European Conference on Computational Fluid Dynamics
ECCOMAS CFD 2006
P. Wesseling, E. Oñate and J. Périaux (Eds)

THE USE OF AUTOMATIC DIFFERENTIATION FOR ADJOINT CFD CODES

Mihai C. Duta* **and Michael B. Giles†**

*Oxford University Computing Laboratory,
Wolfson Building, Parks Road
Oxford OX1 3QD, Great Britain
e-mail: Mihai.Duta@comlab.ox.ac.uk
web page: www.comlab.ox.ac.uk/mihai.duta/
†Oxford University Computing Laboratory,
Wolfson Building, Parks Road
Oxford OX1 3QD, Great Britain
e-mail: Mike.Giles@comlab.ox.ac.uk

Key words: Automatic Differentiation, Discrete Adjoint, Design Optimisation

Abstract. *Optimisation of aerospace designs uses the linear gradients of the minimised objective functionals with respect to the specified design variables. These gradients can be computed using a linearised version (or the adjoint equivalent) of the CFD code that outputs the original objective functionals. The task of generating linear or adjoint code from CFD applications benefits greatly from using Automatic Differentiation. This paper discusses some of the theoretical and practical aspects of using Automatic Differentiation on a large CFD code efficiently. Particular emphasis is put on the validation procedures to verify the correctness of the resulting linear and adjoint codes.*

1 INTRODUCTION

Design optimisation of aeronautical components is based on modelling the variation of some objective functionals of engineering merit with specified design parameters. Various design optimisation methods require the gradients of the objective functionals in the design space defined by the parameters. In the straightforward approach, these gradients can be obtained by simply linearising the CFD code that computes the objective functionals. Alternatively, the gradients can be obtained using the adjoint method, which is more efficient than the direct linearisation in the case of only a few objective functionals and of a large number of design variables.

The adjoint method in the context of aeronautical design optimisation has been pioneered by Jameson[1,2] and has become widely popular over the last decade. The adjoint approach allows the efficient computation of the gradients of a single objective functional using the solution of the adjoint system of equations at a cost independent of the number of design variables. The discrete adjoint equations can be formulated using either the "continuous" approach favoured

by Jameson or the "discrete" approach used for example by Elliott and Peraire[3]. In the "continuous" approach, the adjoint equations are defined from the continuous differential equations of flow and then discretised. On the contrary, following the "discrete" approach, the discrete flow equations are directly used to formulate the adjoint problem.

Although there is no fundamental reason to opt for one approach or the other, the authors prefer the continuous approach because of the following three practical reasons:

- The construction of the discrete adjoint problem (including the adjoint weak boundary conditions) can be mathematically described as a simple and general process, independent of the exact CFD model (e.g. turbulence model) employed in the analysis;
- The gradients of the objective functional in the design space are perfectly consistent with the functional being optimised. Due to this, the discrete adjoint code can be reliably tested against the original CFD code;
- Automatic Differentiation can be employed to substantially reduce the programming effort in developing the adjoint CFD code.

The term Automatic Differentiation (AD) designates a methodology to evaluate numerically the derivative of a functional defined by a computer program. The practical motivation for the development of AD was precisely the generation of reliable sensitivity-enhanced versions of arbitrary computer programs with little human effort. In fact, one of the earliest applications which motivated the development of the "reverse" (adjoint) mode AD was in fluid dynamics, an adjoint version of an ocean circulation model developed at MIT[4]. In the aeronautical context, the first application of AD was probably due to Mohammadi[5].

The objective of this paper is to explain some of the theoretical and practical aspects of using AD successfully for large industrial codes. First, to understand the development of the adjoint code in general terms, both the discrete adjoint problem and the functioning modes of AD methodology (direct and adjoint) are introduced formally. Then, the selective use of AD is explained to be the key to generating computationally efficient adjoint code, in preference to "black-box" application on the entire original CFD solver. Lastly, the different tests employed to validate the adjoint code generated are looked at in detail.

2 LINEAR SENSITIVITIES

In general terms, an optimisation process starts with a set of design variables α and first generates a computational grid of nodal coordinates X conforming to the design geometry described by α. Then, a discrete flow solution U is computed on this grid and a scalar objective functional J is finally obtained. Suppose a gradient based optimisation method is employed to minimise the objective functional J.

The AD community employs the dot symbol to indicate derivatives with respect to one particular design variable; *e.g.* $\dot{J}$ is the derivative of the objective functional J with respect to one component of the vector α. With this notation, the gradient of J can be expressed using the

chain differentiation rule as the product of derivatives:

$$\dot{J} = \frac{\partial J}{\partial U}\,\frac{\partial U}{\partial X}\,\frac{\partial X}{\partial \alpha}\,\dot{\alpha}.$$

This expression represents the "forward" propagation of sensitivities, using the "linear" sensitivities

$$\dot{X} = \frac{\partial X}{\partial \alpha}\,\dot{\alpha}, \quad \dot{U} = \frac{\partial U}{\partial X}\,\dot{X}, \quad \dot{J} = \frac{\partial J}{\partial U}\,\dot{U}.$$

Of course, $\dot{\alpha}$ is 1 by definition but it is still useful to use the symbol.

Using again notations followed in the AD community, the bar symbol denotes the derivative of J with respect to the quantity over which it is used; *e.g.* $\overline{\alpha}$ denotes the functional gradient $\partial J/\partial \alpha$. With this notation and using the superscript T to mean matrix (or vector) transposition, the gradient becomes

$$\overline{\alpha} = \left(\frac{\partial X}{\partial \alpha}\right)^T \left(\frac{\partial U}{\partial X}\right)^T \left(\frac{\partial J}{\partial U}\right)^T \overline{J},$$

because

$$\overline{\alpha} \stackrel{\text{def}}{=} \left(\frac{\partial J}{\partial \alpha}\right)^T = \left(\frac{\partial J}{\partial X}\,\frac{\partial X}{\partial \alpha}\right)^T = \left(\frac{\partial X}{\partial \alpha}\right)^T \overline{X},$$

and similarly

$$\overline{X} = \left(\frac{\partial U}{\partial X}\right)^T \overline{U}, \quad \overline{U} = \left(\frac{\partial J}{\partial U}\right)^T \overline{J}.$$

This calculation of the gradient is the "backward" (or "reverse") propagation of the sensitivities, which uses the "adjoint" quantities $\overline{X}$, $\overline{U}$ and $\overline{J}$.

The terms "forward" and "backward" come from the observation that whereas the linear sensitivity analysis propagates sensitivities from design variables to objective functional, *i.e.*

$$\dot{\alpha} \quad \longrightarrow \quad \dot{X} \quad \longrightarrow \quad \dot{U} \quad \longrightarrow \quad \dot{J},$$

the adjoint analysis proceeds backwards, from functional to design variables, *i.e.*

$$\overline{\alpha} \quad \longleftarrow \quad \overline{X} \quad \longleftarrow \quad \overline{U} \quad \longleftarrow \quad \overline{J}.$$

Given the above definitions, the sensitivity of the output functional J to the design variables α can be evaluated in a number of ways:

$$\dot{J} = \overline{U}^T \dot{U} = \overline{X}^T \dot{X} = \overline{\alpha}^T \dot{\alpha}.$$

It is thus possible to proceed through a part of the forward process above and combine the result of that with going through the other part of the backward process. This is illustrated in Fig. 1, in which the dashed lines represent the stages at which the sensitivity dot products in the expression above can be obtained. This observation is particularly useful in applications in which part of the process is a black-box which cannot be touched. For example, if the step $\alpha \rightarrow X$ involves a proprietary CAD system or grid generator, then the only option may be to approximate the forward mode linear sensitivity $\dot{X}$ through a central finite difference using $X(\alpha \pm \Delta\alpha)$ and obtain the objective gradient as $\overline{X}^T \dot{X}$.

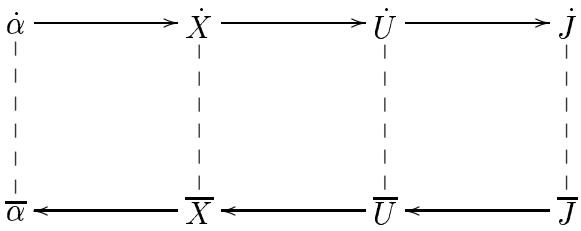

Figure 1: Diagram to illustrate the possible ways in which the sensitivity of the objective functional can be computed.

3 LINEAR AND ADJOINT EQUATIONS

In a steady-state CFD model, the vector of discrete flow quantities U is the solution of a set of nonlinear discrete flow equations of the form

$$N(U, X) = 0, \tag{1}$$

where the vector of discrete nonlinear residuals N depends on the nodal coordinates as well as on the flow quantities. As a results of this, the flow solution U is an implicit function of the coordinates X. The set of equations (1) is normally solved using an iterative process and the preconditioned Runge–Kutta is a particularly popular method.

Supposing the sensitivity $\dot{X}$ of the grid coordinates to the design variables are known, Eqn. (1) is differentiated to obtain the linear system

$$L\,\dot{U} + \dot{N} = 0, \tag{2}$$

with the matrix and r.h.s. respectively defined by

$$L = \frac{\partial N}{\partial U}, \qquad \dot{N} = \frac{\partial N}{\partial X}\,\dot{X}.$$

The solution $\dot{U} = -L^{-1}\dot{N}$ of the linear system (2) is the sensitivity of the steady flow to changes in the design variables α. Introducing this into the expression for the gradient of J, the following identities result

$$\dot{J} = \overline{U}^T\dot{U} = \overline{U}^T(-L^{-1}\dot{N}) = (-(L^T)^{-1}\overline{U})^T\dot{N} = \overline{N}^T\dot{N},$$

where the vector of adjoint quantities $\overline{N}$ is the solution to the adjoint system

$$L^T\overline{N} + \overline{U} = 0. \tag{3}$$

It results again the gradient $\dot{J}$ can be computed in two modes, starting from the linear solution $\dot{U}$ or from the ajoint $\overline{N}$. Obtaining $\dot{J}$ as the dot product $\overline{N}^T\dot{N}$ can be seen as an extra stage in Fig. 1, with the intermediary calculation of $\dot{N}$ between $\dot{X}$ and $\dot{U}$ and the corresponding $\overline{N}$ in the backward sequence.

The linear and adjoint systems of equations are solved using iterative solution methods which are derived from the nonlinear Runge–Kutta process but are not described here. While it suffices for this paper to mention the adjoint iterations are the algebraic adjoint (*i.e.* transpose) of the linear, see Giles[6] for an algorithmic discussion of the adjoint iterative process.

It is also assumed the discrete nonlinear Eqn. (1) include the weak boundary conditions, therefore Eqns. (2) and (3) naturally include the correct linear and adjoint boundary conditions, respectively. For a description of the linear and adjoint boundary conditions and a detailed discussion of the effect of hard boundary conditions at solid walls, see Duta[7].

4 AUTOMATIC DIFFERENTIATION

Any of the discrete quantities involved in the sensitivity analysis introduced in the previous sections is the output of a computer program. Therefore, the sensitivity of such a quantity with respect to the input to that program is defined by a linearisation of the computer program. Code linearisation yields sensitivities that are consistent with the original discrete quantity and is preferred to alternatives (such as finite differencing) whenever the source code is available.

The programming effort of linear and adjoint code generation is greatly reduced by using AD technology. In principle, AD software is designed to transform any computer program into new code that computes sensitivities in either the forward or reverse mode. In practice, the selective use of AD targeting specific code sequences rather than "black-box" application to an entire code leads to more efficient compurter code, especially in the reverse mode.

In order to understand how AD works, each step in the execution of the computer program can be described as a function of two values; *e.g.* the result of addition is a function of two operands and unitary functions such as $\exp(x)$ are binary function with no dependence on the second parameter. Consequently, a program starts with a number of input variables $u_i, i = 1, \ldots I$, represented collectively as the input vector $\mathbf{u}^0$, and each step adds a new active variable to this initial vector, with its value given by a binary function. Mathematically, the execution of the nth step can be written as

$$\mathbf{u}^n \;=\; \mathbf{f}^n(\mathbf{u}^{n-1}) \;\equiv\; \begin{pmatrix} \mathbf{u}^{n-1} \\ f_n(\mathbf{u}^{n-1}) \end{pmatrix}, \tag{4}$$

where f_n is a scalar function of two of the elements of $\mathbf{u}^{n-1}$. The dimension of vector $\mathbf{u}^n$ is that of vector $\mathbf{u}^{n-1}$ plus 1. The result of all the N steps of the complete program can be expressend as a composition of individual functions applied to the input vector

$$\mathbf{u}^N = \mathbf{f}^N \circ \mathbf{f}^{N-1} \circ \ldots \circ \mathbf{f}^2 \circ \mathbf{f}^1(\mathbf{u}^0). \tag{5}$$

Now, define $\dot{\mathbf{u}}^n$ to be the derivative of the vector $\mathbf{u}^n$ with respect to one particular element of $\mathbf{u}^0$, *i.e.* to one particular input variable. Differentiating (4),

$$\dot{\mathbf{u}}^n = L^n \, \dot{\mathbf{u}}^{n-1} \tag{6}$$

where the step sensitivity L^n is

$$L^n = \frac{\partial \mathbf{u}^n}{\partial \mathbf{u}^{n-1}} = \begin{pmatrix} I^{n-1} \\ \frac{\partial f_n}{\partial \mathbf{u}^{n-1}} \end{pmatrix},$$

where I^{n-1} is the identity matrix of dimension equal to that of the vector $\mathbf{u}^{n-1}$. Then, the derivative of (5) gives the sensitivity of the output vector $\mathbf{u}^N$ with respect to one particular element of the input vector as a product of sensitivities

$$\dot{\mathbf{u}}^N = L^N \, L^{N-1} \ldots L^2 \, L^1 \, \dot{\mathbf{u}}^0. \tag{7}$$

Here, the elements of $\dot{\mathbf{u}}^0$ are all zero except for a unit value corresponding to the particular element whose sensitivity is of interest. Equation (7) can be regarded as the propagation of the initial sensitivity $\dot{\mathbf{u}}^0$ through the whole linearised program using the step sensitivities L^n. If one is interested in the sensitivities $\dot{\mathbf{u}}^N$ with respect to N_I different input elements, the expression (7) must be evaluated for each case, which makes the total evaluation cost proportional to N_I.

The reverse mode of AD sensitivity calculation can be described using the column vector $\overline{\mathbf{u}}^n$, which is the derivative of a particular element u_i^N of the output vector with respect to the elements of $\mathbf{u}^n$. First, the adjoint equivalent to (6) is obtained by chain differentiation

$$\overline{\mathbf{u}}^{n-1} \;=\; \left(\frac{\partial u_i^N}{\partial \mathbf{u}^{n-1}}\right)^T \;=\; \left(\frac{\partial u_i^N}{\partial \mathbf{u}^n}\,\frac{\partial \mathbf{u}^n}{\partial \mathbf{u}^{n-1}}\right)^T \;=\; \left((\overline{\mathbf{u}}^n)^T \, L^n\right)^T \;=\; (L^n)^T \, \overline{\mathbf{u}}^n.$$

From this, the sensitivity of the output with respect to one particular input is given by

$$\overline{\mathbf{u}}^0 = \left(L^1\right)^T \, \left(L^2\right)^T \ldots \left(L^{N-1}\right)^T \, \left(L^N\right)^T \, \overline{\mathbf{u}}^N. \tag{8}$$

This expression is the adjoint of (7) and computes the same sensitivity: $\overline{\mathbf{u}}^0 = \dot{\mathbf{u}}^N$.

There are two important aspects to notice in relation with Eqn. (8). First, the reverse mode calculation represented by this expression proceeds backwards from the last step of the original program $n = N$ to the first $n = 1$. However, the linear operators L^n are sensitivities of the original steps and must be computed in forward mode, from $n = 1$ to $n = N$. Therefore, it is necessary to first perform the original calculation forwards, storing all of the partial derivatives needed for L^n, before doing the reverse mode calculation. The first part of the AD generated program is called the "forward sweep" and replicates the original program, with the original operations embedded in the original control, at the same time storing the partial derivatives required by the second part. The second part is called the "reverse sweep" and represents the adjoint calculations proper as it reverses the original flow control, from the last operation to the first, replacing each operation by its reverse derivative correspondent.

The second important aspect justifies the effort to develop adjoint codes. If the sensitivities of N_O different output elements are sought, then (8) must be evaluated for each one, at a cost which is proportional to N_O. Then, as the cost of the forward mode is proportional to N_I, it is

clear that the reverse mode is computationally more efficient than the forward mode when N_O is much smaller than N_I. This is precisely the case in aerospace design, where required are the gradients of a few objective functionals defined in a large dimensional design space.

The purpose of AD tools is to automate the process of obtaining the gradients in either the forward or the reverse mode. There is a growing AD literature and one can start with the community website at `www.autodiff.org`, which includes links to all of the major groups working in the field. A good reference for the theoretical background is due to Griewank[8] but a more practical short introduction in the context of optimisation is due to Mohammadi[9]. It is enough here to mention that there are two basic approaches to AD implementation: *o*perator overloading and *s*ource transformation. Operator overloading (a feature familiar to programmers in C++) is the augmentation of the the mathematical operators and intrinsic functions so that, in addition to the original calculations, they also produce the derivatives of all the active variables. In the reverse mode, overloading involves a process known as "taping" which first records all the values of the partial derivatives in the original calculations and then performs the reverse mode calculations using these values[10]. The second approach is arguably more suitable for large, industrial codes and generates a new computer program (in direct or reverse mode) from the original program to perform all the necessary calculations[4]. Unlike operator overloading, source transformation generates new code that is compiled independently of the original and leaves the programmer more flexibility to control the AD process in order to obtain computationally efficient adjoint code[11].

5 AN APPLICATION OF AUTOMATIC DIFFERENTIATION

The work presented here used the AD engine Tapenade, which is being developed by Hascoët and Pascual at INRIA[12,13]. Currently, the software applies source transformation to codes written in FORTRAN77 only but work is in progress to extend the software to C and C++. Tapenade is written in Java and can be installed locally as a set of Java classes and run by a simple command line which can be included into a Makefile.

Tapenade was applied to the FORTRAN77 turbomachinery flow solver HYDRA, which approximates the Reynolds-averaged Navier–Stokes equations on unstructured hybrid grids, using an edge-based discretisation[14,15]. The solution procedure solves the discrete nonlinear equations using Runge–Kutta time-marching accelerated by Jacobi preconditioning and multigrid, with dual-timestepping for unsteady flows. The task was to use Tapenade to generate the steady linear and adjoint solvers from the steady version of HYDRA in order to extract the functional sensitivities to changes in the geometry and grid. This was done in preparation for turbomachinery steady-state design optimisation. There are two important aspects in applying AD to a code of this magnitude: a re-structuring of the code and the preparation of a Makefile to store the AD instructions. Both aspects are discussed in this section.

Although AD technology can be used on codes of any size, some initial preparation is recommended for large codes. The minimal code preparation involves the replacement (or only re-arrangement) of those instructions in the original program that lead to inefficient reverse mode code through the AD process. The performance of adjoint AD code can be substantialy

improved by such changes, *e.g.* the replacement of a `max` statement by an equivalent `if` control; a list of typical situations is given by Müller[11].

In addition to this, the authors recommend further code preparation according to the following two steps:

- the re-structuring of the original code in order to separate the floating point calculations (*e.g.* flux contributions) from the framework of the code (*e.g.* loops over edges);
- the creation of a code framework for the linear and adjoint codes, based on the existing framework of the original code.

In greater detail, the first step involves the re-structuring of the routines computing the edge and boundary flux contributions to the discrete nonlinear residual. This is done by isolating the calculations at the edge or boundary node level from the rest of the routines into new routines. The idea is depicted in Fig. 2, where the operations originally performed within the loop over the grid edges are isolated from the loop and performed within a new routine. Doing this, all the code instructions that need to be submitted to AD are isolated from the rest of the code. The new routines represent the bulk of the computational load so, as a result of the first step, the restructured program consists of a "lightweight" framework (`do` loops, mainly) in which the routines that perform the calculations are embedded and controlled.

```
do ie = 1, nedge
  operation_1
  operation_2
  ...
enddo
```

```
do ie = 1, nedge
  call edge_routine(...)
enddo

subroutine edge_routine(...)
operation_1
operation_2
...
return
end
```

Figure 2: Restructuring example: the operations within a simple edge loop are isolated into a separate routine.

The first step has a practical reason; the selective aplication of AD only to the new routines (performing the operations at edge or node level) leads to an easy management of the AD process. Thus, the AD code transformation can be controlled through a Makefile containing all the Tapenade commands to transform the routines and any change in the code (*e.g.* a change in the characteristic smoothing technique) can be directly AD transformed with no change to the Makefile.

The second step adapts the nonlinear code framework resulting at the first step to the emerging linear and adjoint codes. Whereas the linear code framework is straightforward to implement, having almost the same structure as that of the nonlinear, programming the framework of the adjoint code requires understanding at the algorithmic level. Programming the framework of the HYDRA adjoint code demanded particular attention in the following areas:

- the hard boundary conditions at the solid walls need to be treated separately from the weak conditions which are handled directly through AD[7];
- although the Runge–Kutta iteration can be used unmodified to time-march the adjoint equations to convergence, employing the true adjoint of the Runge–Kutta algorithm has advantages[16];
- the linear multigrid restriction and prolongation routines can be used directly for the adjoint solver but they exchange roles: the linear prolongation becomes the adjoint restriction and *vice-versa*[6];
- the calling sequence of the flux contribution routines in the nonlinear code is reversed in the adjoint code as a consequence of the reverse mode operation, *e.g.* whereas the nonlinear code first computes flow gradients and then the smoothing viscous fluxes, the adjoint code correspondents are called in reverse sequence.

So, by separating the general code framework from the calculations, control is retained in the adjoint code over all the algorithmic features that are understood at a theoretical level. Then, the AD targets only the low-level flux contribution routines, which are, in fact, the code sequences where AD is needed.

6 PRACTICAL ASPECTS

The generation of the linear and adjoint versions of the nonlinear edge and node level routines is managed automatically by a Makefile. The Makefile first invokes Tapenade to AD transform the nonlinear source code, specifying the appropriate independent and dependent variables in each routine and then compiles the AD generated code to object files. The AD generated code is not needed, except perhaps to gain some insight on how Tapenade works, so it is routinely deleted after compilation.

The only delicate part of using Tapenade is correctly specifying the status of the active variables in each routine. There are three cases:

- *input only* if the variable is never used again after the routine finishes, therefore its output value is irrelevant;
- *output only* if the variable is assigned a value within the routine, therefore its input value is irrelevant;

- *input and output* if the "input" variables, such as the flow variables, are used again later, and "output" variables, such as the flux residual, are in fact increments added to pre-existing values. This is the the most common case in practice.

To generate the linear code, the nonlinear routines were linearised twice, once with respect to the flow variables and a second time with respect to grid coordinates. This was necesary in order to generate the linear sensitivities L and $\dot{N}$ in Eqn. (2). In order to cover both cases, the Makefile has separate sets of Tapenade commands, with different independent variables specified in each case. Due to its definition, there are no sensitivities to the grid coordinates in the adjoint equations (3), so the Makefile contains only one Tapenade command for each adjoint AD routine as opposed to two in the linear case.

7 VALIDATION TESTS

Although Tapenade produces correct transformed code, it is good porgramming practice to design a suite of validation tests in order to verify that the AD generated code performs as expected within the linear and adjoint code framework. There are two main sources of programming errors: first, the status of the active variables can be incorrectly specified in the Makefile and second, the AD generated low-level routines can be mis-assembled within the code framework. These make testing particularly desirable in the case of a large code like HYDRA. Fundamentally, there are two levels of validating tests, one aimed at individual routines or assemblies of routines and the other at the complete linear and adjoint codes.

Individual routines are tested in two ways. First, the consistency of the linear routines (or assembly of routines) with their correspondents in the original code is tested using complex Taylor series[17,18]. If f is assumed to be a complex analytic function which is real-valued along the real axis, a Taylor series expansion of f along the imaginary axis but about a real-valued u gives

$$\lim_{\epsilon \to 0} \frac{\mathcal{I}\{f(u+\mathrm{i}\epsilon\dot{u})\}}{\epsilon} = \frac{\partial f}{\partial u}\,\dot{u}.$$

Thus, the sensitivity of f at u along the real axis is given by the imaginary part of the value of f at u perturbed with a vanishingly small imaginary value. The convergence to the sensitivity value is quadratic as for a central finite difference (FD) approximation but, whereas the FD approach suffers from cancellation errors at small perturbations, the complex Taylor series can work with very small values, its accuracy being limited only by machine precision. These properties are illustrated for a simple exp function in Fig. 3.

This method is excellent as a testing tool as very accurate and reliable value for sensitivities are obtained directly from the original nonlinear code. Applying this technique to a FORTRAN code requires little more than replacing all double precision declarations `REAL*8` by `COMPLEX*16`, and defining appropriate complex analytic versions of the intrinsic functions `min`, `max` and `abs`. The sensitivies obtained from the complex Taylor series are used to validate the linear routines. Notice that, despite the attraction of its simplicity, the method cannot be successfully used to obtain sensitivities at the level of a full code because the computational

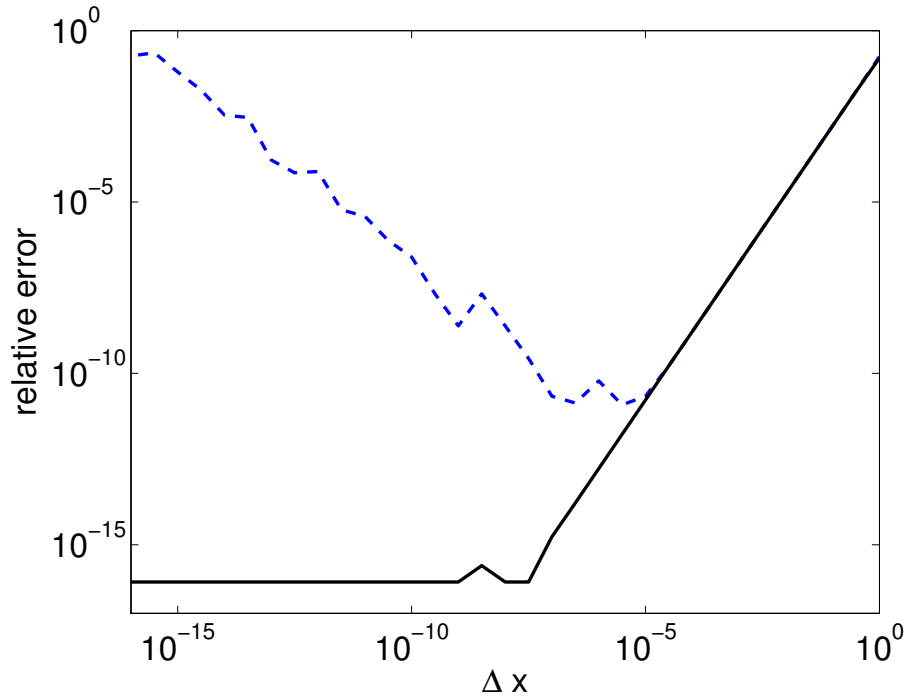

Figure 3: Variation of the relative error in approximating the derivative of a simple function ($\exp(x)$) through central FD (blue dashed line) and the complex Taylor series (black continuous line). The FD error decreases quadratically over a range of perturbation values but deteriorates for too small perturbations because of cancellation errors. The error of the complex Taylor approximation decreases quadratically down to machine epsilon, by which its limited.

penalties are too great. The execution of the nonlinear code in `COMPLEX*16` mode is substantially slower that that of the AD linearised code and the method is a poor alternative to AD.

The second set of tests for individual routines checks the adjoint routines against their linear correspondent. Conceptually, these tests are based on:

- a linear routine which computes the linear sensitivity $\left(\dfrac{\partial f}{\partial u}\right)\dot{u}$;
- the adjoint correspondent which computes the adjoint sensitivity $\left(\dfrac{\partial f}{\partial u}\right)^T \overline{f}$.

By calling the linear and adjoint routines with input vectors $\dot{u}$ and $\overline{f}$ that are zero except for one unit entry, the sensitivity matrices $\partial f/\partial u$ and its transpose can be independently constructed from the linear and adjoint routines and compared.

The final tests are made at the level of the complete codes:

- a nonlinear code which computes $J(\alpha)$;

- a linear code which computes $\dot{J}$;
- an adjoint code which also computes $\dot{J}$.

Again, the functional sensitivity output by the linear code is verified for consistency with the original nonlinear code. It is impractical to apply the complex Taylor series method to the entire nonlinear HYDRA code even only for testing purposes because this requires too much code intervention. Instead, the validation test compares $\dot{J}$ with the finite difference approximation

$$\frac{1}{2\Delta\alpha}\left(J(\alpha+\Delta\alpha) - J(\alpha-\Delta\alpha)\right).$$

The optimal magnitude of the perturbation $\Delta\alpha$ which gives the best accuracy in the FD aproximation is case dependent and it is good practice to try several values over a range of magnitude orders in order to find it. However, it can be accepted as a rule of thumb that, at a unit reference length in the geometry/grid size, a perturbation of 10^{-6} is a good guess.

Further to be tested is whether the linear and adjoint codes produce the same value for $\dot{J}$. Obviously, the values should agree to within machine accuracy at full convergence of the linear and adjoint solution. Also, because the adjoint HYDRA solver is the reverse mode of the linear one, the values must agree after a fixed number of iterations of the two solvers. This is a very helpful debugging feature for a large code with long execution times, as checking for full agreement of functional sensitivities after a few iterations bears a low cost while revealing most programming errors.

8 VALIDATION EXAMPLE

The linear and adjoint HYDRA codes were verified using all the tests described in the previous section. One of the tests at the level of the complete codes was carried out on the Nasa Rotor 37 turbomachinery testcase[19]. This testcase is a highly loaded transonic compressor fan blade geometry in viscous turbulent flow (the Spalart–Allmaras turbulence model was used), so the correct AD treatment of many of the HYDRA modelling capabilities were tested.

The objective functional used was the massflow through the entire annulus, whose computed value at the choking point is 20.85 kg/s. The initial rotor geometry, depicted in Fig. 4, was perturbed by twisting the blade at midheight by an angle $\Delta\alpha$ measured at leading edge, the twist gradually vanishing to zero towards hub and casing. This perturbation has an effect on the entire grid, including the outflow boundary, so the correct linearisation of the boundary conditions is tested too. 6 different values were chosen for the twist angle: $\pm 10^{-6}$, $\pm 10^{-5}$ and $\pm 10^{-4}$ degrees (the tip diameter of the rotor blade at leading edge is 0.5074 m). The values of the massflow as output by the nonlinear HYDRA solver on the perturbed grids as well as on the initial geometry are tabulated in Table 1.

The values in Table 1 were obtained from flow solutions fully converged down to machine accuracy. The variation of massflow is almost linear with the chosen twist angle perturbation, which makes it ideal for validation as the FD approximation is accurate in this case. The massflow sensitivity estimated as a FD from the values obtained at $\Delta\alpha = \pm 10^{-6}$

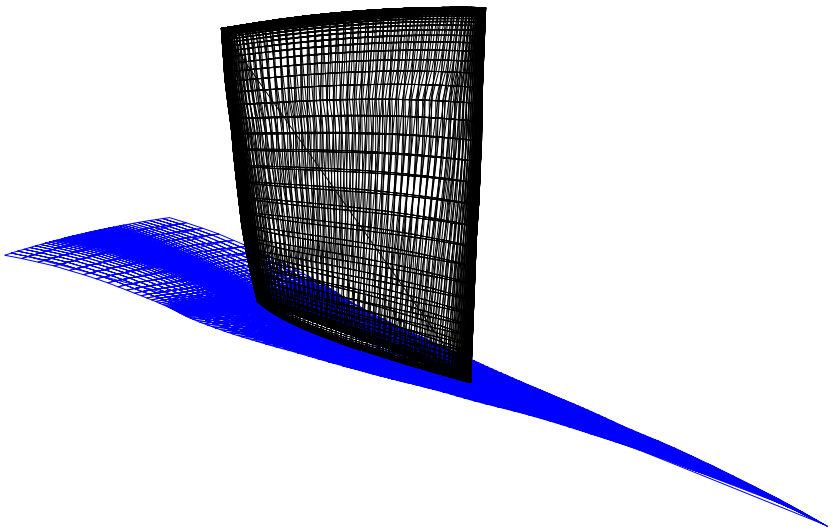

Figure 4: The geometry of the Nasa Rotor 37 blade (black), also showing the hub (blue).

twist angle (degrees)	**massflow** (kg/s)
-1.e-4	20.8493180070651
-1.e-5	20.8493181280627
-1.e-6	20.8493181401630
0	20.8493181415069
1.e-6	20.8493181428519
1.e-5	20.8493181549520
1.e-4	20.8493182759513

Table 1: Variation of the massflow output by nonlinear HYDRA with the twist angle at blade midspan.

is $1.344457 \cdot 10^{-9}$ kg/s and that obtained from the linear calculation is $1.344464 \cdot 10^{-9}$ kg/s, so the agreement is to within around $5.27 \cdot 10^{-6}$ relative error.

This level of agreement validates the entire linear code against the original nonlinear HYDRA solver. Also, the massflow sensitivity output by the adjoint code agreed with the value from the linear calculation to within machine accuracy, both at full convergence and after a common arbitrary number of iterations.

9 CONCLUSIONS

This paper discussed the main aspects of applying the AD technology to a large industrial CFD turbomachinery solver in order to generate linear and adjoint code to be used in design optimisation. The theoretical background of both the adjoint equations and AD technology was briefly presented. The practicalities of using the AD software Tapenade were also discussed.

It was argued that the arrangement best suited for a large CFD code is to separate the floating point operations at the level of grid edges or nodes from the flow control (mainly do loops) in the nonlinear code. In this way, the nonlinear code consists of a "lightweight" control framework and of low-level routines containing most of the floating point calculations. Then, the linearisation of the CFD code has two stages: a) the creation of similar frameworks for the linear and adjoint codes and b) the AD source transformation of the low level routines.

The purpose of the first stage is to retain control on the structure of the adjoint solver. While the nonlinear control framework is copied with little alteration to the linear code, the framework of the adjoint code needs an algorithmic understanding of the adjoint method. The resulting linear and adjoint CFD turbomachinery codes have a fixed hand-coded structure consisting of the Runge–Kutta smoother, the multigrid iterations and the edge and node do loops.

The second stage of the linearisation uses the AD software to generate the low-level linear and adjoint routines from the original nonlinear. The AD transformation is carried out through Tapenade commands controlled by a Makefile and the whole process of generating the linear and adjoint codes from nonlinear source code to executables becomes completely automatic. Tapenade handles without any difficulty the complicated nonlinearities and conditional branching in the turbulence modelling and the characteristic-based numerical smoothing. The use of AD is essential in keeping the linear and adjoint codes consistent with the latest incremental changes to the nonlinear code.

It is the author's view that validation is an important part of the linearisation work and the principles of testing the linear and adjoint codes were also discussed. A hierarchy of tests was prescribed, ranging from testing the low level routines to verifying the correct programming of the control frameworks of the complete linear and ajoint codes.

Acknowledgements

This research was performed as part of the MCDO project funded by the UK Department for Trade and Industry and Rolls-Royce plc, and coordinated by Yoon Ho, Leigh Lapworth and Shahrokh Shahpar. We are grateful to Laurent Hascoët for making Tapenade available to us, and for being responsive to our queries.

REFERENCES

[1] A. Jameson. Aerodynamic design via control theory. *J. Sci. Comput.*, **3**:233–260, (1988).

[2] A. Jameson, N. Pierce, and L. Martinelli. Optimum aerodynamic design using the Navier-Stokes equations. *J. Theor. Comp. Fluid Mech.*, **10**:213–237, (1998).

[3] J. Elliott and J. Peraire. Practical 3D aerodynamic design and optimization using unstructured meshes. *AIAA J.*, **35(9)**:1479–1485, (1997).

[4] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Trans. Math. Software*, **24(4)**:437–474, (1998).

[5] B. Mohammadi and O. Pironneau. Mesh adaption and automatic differentiation in a CAD-free framework for optimal shape design. *Internat. J. Numer. Methods Fluids*, **30(2)**:127–136, (1999).

[6] M. B. Giles, M. C. Duta, J.-D. Müller and N. A. Pierce. Algorithm developments for discrete adjoint methods. *AIAA J.*, **41(2)**:198–205, (2003).

[7] M. C. Duta. *The Use of the Adjoint Method for the Minimisation of Forced Vibration in Turbomachinery*. Ph.D. Dissertation, University of Oxford, United Kingdom, (2002).

[8] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, (2000).

[9] Bijan Mohammadi and Olivier Pironneau. *Applied Shape Optimization for Fluids*, OUP, (2001).

[10] A. Griewank, D. Juedes, and J. Utke. ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, **22(2)**:437–474, (1996).

[11] J.-D. Müller and P. Cusdin. On the performance of discrete adjoint CFD codes using automatic differentiation. *Int. J. Numer. Meth. Fluids*, **47**:939–945, (2005).

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

[13] L. Hascoët and V. Pascual. *Tapenade 2.1 user's guide*. Technical Report 0300, INRIA, (2004), `http://www-sop.inria.fr/tropics/`.

[14] P. Moinier. *Algorithm developments for an unstructured viscous flow solver*. Ph.D. Dissertation, University of Oxford, United Kingdom, (1999).

[15] P. Moinier, J.-D. Müller and M. B. Giles, Edge-based multigrid and preconditioning for hybrid grids, *AIAA Journal*, **40(10)**:1954–1960, (2002).

[16] M.B. Giles. On the iterative solution of adjoint equations. In G. Corliss, C. Faure, A. Griewank, L. Hascoët and U. Naumann, editors, *Automatic Differentiation: From Simulation to Optimization*, Springer–Verlag, (2001).

[17] W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Rev.*, **10(1)**:110–112, (1998).

[18] W.K. Anderson and E. Nielsen. Sensitivity analysis for Navier-Stokes equations on unstructured grids using complex variables. *AIAA J.*, **39(1)**:56–63, 2001.

[19] L. Reid and R. D. Moore *Design and overall performance of four highly loaded, high-speed inlet stages for an advanced high-pressure-ratio core compressor*, Technical Report NASA TP 1337, (1978).