TURBOMACHINERY DESIGN OPTIMIZATION USING AUTOMATIC DIFFERENTIATED ADJOINT CODE

Mihai C. Duta
University of Oxford
Computing Laboratory
Parks Road, Wolfson Building
Oxford, OX1 3QD, Great Britain
Email: Mihai.Duta@comlab.ox.ac.uk

Shahrokh Shahpar
Aerothermal Methods Group
Rolls-Royce Plc.
Derby, DE21 2XL, Great Britain
Email: Shahrokh.Shahpar@rolls-royce.com

Michael B. Giles
University of Oxford
Computing Laboratory
Parks Road, Wolfson Building
Oxford, OX1 3QD, Great Britain
Email: Mike.Giles@comlab.ox.ac.uk

ABSTRACT
The last decade has established the adjoint method as an effective way in Computational Fluid Dynamics of calculating the gradients of an objective functional in a large dimensional design space. This paper addresses the concerns that code developers face when creating a discrete adjoint computer program for design optimization, starting from a nonlinear flow solver and using Automatic Differentiation. Adjoint code development benefits greatly from using Automatic Differentiation but at its current state of maturity, this technology is best applied selectively rather than on entire codes. The paper discusses the practical aspects of using Automatic Differentiation on a large industrial turbomachinery flow solver with the objective of deriving efficient adjoint code. The use of the adjoint gradients is illustrated in an optimization exercise using gradient based methods on the NASA Rotor 37 public testcase.

NOMENCLATURE
\( \alpha \) (vector of) design variable(s)
\( X \) vector of mesh nodal coordinates
\( U \) vector of nodal flow variables
\( J \) (vector of) objective functional(s)
\( \bar{q} \) adjoint sensitivity \( \partial J / \partial q \)
\( N \) vector of discrete flow equation residuals

\[ L \] matrix operator of the linearized flow equations

INTRODUCTION
Design optimization of turbomachinery systems is a complex iterative process which uses computer models to analyze the physics of an existing design in order to modify it for an improved performance. The modeling of the fluid flow past the compressor and turbine components using Computational Fluid Dynamics (CFD) is the most important element in turbomachinery optimization. Optimization is normally performed using one or several objective functionals, which are figures of engineering merit (e.g. stage efficiency or total pressure loss), extracted from the computed flow field.

The ability to compute the gradients of the objective functionals in the design space is of great importance. Gradients can assist the optimization process in two ways; on the one hand, they are fundamental to gradient-based local optimizers and, on the other hand, they can enhance surrogate models of the functional variation in the design space used in expensive global search methods.

The adjoint method is a computationally efficient way of calculating the gradients of an objective functional in a large dimensional space at a cost independent from the number of design variables. The adjoint method in the context of aeronautical design optimization has been pioneered by Jameson [1, 2] and has become widely popular over the last decade. Although the method is usually associated with gradient based optimiz-
tion methods, it has recently been employed to improve surrogate models used by global algorithms [3, 4], for instance to enhance Kriging models [5] or even neural network models [6, 7].

The adjoint method formulated in this work follows the discrete approach, whereby the discrete flow equations are linearized and transformed to output the adjoint gradients [8]. The other option is the continuous approach, which first formulates the continuous adjoint flow equations and then discretizes them to compute the adjoint gradients. Although there is no fundamental reason to opt for one particular approach, the existence of Automatic Differentiation (AD) software to facilitate adjoint transformation at code level makes a strong case for the discrete derivation.

The term AD designates a methodology to evaluate numerically the derivative of a functional defined by a computer program. The practical motivation for the development of this technique was precisely the generation of reliable sensitivity-enhanced versions of arbitrary computer programs with minimal human effort.

This paper has two main objectives: a) to explain the practical aspects of using AD successfully and b) to illustrate the use of the discrete adjoint method on a practical example.

Drawing on experience gained from AD transforming a large industrial CFD code, the first part of the paper discusses in detail the selective use of AD as the key to generating computationally efficient adjoint code. With the present state of the technology, “black-box” application of AD to an entire code can lead to very poor adjoint performance. Although this situation is improving continuously with the research maturing, the guidelines offered in this paper remain valid as a principle. The central idea discussed is the separation of the structure of the adjoint code (e.g. edge loops, parallel execution instructions, the iterative solution process) from the low-level routines performing the floating point operations. While the adjoint code structure involves the minimum of hand-coding, the adjoint low-level routines are generated by AD without human intervention. In this way, an optimum is achieved between the efficiency of hand-coded adjoint code and the ease of using AD.

To illustrate the use of the adjoint gradients, the last part of the paper presents an optimization exercise carried out on the NASA Rotor 37 public domain testcase.

THE ADJOINT METHOD

The adjoint method can be introduced in many ways, ranging from the practical [8, 9] to the theoretical [10]. This section explains the principle of applying the discrete adjoint method to CFD from the perspective of AD. To understand the method, consider first the distinct steps involved in a generic CFD calculation. Such a calculation starts with a set of design variables represented by the vector \( \alpha \) and generates a computational mesh which conforms to the design geometry described by \( \alpha \), having the nodal coordinates given by the vector \( X \). Then, the discrete flow quantities \( U \) at the mesh nodes are computed by the flow solver and a set of scalar objective functional values \( J \) are obtained at the post-processing step. The adjoint method can compute the gradients of the functionals \( J \) in the design space defined by the variables \( \alpha \) at a cost independent of the dimension of the design space.

Conceptually, the sensitivity analysis of the effect of a single design variable \( \alpha \) on a single objective functional \( J \) requires the propagation of a perturbation to \( \alpha \) through all the steps of the CFD calculation. Mathematically, this can be expressed using the dot symbol of the AD community to indicate derivatives with respect to one particular design variable. For instance, \( J \) is the derivative of the objective functional with respect to the scalar \( \alpha \). With this notation, the gradient is the product of partial derivatives

\[
J = \frac{\partial J}{\partial \alpha} = \frac{\partial J}{\partial U} \frac{\partial U}{\partial X} \frac{\partial X}{\partial \alpha}, \tag{1}
\]

which simply expresses the chain rule differentiation of \( J \). This represents the so-called “forward” propagation of sensitivities and involves the following “linear” sensitivities:

\[
\dot{X} = \frac{\partial X}{\partial \alpha} \dot{\alpha}, \quad \dot{U} = \frac{\partial U}{\partial X} \dot{X}, \quad \dot{J} = \frac{\partial J}{\partial U} \dot{U}. \tag{2}
\]

Of course, \( \dot{\alpha} = 1 \) by the definition of the dot symbol and \( \dot{J} \) is the effect of a unit linear perturbation on \( \alpha \).

AD notations are employed again to write the gradient in a different way. Thus, the bar symbol denotes the derivative of \( J \) with respect to the quantity over which the symbol is used, e.g. \( \overline{\alpha} \) is the functional gradient \( \partial J/\partial \alpha \) and is thus the same quantity as \( J \). With this notation and using the superscript \( T \) to mean matrix (or vector) transposition, the gradient can also be expressed as the transposed chain differentiation above:

\[
\overline{\alpha} = \left( \frac{\partial X}{\partial \alpha} \right)^T \left( \frac{\partial U}{\partial X} \right)^T \left( \frac{\partial J}{\partial U} \right)^T \overline{J}. \tag{3}
\]

This expression involves the following “adjoint” sensitivities:

\[
\overline{U} = \left( \frac{\partial J}{\partial U} \right)^T \overline{J}, \quad \overline{X} = \left( \frac{\partial U}{\partial X} \right)^T \overline{U}, \quad \overline{\alpha} = \left( \frac{\partial X}{\partial \alpha} \right)^T \overline{X}. \tag{4}
\]

where \( \overline{J} \) is 1. 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, schematically represented as

$$\alpha \rightarrow \chi \rightarrow U \rightarrow J,$$

the adjoint analysis proceeds backwards, from the functional to the design variables, that is

$$\alpha \leftarrow \chi \leftarrow U \leftarrow J.$$

In other words, while the forward propagation follows the “flow” in the original CFD calculation, from the design variable perturbation to the functional gradient, the backward propagation reverses this flow and propagates sensitivities from the objective functional to the design variables.

The advantage of using the adjoint method, i.e. the backward propagation, is that of a reduced cost. More precisely, assume the studied geometry is controlled by $n$ design variables and the post-processing of the flow solution outputs $m$ objective functionals. If the gradients of all $m$ functionals in the design space are sought, the forward mode computes them at a cost proportional to the dimension $n$ of the design space because the process propagates $n$ design sensitivities. At the same time, the backward mode computes the same gradients propagating $m$ functional sensitivities, hence the cost is proportional to $m$. Because the cost of evaluating any of the linear sensitivities is comparable to that for obtaining the corresponding adjoint sensitivity, the adjoint mode is advantageous for computing the gradients when $m$ is much smaller than $n$. This is precisely the case in optimization, where the gradients of a few objective functionals in a large dimensional design space are required.

An observation of practical importance is that, using the linear and adjoint sensitivities defined in (2) and (4), it is possible to evaluate the sensitivity of $J$ to any $\alpha$ in a number of ways. Thus, it can be easily verified that

$$J^T J = U^T U = \chi^T \chi = \chi^T \alpha,$$

in which the first dot product is the the result of full forward propagation and the last that of full reverse propagation. This concept is illustrated in Fig. 1, with the top horizontal line representing the forward mode sensitivity propagation and the bottom line the backward propagation. The vertical dashed lines show the stages at which the sensitivity vector products in (5) can be computed to obtain $J$. It is thus possible to proceed through a part of the forward process and combine the result of that with going through the other part of the backward process. This observation is particularly useful if a stage in the CFD calculation is a black-box application which cannot be directly linearized. For example, if the step $\alpha \rightarrow \chi$ involves a proprietary CAD system or mesh generator, the corresponding adjoint step $\alpha \leftarrow \chi$ cannot be achieved and the only option may be to approximate the forward mode linear sensitivity $\chi$ through a central finite difference using $\chi(\alpha \pm \Delta \alpha)$ in order to obtain the objective gradient as $\chi^T \chi$.

A partial sensitivity propagation in the forward mode does not undermine the efficiency argument for the adjoint method because the costs of evaluating the sensitivities in (2) or (4) are very different. For example, the cost of the step $\chi \rightarrow U$ is comparable to that of $\chi \leftarrow U$ but orders of magnitude higher than the cost of computing $\chi$ through finite differences or otherwise. Then, the forward mode evaluation of $\chi$ has a negligible cost in obtaining the gradient as $\chi^T \chi$, even when the number $n$ of design variables is large, so the benefit of employing the adjoint method remains.

**THE ADJOINT EQUATIONS**

While the previous section introduced the general ideas behind the adjoint approach, this section discusses some details of the adjoint solution process. This is important because computing the flow solution is the most expensive step of the generic CFD calculation, so the corresponding adjoint solution is the most expensive step in the reverse sensitivity propagation.

The vector of flow quantities $U$ is the solution of a set of nonlinear discrete steady-state flow equations of the form

$$N(U, \chi) = 0. \quad \text{(6)}$$

The vector of discrete nonlinear residuals $N$ depends on the nodal coordinates as well as on the flow quantities, so the flow solution $U$ is itself an implicit function of the coordinates $\chi$.

A typical turbomachinery application exploits the circumferential periodicity of the geometry and computes $U$ on a single blade-to-blade passage by using a periodicity boundary conditions. Other boundary conditions are viscous no-slip on the rigid walls (blade, hub and casing) as well as a characteristic-based treatment of inflow and outflow. In a weak numerical formulation, all these boundary conditions can be incorporated into the definition of the discrete residuals $N$ as flux equilibrium equations [11], such that the details can be ignored in the discussion.
of the adjoint equations. However, if a hard wall boundary condition is used (as in vertex centered finite volume codes), the form of Eqn. (6) is slightly more complicated and the condition has an effect on the exact definition of the discrete adjoint equations; for an in-depth discussion, see [8]. Nevertheless, this detail can be omitted for the purpose of explaining the application of AD.

The set of equations (6) is solved using an iterative process, commonly employing the popular explicit Runge–Kutta scheme [12, 13].

The sensitivity $\dot{U}$ of the flow changes in the design variables is obtained through linearizing the discrete nonlinear flow equations. Supposing the sensitivity $\dot{X}$ of the mesh coordinates to the design variables is known, Eqn. (6) is differentiated to obtain the linear system

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

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

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

The adjoint equations corresponding to (7) are found by re-expressing the gradient $J$ as a new dot product:

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

The vector of adjoint quantities $\overline{N}$ is thus defined as the solution to the adjoint system

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

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

In this formulation, Eqns. (7) and (8) naturally include the correct linear and adjoint weak boundary conditions, respectively. For an exact description of the linear and adjoint discrete boundary conditions and a detailed discussion of the effect of the hard boundary conditions at solid walls, see [11].

The adjoint system of equations (8) can be solved either directly using the same Runge–Kutta scheme as for the steady-flow solution or employing the adjoint version of this iterative scheme [8,14]. By construction, the latter approach has the property that the original Runge–Kutta scheme applied on the linear system (7) has the same iteration history as the adjoint scheme applied on the adjoint equations (8). The authors have found this to be very advantageous during the development of the adjoint code. Alternatively, one can use a specialized linear solver, like the Generalized Minimal Residual method [15].

**AUTOMATIC DIFFERENTIATION**

Any of the discrete sensitivities in the previous sections is the linearizion of the output of a computer program with respect to the input of that program, e.g. $\dot{U}$ is the linearizion of the CFD code which computes the flow solution $U$ with respect to the nodal coordinates $X$. Such a linearizion can be achieved by modifying the original computer program to perform, apart from the original operations, the linear or adjoint operations necessary to output the required sensitivity. Such a modified code 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 purpose of AD tools is to automate the process of obtaining the sensitivities from a given computer programs 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 [10] but a more practical short introduction in the context of optimization is due to Momhaddi [9].

It is useful to remark that there are two basic approaches to AD implementation: *operator overloading* and *source transformation*. Operator overloading (a feature familiar to programmers in C++) means the augmentation of the the mathematical operators and intrinsic functions in the original code such 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 [16].

Source transformation 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 [17]. Unlike operator overloading, the computer program generated by source transformation is compiled and executed independently of the original and leaves the programmer more flexibility to control the AD process in order to achieve computational efficiency in the adjoint code [18]. While an exhaustive mathematical description of how source transformation functions is given in [10], readers may find the shorter overview in [19] helpful.

The work reported here employed the AD software Tapeenade, which is being developed at INRIA [20, 21]. Currently, this tool applies source transformation to codes written in Fortran 77 only but work is in progress to extend the software to C

---

4 Copyright © 2007 by ASME
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.

In principle, AD software is designed to transform any computer program into new code that outputs the required sensitivities. In practice, the selective use of AD targeting specific code sequences, rather than “black-box” application to an entire code, leads to more efficient computer code, particularly in the reverse mode.

Using AD on CFD Codes

Although AD technology can, in principle, be used directly on codes of any size, some initial preparation is recommended for large codes. The aim of this preparation is to force AD to produce efficient adjoint code and is not restricted to a particular AD software or to a programming language. Without preparation, AD generated adjoint code is likely to be unacceptably costly to run, which is obviously more of a concern with a large code.

The minimal code preparation involves the replacement (or only re-arrangement) of those instructions in the original program that typically lead to inefficient reverse mode code through the AD process. Adjoint code performance can be substantially improved by such changes, e.g. the replacement of a \texttt{max} statement by an equivalent \texttt{if} control; a list of typical situations is given in [18]. Also, some discussion of how more complex situations can be treated using an algorithmic understanding of the code in order to produce efficient AD adjoint code is in [22].

In this work, the task was to use Tapenade to create adjoint code from the steady-state version of the Fortran 77 turbomachinery flow solver Hydra [23, 24]. Hydra is a vertex centered code that approximates the Reynolds-averaged Navier–Stokes equations on unstructured hybrid meshes, using an edge-based discretization [25]. The solution procedure solves the discrete nonlinear equations using Runge–Kutta time-marching accelerated by Jacobi preconditioning and multigrid, with dual-time stepping for unsteady flows.

The preparation of Hydra for optimal AD adjoint code generation consisted of two main 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 adjoint code, based on the existing framework of the original code.

The first step involved the re-structuring of the routines which compute the edge and boundary flux contributions to the discrete nonlinear residual $N$. This was done by isolating the calculations carried out at the level of edges or boundary nodes from the rest of the routines into new routines. This idea is sketched in Fig. 2, where the operations originally occurring within the loop over the mesh 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 were isolated from the rest of the code. The new routines contain the edge or node floating point calculations, which represent the bulk of the CFD solution computational load. Then, 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.

The main reason behind the first step was to maintain control over the parallel execution of the AD generated code. The parallel execution of the Hydra loops is achieved using instructions from the OPlus library [26], which interface the CFD code with MPI low-level instructions. The adjoint code loops require new parallel instructions for the new adjoint variables which are cannot be generated by AD and have to be hand-coded. By restructuring the code in this way, the control of the parallel execution is separated from the floating point calculations on which the AD acts.

At a higher level than the re-structuring, the second step adapts the framework of the Hydra code resulting at the first step to the adjoint code. The adaptation of the adjoint code framework can be illustrated in a simple way by Fig. 3, in which the structure of the nonlinear code routine flux is transferred to the adjoint routine flux_adj. This is an easy hand-coding effort which requires no more than the necessary changes to include the appropriate adjoint variables and the corresponding parallel instructions. The new adjoint flux routine flux_adj makes use the edge routine edge_flux_b, which is automatically obtained from the original edge_flux using AD.

Most of the effort involved in the second step is spent in coding adjoint routines as illustrated in Fig. 3. Additionally, programming the framework for the Hydra adjoint code also demanded attention to higher level algorithmic aspects of the adjoint method. Mainly these aspects were:

- the hard boundary conditions at the solid walls need to be
treated separately from the weak conditions which are handled directly through AD [11];

– 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 [25];

– the multigrid restriction and prolongation routines can be used with little modification for the adjoint solver but they exchange roles: the linear prolongation becomes the adjoint restriction and vice-versa [8];

– 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.

Performance Aspects

AD theoretical results show that the number of operations in the adjoint code is no larger than three times the number of operations in the nonlinear code [10]. Then, the ideal clock time for the execution of the adjoint operations has a corresponding upper limit. In practice, the adjoint wall clock time can be much larger than three times the run time for the nonlinear code [18]; this mainly happens because of memory access and parallel communication overheads. Also, hand-coded programs which make informed use of quantities common to the original sequence of operations and the reverse mode operations can always outperform AD generated adjoint code.

However, careful automatic generation of the adjoint code following the above guidelines should prevent unacceptable performance while retaining the comfort of using AD. Thus, the AD generated adjoint Hydra code is indeed times more expensive in terms of run time and about twice more expensive in terms of memory than the original nonlinear solver. In the practical use of the method, it is conventional to consider the cost of the nonlinear and adjoint runs as being equivalent.

Other Practical Aspects

By separating the general code framework from the calculations, as described above, control is retained in the adjoint code over all the algorithmic features while the AD targets only the low-level flux contribution routines. In the author’s experience, this is the optimal balance between programming for efficient code and ease of development.

The generation of the adjoint low-level routines is very simple and is managed automatically by a Makefile. The Makefile first invokes Tapenade to 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. In Fig. 3, this represents the transformation of the routine edge_flux into edge_flux_b. The Makefile then compiles the AD generated code into object file after which the AD code is no longer needed and the Makefile deletes it.

The only delicate part of using Tapenade is to correctly specify 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 are used again later and the “output” variables are increments added to pre-existing values.

The last is the the most common case in practice; an example is the edge flux routines, in which the input flux variables are re-used by subsequent routines and the output flux residuals are incremental contributions to mesh nodal values. For a quick start in using AD for CFD, very useful is the step-by-step guide exemplified on a 2D airfoil Euler code [27].

Validation

Although AD produces correct transformed code, it is good programming practice to design a suite of validation tests in order to verify that the AD generated code performs as expected. There are two main sources of programming errors: first, the status of the active variables may be incorrectly specified in the Makefile AD commands and second, the AD generated low-level routines may be mis-assembled within the adjoint hand-coded framework. These make testing particularly desirable in the case of a large code like Hydra.

Fundamentally, two levels of validating tests are recommended, one aimed at individual low-level routines or assemblies of routines and the other at the complete adjoint code. The first level is designed to identify possible errors in specifying the variable status in AD commands, while the second level is intended to validate the correct functioning of the adjoint code as a whole. Validation of adjoint sensitivities at both levels is carried out by comparison with sensitivities obtained from the nonlinear code. More thorough descriptions of the validation process can be found in [11] and [19].
It is worth mentioning here that in the case of low-level routines, sensitivities are estimated from the nonlinear code using a Taylor series expansion in the complex plane [8]. This method has in fact been used in the past to compute linear sensitivities of entire codes [28], is easy to implement and yields sensitivities accurate to within machine accuracy. Despite this, it is relatively expensive and is not an adjoint method as sensitivities can be propagated in the forward mode only. Nevertheless, this method constitutes an excellent debugging instrument and is warmly recommended. At this level, the agreement between adjoint gradients and sensitivities derived using the complex Taylor series is within machine accuracy.

The complex Taylor series method is suited only for low-level routine validation [19]. The validation of the entire adjoint code has to rely on sensitivities obtained from finite differences on nonlinear flow calculations. The adjoint Hydra code was validated on the NASA rotor testcase presented below, with the initial rotor geometry perturbed by twisting the blade at midheight by an angle $\Delta \theta$ about the leading edge, the twist gradually vanishing to zero towards hub and casing. This perturbation has an effect on the entire mesh, including the hub, casing, inflow and outflow boundaries, so the correct AD treatment of all the boundary conditions is tested. Six different values were chosen for the twist angle and the mass flow was output by the nonlinear Hydra solver on the perturbed meshes as well as on the initial geometry, Table 1. The massflow values 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. The finite difference sensitivity estimated from the values at $\Delta \theta = \pm 10^{-6}$ is $1.344457 \times 10^{-9}$ kg/s and that obtained from the adjoint calculation is $1.344462 \times 10^{-9}$ kg/s, so the agreement is to within around $4 \times 10^{-6}$ relative error.

<table>
<thead>
<tr>
<th>twist angle [deg]</th>
<th>massflow [kg/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.e-4</td>
<td>20.8493180070651</td>
</tr>
<tr>
<td>-1.e-5</td>
<td>20.8493181280627</td>
</tr>
<tr>
<td>-1.e-6</td>
<td>20.8493181401630</td>
</tr>
<tr>
<td>0</td>
<td>20.8493181415069</td>
</tr>
<tr>
<td>+1.e-6</td>
<td>20.8493181428519</td>
</tr>
<tr>
<td>+1.e-5</td>
<td>20.8493181549520</td>
</tr>
<tr>
<td>+1.e-4</td>
<td>20.8493182759513</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>number of blades</th>
<th>rotational speed</th>
<th>tip diameter (leading edge)</th>
<th>hub diameter (leading edge)</th>
<th>tip speed</th>
<th>tip clearance</th>
<th>pressure ratio</th>
<th>polytropic efficiency</th>
<th>mass flow rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>17,188.7 rpm</td>
<td>0.5074 m</td>
<td>0.3576 m</td>
<td>454.14 m/s</td>
<td>0.356 mm</td>
<td>2106</td>
<td>0.889</td>
<td>20.19 kg/s</td>
</tr>
</tbody>
</table>

Table 2. Main design characteristics of the NASA Rotor 37 testcase.

**APPLICATION**

The design problem chosen to illustrate the use of adjoint gradients is the performance optimization of the NASA Rotor 37. This is a highly loaded transonic compressor, designed and studied experimentally at the NASA Glenn Research Center [29]. It became one of the most popular rotor testcases and was used in a blind test sponsored by the ASME and IGTI for the 39th International Gas Turbine Conference (unpublished). It was also used by the AGARD Propulsion and Energetics Panel Working Group 26 as a testcase for examining the effects of mesh and turbulence model on solution accuracy [30]. The characteristics of this testcase are furnished in Table 2.

The optimization exercise was performed using Soft [31], an optimization environment with state-of-the-art libraries for local and global search. The geometry parametrization and meshing was done by Padram [32], a rapid multi-block meshing system which controls a rich design space, which ranges from blade lean and sweep to endwall profiling and fillet definition. The flow solutions were calculated by the turbomachinery flow solver Hydra.

The design space was defined by 28 parameters: there were 5 types of design perturbations (sweep, lean, skew, leading and trailing edge re-cambering) applied at 6 different locations along the blade span (hub, tip and 20%, 40%, 60% and 80% of height). Sweep and lean were kept on the reference value zero at the hub.

The task was to maximize the isentropic efficiency of the NASA Rotor 37 with a constraint on the stage pressure ratio, which was not allowed to vary by more than 2% of the initial value. The optimisation was carried out at a single working point on the compressor characteristic (97.91% of choke massflow, 100% engine speed) but the optimized configuration was assessed over the entire full speed characteristic.

The adjoint gradients computed by the Hydra adjoint code were used by a Sequential Quadratic Programming (SQP) algorithm, based on the NLPQL algorithm due to Schittkowski [33]. Optimization starts with a quadratic approximation of the Lagrangian function and a linearization of the constraints to formulate a quadratic sub-problem which is then solved. The search for an optimum continues by solving further quadratic sub-problems.
based on approximations of the functional Hessian matrix obtained from the available gradients using the BFGS formula.

The optimization history of efficiency is shown in Fig. 4 and the optimum was attained at the 3rd iteration. The iterations at which the gradients were required by NLPQL and computed using the adjoint method are depicted with square symbols. The search at the 3rd iteration was constrained, so the 4th iteration corresponds to a lower efficiency and is followed by a gradual return to the region of the optimum along the linearized constraint. At those iterations, depicted using circles, the gradients were not required. The gradients were computed again at the final iteration, where convergence was detected.

At those iteration where they are required, the adjoint method provides the NLPQL algorithm with gradients at an extra cost practically equal to that of a single nonlinear flow run. This cost does not depend on the dimension of design space, which is in contrast with finite difference evaluations whose cost scales linearly with the number of design variables.

Analyzing the radial variation of the circumferentially averaged efficiency immediately downstream of the blade for the initial and optimized blade, a decrease in the hub section efficiency was observed for the optimized geometry, Fig. 5. To alleviate this problem, a second optimization was performed only on the hub section, starting from the already optimized configuration. The drop in efficiency towards the hub was thus reduced in the final optimized blade.

Efficiency was raised with 1% at the working point, at the cost of a relative drop of 2% in the pressure ratio. Figure 6 plots the efficiency vs. massflow characteristic of both the initial rotor and the optimized geometry at full speed. There is a clear increase in efficiency over almost the entire massflow range, while the efficiency remains unchanged in the choke region.

Finally, Fig. 7 depicts the initial and optimized rotor blade configurations. The difference between the two geometries mainly consists of skew and lean, as is also apparent from the hub, midspan and tip sections depicted in Fig. 8. The main effect of this difference was a reduction of the flow separation on the suction side.

**CONCLUSIONS**

This paper presented the practical aspects of using AD on a large industrial CFD solver in order to assist the creation of an adjoint code to be used in design optimization. At the current stage of maturity, AD technology is best applied selectively on large codes, so that the efficiency of the resulting adjoint code
is maximized. Based on the experience of the authors with the Hydra turbomachinery solver, the paper discussed the important steps towards successfully using AD on large CFD codes.

The first step is a re-structuring of the nonlinear CFD solver prior to AD transformation, which involves the isolation of floating point operations at the level of mesh edges or nodes from the flow control (mainly do loops). This step separates the algorithmic structure of the code from the operations on which AD has an effect. This structure is adapted and hand-coded in the adjoint code, incorporating all the necessary algorithmic aspects: adjoint multigrid, adjoint Runge–Kutta iterations, etc. The adjoint structure controls the low-level routines that are AD generated and contain all the adjoint floating point operations. A balance is thus reached between hand-coding for efficient adjoint execution and the drive to make as much use as possible of the AD technology.

To illustrate the use of the adjoint gradients, a design optimization was performed on the NASA Rotor 37 testcase. The isentropic efficiency was raised by one percent using an SQP algorithm with a constraint on the stage pressure ratio value. The adjoint method provided the optimization process with inexpensive gradients.

ACKNOWLEDGMENT
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 Shaheen. We are grateful to Laurent Hascoët for making Tapenade available to us, and for being responsive to our queries.

REFERENCES


http://www-sop.inria.fr/tropics/.


