = ndim - 2
                  bcsurb = 0.0
                  xxb = 0.0
                  ppb = 0.0
                  DO in=nb,1,-1
                    pf = pp(ibcn(in)) - pref
                    sz = bcsur(in, ndim)*rfdim
                    dz = (xx(ibcn(in), ndim)-xm(3))*rfdim
                    dx = xx(ibcn(in), 1) - xm(1)
                    dy = xx(ibcn(in), 2) - xm(2)
                    pm = pp(ibcn(in)) - pref
                    sx = bcsur(in, 1)
                    sy = bcsur(in, 2)
                    tempb = pm*dmb(3)
                    pmb = (dz*sx-dx*sz)*dmb(2) + &
              (dy*sz-dz*sy)*dmb(1) + (dx*sy-dy*sx)*&
                &       dmb(3)
                    tempb0 = pm*dmb(2)
                    dxb = sy*tempb - sz*tempb0
                    sxb = dz*tempb0 + pf*dfb(1) - dy*tempb
                    tempb1 = pm*dmb(1)
                    syb = pf*dfb(2) - dz*tempb1 + dx*tempb
                    dyb = sz*tempb1 - sx*tempb
                    dzb = sx*tempb0 - sy*tempb1
                    szb = dy*tempb1 + pf*dfb(3) - dx*tempb0
                    pfb = sy*dfb(2) + sx*dfb(1) + sz*dfb(3)
                    ppb(ibcn(in)) = ppb(ibcn(in)) + pmb
                    ppb(ibcn(in)) = ppb(ibcn(in)) + pfb
                    bcsurb(in, ndim) = bcsurb(in, ndim) + rfdim*szb
                    bcsurb(in, 2) = bcsurb(in, 2) + syb
                    bcsurb(in, 1) = bcsurb(in, 1) + sxb
                    xxb(ibcn(in), ndim) = xxb(ibcn(in), ndim) + rfdim*dzb
                    xxb(ibcn(in), 2) = xxb(ibcn(in), 2) + dyb
                    xxb(ibcn(in), 1) = xxb(ibcn(in), 1) + dxb
                  END DO
                  dfb = 0.0
                  dmb = 0.0
                END SUBROUTINE BFCFOAL
```

3 Program dforce

The Fortran program dforce, in the EDGE distribution, has thus been generated with help of Tapenade from the program force and routines that calculate the DC60 (distortion). The program creates files that contain the Jacobians of the functions (drag, lift …).

The content of the file created by dforce for an inviscid calculation is obtained using the help program ffalist that summarizes the type, sizes and some first numbers for each data in FFA-format data-structures. The CPU time to obtain the Jacobians is on this example 1s and there are 134258 nodes on the wall boundary (the list "b_nodes"). Note that inviscid forces and moments only depend on the pressures, the coordinates (moments) and the surface elements vectors. The corresponding components of the Jacobians are denoted "_dro", "_du", "_dp", "_dx" and "_ds". The data structure of the file corresponds to the boundary ordering in each region. If a boundary is not a wall, the only information stored about this boundary is its name in order to facilitate detection of errors when uploading data from this file in Edge.

```
N 0 x 0      1/ jacobians
N 0 x 0      2/    region
N 0 x 0      7/       boundary
L 1 x 1      b_name= "wall"
IF 256 x 1       b_nodes =         61        62        63
DF 256 x 6       dfdm_dro        =  0.000000000000000E+000  0.000000000000000E+000  0.000000000000000E+000
DF 768 x 6       dfdm_du         =  0.000000000000000E+000  0.000000000000000E+000  0.000000000000000E+000
DF 256 x 6       dfdm_dp         = -1.507584493083414E-004  1.172207994386554E-004  1.163068918685894E-004
DF 768 x 6       dfdm_dx         =  0.000000000000000E+000  0.000000000000000E+000  0.000000000000000E+000
DF 768 x 6       dfdm_ds         =   5008.70892556716       6373.97698499611       5933.34815166263
N 0 x 0      1/       boundary
L 1 x 1      b_name= "outer_boundary"
```

In an other example the jacobians of the total forces, inviscid and viscous, are calculated for the RAE2822 airfoil, the CPU time on the same computer as in the previous case was 2,85s, for only 224 nodes on the wall boundary. This is due to the dependencies of the Jacobians on the boundary velocity and density, but also on flow and coordinates at the interior points attached to each boundary node. Those additional components of the Jacobians are denoted by an additional "i" and the indexes of the internal nodes are given under the list "bi_nodes".

```
N    0 x 0 1/ jacobians
N    0 x 0 3/    region
N    0 x 013/ boundary
L    1 x 1 b_name    = "wall"
IF 224 x 1 b_nodes   =   25  26  27
DF 224 x 6 dfdm_dro  = -6.508429988710897E-004 -0.315458254324319     -0.244189479287365
DF 672 x 6 dfdm_du   = -1.322096424261564E-005 -1.028600852743669E-002 -1.216133969346304E-002
DF 224 x 6 dfdm_dp   = -4.014931448911176E-004  3.043218476589740E-004  3.280717477480970E-004
DF 672 x 6 dfdm_dx   =  -2.30277856396984  -1907.33010066122  -1354.88229166740
DF 672 x 6 dfdm_ds   =   2123.01237957266 2912.61265541913 3320.35891083859
IF 224 x 1 bi_nodes  =  273  274  275
DF 224 x 6 dfdm_droi = -6.400226009479414E-004 -0.315350707664798     -0.244378271895287
DF 672 x 6 dfdm_dui  =  1.322096424261564E-005  1.028600852743669E-002  1.216133969346304E-002
DF 224 x 6 dfdm_dpi  =  6.785307642568260E-009  3.320159667084826E-006  2.572324683330609E-006
DF 672 x 6 dfdm_dxi  =   2.30277856396984 1907.33010066122 1354.88229166740
```

```
DF  672 x 6  dfdm_dsi   =   0.000000000000000E+000  0.000000000000000E+000  0.000000000000000E+000
N     0 x 0 1/  boundary
L     1 x 1  b_name     = "External_1"
N     0 x 0 1/  boundary
L     1 x 1  b_name     = "External_2"
```

In a third example the program dforce calculates the Jacobians of the DC60, which took 1s CPU; the time for the program to read the mesh with 13,5M nodes and the flow solution being much larger.

```
N 0 x 0       1/  jacobians
 N 0 x 0      9/     region
 N 0 x 0      1/       boundary
 L 1 x 1       b_name = "body"
 N 0 x 0      1/       boundary
 L 1 x 1       b_name = "lip"
 N 0 x 0      1/       boundary
 L 1 x 1       b_name = "fairing"
 N 0 x 0      1/       boundary
 L 1 x 1       b_name = "outlet"
 N 0 x 0      1/       boundary
 L 1 x 1       b_name = "extension"
 N 0 x 0      1/       boundary
 L 1 x 1       b_name = "farfield"
 N 0 x 0      1/       boundary
 L 1 x 1       b_name = "duct"
 N 0 x 0      1/       boundary
 L 1 x 1       b_name = "forebody"
 N 0 x 0      4/       aip_record
 IF 328 x 1 aip_nodes=     5672448     5672116     5842289
 DF 328 x 1 ddc60_dp =   0.000000000000000E+000 -4.432018250705286E-007  0.000000000000000E+000
 DF 984 x 1 ddc60_du =   0.000000000000000E+000 -1.703101059580300E-004  0.000000000000000E+000
 DF 328 x 1 ddc60_dro=   0.000000000000000E+000 -1.394572856681735E-002  0.000000000000000E+000
```

These can thus be used in Edge by the adjoint solver in order to create right-hand-sides to the adjoint equation, therefor enabling to calculate efficiently the gradients of those functions with respect to unlimited number of parameters. A limitation of the present version is that it is only differentiated in forward mode, only the inviscid forces have also been differentiated in reverse mode in order to validate the calculation of the Jacobians.

3.1 Details of implementation

The program force calls routines that calculate the inviscid and viscous forces and moments. These are the most common functionals used in optimization of aircraft external aerodynamic. Another functional is very useful for internal aerodynamic, and its calculation already implemented in Edge: the distorsion factor or DC60.

The program dforce allows creating right-hand sides of the adjoint equations corresponding to the inviscid and viscous forces and moments, as well as to the DC60.

3.1.1 Dforce

The routines present at the time of the publication of this report have been generated using backward difference only for the inviscid forces and moments, application of the backward differencing not only would involve introducing in Edge routines that are distributed by INRIA, but it so far failed to produce a code that could be run for the viscous types of functions (forces, moments and DC60). Those functions have thus been differentiated using forward difference only. This is a minor limitation for the DC60 type of function, but in case of the aerodynamic force and moments it reduces the use of dforce to small meshes because of the cost of FW. Similar difficulties have been met by others [11]. A common difficulty is the use of branch statements.

3.1.2 Edge solver

Modifications in Edge concern the adjoint flow solver as it it possible for a user to choose the origin of the right-hand side (RHS), between hand differentiated or generated by dforce. In the last case the program dforce will run, before running the adjoint solver, in order to generate an FFA-format file containing all terms of the Jacobians. The adjoint solver also allows general volume terms for the RHS such as in the case of the DC60.

4 Conclusions

The application of automatic differentiation (AD) in Edge was carried out within the NFFP project MADEF and it allowed the development of an analogue program to *force* in order to generate right-hand-sides to the adjoint code. This is in particular beneficial for those functions that have not been hand-differentiated (DC60). It also allows generating right-hand sides for any functions composed of the forces, moments and the DC60. So far the simplest way is to use the Edge toolbox in Matlab for manipulating files and data in FFA format.

However, application of AD requires many transformations of the source code even for the simplest routines; this is another important outcome of our investigations. The reason is that most routines in Edge make used of a data structures instead of passing parameters such as scalars, vectors and arrays when calling routines. This is probably the case for all large software developed for industrial use. An alternative approach to AD based on code transformation is operator overloading [8], but it seems to be essentially restricted to forward differencing, backward differentiation being thus reserved to code transformations methods. It seems that AD will never be entirely automatic, unless on the simplest codes, and that following an “adjoint-like” approach is the main guideline not only to apply AD on larger codes but also in the hope to circumvent the inherent complexity of backward differentiation [12].

The research on automatic differentiation deals with the efficiency of the transformed code (memory usagel), activation (templated code), analysis tools for AD application, reducing the complexity of the transformed code (see examples in appendix of code obtained with Matlab tools for AD), and the differentiation of mixed-language programs. Maybe one of the most needed developments of AD remains its level of automatism [11].

References

[1] Griewank A, On Automatic Differentiation, Technical report, Argonne National Laboratory, IL, November 1988

[2] Iri M., Simultaneous Computations of Functions, Partial Derivatives and Estimates of Rounding Errors – Complexity and Practicality, Japan Journal of Applied Mathematics, Vol.1, No.2, pp.223-252, 1984.

[3] Cacuci, Sensitivity Theory for Nonlinear systems, I&II, Journal of Mathematical Physics, Vol.22, No12, 1981.

[4] Christianson B. Reverse accumulation and implicit functions, Optimization Methods and Software, Vol. 9 pp.307-322, 1998

[5] Hascoet L. and Pascual V., The Tapenade Automatic Differentiation tool: principles, model, and specification. INRIA Research Report number 7957, May 2012.

[6] Griewank A. and Walther A., Evaluating Derivatives: Principle and Techniques of Algorithmic Differentiation. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008

[7] http://www.autodiff.org (list of references per application , year, codes etc …)

[8] Yu W. and Blair M., DNAD, a simple tool for automatic differentiation of Fortran codes using dual numbers, Computer Physics Communications 184, pp. 1446-1452, 2013

[9] Jones, D.P., An AD Approach using F90 and Tapenade, Queen Mary, University of London, presentation November, 2009.

[10] Amoignon O., AESOP - A numerical platform for aerodynamic shape optimization, Journal of Optimization and Engineering, vol 11, pp 555-581, 2010.

[11] Siskind J.M and Pearlmutter B.A., Putting the Automatic Back into AD: Part I, What's Wrong, School of Electrical and Computer Engineering, Purdue University, Technical Report, 2008

[12] Naumann U., Call Tree Reversal is NP-Complete, in Advances in Automatic Differentiation, pp13-22, Lecture Notes in Computational Science and Engineering, Volume 64 2008, ISBN: 978-3-540-68935-5 (Print) 978-3-540-68942-3 (Online)

Appendix

Below we show the codes produced by ADimat performing the differentiation of the Rosenbrook function in Matlab:

```
function [ r ] = anfforad_vec( x)
% Calculates value of Rosenbrook's function
   r = 100*(x(2)-x(1).^2).^2 + (1 - x(1)).^2;
```

The code result of the forward differentiation requires special routines from ADimat at runtime:

```
function [g_r, r]= g_anfforad(g_x1, x1, g_x2, x2)
  % Calculates value of Rosenbrook's function
  %   r = 100*(x(2)-x(1)^2)^2 + (1 - x(1))^2;
  g_tmp_anfforad_00000= adimat_g_pow_left(g_x1, x1, 2);
  tmp_anfforad_00000= x1^ 2;
  g_tmp_anfforad_00001= g_x2- g_tmp_anfforad_00000;
  tmp_anfforad_00001= x2- tmp_anfforad_00000;
  g_tmp_anfforad_00002= adimat_g_pow_left((g_tmp_anfforad_00001), (tmp_anfforad_00001), 2);
  tmp_anfforad_00002= tmp_anfforad_00001^ 2;
  g_tmp_anfforad_00003= 100* g_tmp_anfforad_00002;
  tmp_anfforad_00003= 100* tmp_anfforad_00002;
  g_tmp_anfforad_00004= -g_x1+ g_zeros(1);
  tmp_anfforad_00004= 1- x1;
  g_tmp_anfforad_00005= adimat_g_pow_left((g_tmp_anfforad_00004), (tmp_anfforad_00004), 2);
  tmp_anfforad_00005= tmp_anfforad_00004^ 2;
  g_r= g_tmp_anfforad_00003+ g_tmp_anfforad_00005;
  r= tmp_anfforad_00003+ tmp_anfforad_00005;
```

After reverse differentiation the code requires, as in the forward case special routines from ADimat at runtime:

```
function [a_x nr_r] = a_anfforad_vec(x, a_r)
%   r = 100*(x(2)-x(1)^2)^2 + (1 - x(1))^2;
tmpca6 = 1 - x(1);
tmpca5 = tmpca6 .^ 2;
tmpca4 = x(1) .^ 2;
tmpca3 = x(2) - tmpca4;
tmpca2 = tmpca3 .^ 2;
tmpca1 = 100 * tmpca2;
r = tmpca1 + tmpca5;
nr_r = r;
[a_tmpca6 a_tmpca5 a_tmpca4 a_tmpca3 a_tmpca2 a_tmpca1 a_x] =
a_zeros(tmpca6, tmpca5, tmpca4, tmpca3, tmpca2, tmpca1, x);
if nargin < 2
    a_r = a_zeros(r);
  end
a_tmpca1 = a_tmpca1 + adimat_adjred(tmpca1, a_r);
a_tmpca5 = a_tmpca5 + adimat_adjred(tmpca5, a_r);
a_tmpca2 = a_tmpca2 + adimat_adjmultr(tmpca2, 100, a_tmpca1);
 a_tmpca3 = a_tmpca3 + adimat_adjred(tmpca3, 2 .* tmpca3 .* a_tmpca2);
a_x(2) = a_x(2) + adimat_adjred(x(2), a_tmpca3);
a_tmpca4 = a_tmpca4 + adimat_adjred(tmpca4, -a_tmpca3);
a_x(1) = a_x(1) + adimat_adjred(x(1), 2 .* x(1) .* a_tmpca4);
a_tmpca6 = a_tmpca6 + adimat_adjred(tmpca6, 2 .* tmpca6 .* a_tmpca5);
 a_x(1) = a_x(1) + adimat_adjred(x(1), -a_tmpca6);
end
```

FOI, Swedish Defence Research Agency, is a mainly assignment-funded agency under the Ministry of Defence. The core activities are research, method and technology development, as well as studies conducted in the interests of Swedish defence and the safety and security of society. The organisation employs approximately 1000 personnel of whom about 800 are scientists. This makes FOI Sweden's largest research institute. FOI gives its customers access to leading-edge expertise in a large number of fields such as security policy studies, defence and security related analyses, the assessment of various types of threat, systems for control and management of crises, protection against and management of hazardous substances, IT security and the potential offered by new sensors.

FOI
Defence Research Agency
SE-164 90 Stockholm

Phone: +46 8 555 030 00
Fax: +46 8 555 031 00

www.foi.se