

# Using Automatic Differentiation for Adjoint CFD Code Development

M.B. Giles

D. Ghatta

M.C. Duta

*Oxford University Computing Laboratory, Parks Road, Oxford, U.K.*

有对应的代码已下载到我的电脑里。

This paper addresses the concerns of CFD code developers who are facing the task of creating a discrete adjoint CFD code for design optimisation. It discusses how the development of such a code can be greatly eased through the selective use of Automatic Differentiation, and how the software development can be subjected to a sequence of checks to ensure the correctness of the final software.

*Key words and phrases:* discrete adjoint, design optimisation, automatic differentiation

Oxford University Computing Laboratory  
Numerical Analysis Group  
Wolfson Building  
Parks Road  
Oxford, England OX1 3QD

November, 2005

# 1 Introduction

The use of adjoint methods for design optimisation has been a major research area in the last few years. Pironneau first introduced the idea of using an adjoint approach in fluid dynamic context [18] but the application to aeronautical design optimisation has been pioneered by Jameson [14, 15, 19].

Jameson uses the “continuous” adjoint approach in which the adjoint equation is formulated at the differential equation level, and it is then discretised. In contrast to this, Elliott and Peraire [5] and others [1, 9] follow a “discrete” adjoint approach in which one starts with nonlinear discrete equations, and then formulates the corresponding discrete adjoint equations.

There is no fundamental reason to prefer one approach over the other. Proponents of the discrete approach sometimes point to the fact that the continuous approach yields a design gradient (the gradient of the objective function with respect to the design variables) which is not quite consistent with the discrete objective function being optimised. However, it is as good an approximation to the true design gradient as the discrete gradient, and provided one uses an optimisation strategy (e.g. preconditioned steepest descent) which simply drives the continuous gradient to zero, then it will yield results as good as the discrete approach.

Our strong preference for the discrete approach is pragmatic in nature, and based on the following key points:

- There is a clear prescriptive process for constructing the discrete adjoint equations and boundary conditions; 规定的
- In most cases, if an adjoint iterative solution technique is used then it is guaranteed to give an iterative convergence rate equal to that of the original nonlinear code;
- Automatic Differentiation can be used to substantially ease the development of the adjoint CFD code.

The purpose of this paper is to explain these three points, with particular emphasis on the final point. One of the earliest applications which motivated the development of “reverse mode” AD (which can stand for either Algorithmic Differentiation [11] or Automatic Differentiation [12]) was in fluid dynamics, an adjoint version of an ocean circulation model being developed at MIT [6]. In the aeronautical context, perhaps the first application was by Mohammadi [16]. In simple, small applications, it is sometimes possible to use AD as a “black-box”, feeding in a nonlinear code and obtaining a corresponding linear perturbation (forward mode AD) or adjoint (reverse mode AD) code. However, in real applications with very large codes in which one wants to minimise the CPU and memory requirements, it is usually necessary to apply the AD very selectively. This is particularly the case in design optimisation in which one is using a fixed point iteration to solve the original nonlinear equations [7, 8].

## 2 Mathematical overview

### 2.1 Linear and adjoint sensitivity propagation

Suppose we have a design process in which a set of design parameters  $\alpha$ , leads to a computational grid with coordinates  $X$ , producing a discrete flow solution  $U$ , and hence a scalar output  $J$  which is to be optimised:

$$\alpha \longrightarrow X \longrightarrow U \longrightarrow J.$$

In order to use a gradient-based optimisation method, one wishes to compute the derivative of  $J$  with respect to  $\alpha$ . Adopting the notation used in the AD community, let  $\dot{\alpha}, \dot{X}, \dot{U}, \dot{J}$  denote the derivative with respect to one particular component of  $\alpha$  (or more generally in one particular direction in the design space).

If at each stage in the process the output is an explicit function of the input, then straightforward differentiation gives

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

and hence

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

Again following the notation used in the AD community the adjoint quantities  $\bar{\alpha}, \bar{X}, \bar{U}, \bar{J}$  denote the derivatives of  $J$  with respect to  $\alpha, X, U, J$ , respectively, with  $\bar{J} = 1$  by definition. Differentiating again, (and with a superscript  $T$  denoting a matrix or vector transpose) one obtains

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

and similarly

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

giving

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

Note that whereas the linear sensitivity analysis proceeds forwards through the process (forward mode in AD terminology)

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

the adjoint analysis proceeds backwards (reverse mode in AD terminology),

$$\bar{\alpha} \longleftarrow \bar{X} \longleftarrow \bar{U} \longleftarrow \bar{J}.$$

Given these definitions, the sensitivity of the output  $J$  to the inputs  $\alpha$  can be evaluated in a number of ways,

$$\dot{J} = \bar{U}^T \dot{U} = \bar{X}^T \dot{X} = \bar{\alpha}^T \dot{\alpha},$$

so it is possible to proceed forwards through part of the process and combine this with going backwards through the other part of the process. This is 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 through a central finite difference using  $X(\alpha \pm \Delta\alpha)$ .

The advantage of using the adjoint approach as far as possible is that its computational cost is independent of the number of design variables, whereas the cost of the forward linear sensitivity analysis increases linearly with the number of design variables.

## 2.2 Fixed point iteration

### 2.2.1 Nonlinear and linear equations

In CFD applications, the flow solution  $U$  is not an explicit function of the grid coordinates  $X$ , but instead is defined implicitly through the solution of a set of nonlinear discrete flow equations of the form

$$N(U, X) = 0. \quad (2.1)$$

To solve these equations, many CFD algorithms use iterative methods which can be written as

$$U^{n+1} = U^n - P(U^n, X) N(U^n, X), \quad (2.2)$$

where  $P$  is a non-singular square matrix which is a differentiable function of its arguments. If  $P$  were defined to be  $L^{-1}$  where

$$L = \frac{\partial N}{\partial U},$$

$$\left\{ \begin{array}{l} \frac{\partial N}{\partial U} \cdot \delta U = -N \\ U^{n+1} = U^n + \delta U \end{array} \right.$$

is the non-singular Jacobian matrix, this would be the Newton-Raphson method which converges quadratically. However, in the iterative methods used in CFD,  $P$  is a poor approximation to  $L^{-1}$  and therefore gives linear convergence asymptotically, with the final rate of convergence being given by the magnitude of the largest eigenvalue of the matrix  $I - P(U, X) L(U, X)$ , where  $I$  is the identity matrix.

Differentiating Equation (2.1) gives

$$L \dot{U} + \dot{N} = 0, \quad (2.3)$$

where  $\dot{N}$  is defined as

$$\dot{N} = \frac{\partial N}{\partial X} \dot{X},$$

with both derivatives being evaluated based on the implicitly-defined baseline solution  $U(X)$ .

Similarly, differentiating Equation (2.2) around a fully-converged baseline solution in which  $U^n = U$  gives

$$\dot{U}^{n+1} = \dot{U}^n - P \left( L \dot{U}^n + \dot{N} \right), \quad (2.4)$$

with  $P$  based on  $U(X)$ . This will converge to the solution of Equation (2.3) with exactly the same terminal rate of convergence as the nonlinear iteration.

To be more specific, the simple predictor/corrector time-marching used in the model application to be discussed later has the linearisation

$$\begin{aligned} \dot{U}^* &= \dot{U}^n - T \left( L \dot{U}^n + \dot{N} \right), \\ \dot{U}^{n+1} &= \dot{U}^n - T \left( L \dot{U}^* + \dot{N} \right), \end{aligned}$$

where  $T$  is a diagonal matrix containing the area/timestep values for each cell. This can be expressed in the form of Equation (2.4) with

$$P = T (I - LT).$$

### 2.2.2 Adjoint equations

Since

$$\dot{U} = -L^{-1}\dot{N},$$

the adjoint sensitivities satisfy the equation

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

which implies that

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

This equation can be solved iteratively using the adjoint iteration

$$\overline{N}^{n+1} = \overline{N}^n - P^T (L^T \overline{N}^n + \overline{U}). \quad (2.5)$$

Note the use of the transposed preconditioner  $P^T$ . In the case of the predictor/corrector time-marching,

$$P^T = (I - TL^T) T = T (I - L^T T).$$

Thus the adjoint iteration can be implemented using exactly the same predictor/corrector time-marching.

The adjoint iteration converges at exactly the same rate as the linear iteration, since  $I - P^T L^T$  has the same eigenvalues as  $I - PL$ . Furthermore, if the linear and adjoint iterations both start from zero initial conditions ( $\dot{U}^0 = \overline{N}^0 = 0$ ) then they give identical values for the overall objective function gradient after equal numbers of iterations since

$$(\overline{N}^n)^T \dot{N} = \overline{U}^T \dot{U}^n. \quad (2.6)$$

This equivalence is a consequence of the following two results which can be proved inductively:

$$\begin{aligned}\dot{U}^n &= - \sum_{m=0}^{n-1} (I - PL)^m P \dot{N} \\ \overline{N}^n &= - \sum_{m=0}^{n-1} (I - P^T L^T)^m P^T \overline{U}\end{aligned}$$

### 2.2.3 Other iterative solvers

Not all iterative schemes of the form given by Equations (2.2) and (2.4) are naturally self-adjoint, in the sense that the adjoint iteration is essentially the same as the linear iteration. Reference [7] constructs the adjoint iteration for multi-step Runge-Kutta methods with a partial update of viscous terms (a popular method introduced by Jameson) and also discusses the use of multigrid for which the adjoint restriction operator has to be the transpose of the original prolongation operator, and vice versa.

There are also some iterative solution methods which are not of the form given by Equations (2.2) and (2.4), for example the use of GMRES to solve Equations (2.1) and (2.3). In these cases, the iterative solution of the discrete adjoint equations can probably still be solved with the same final convergence rate, but the proof of this will depend on the fact that  $L$  and  $L^T$  have the same eigenvalues. If a preconditioner  $P$  is used to accelerate the convergence of the linear solver, then  $P^T$  should be used as the preconditioner for the adjoint solver, since again  $PL$  and  $P^T L^T$  have the same eigenvalues. However, the feature of equivalence between the linear and adjoint solutions after equal numbers of iterations is unlikely to exist for these methods.

## 2.3 More general outputs

A final comment is that most objective functions of interest depend on  $X$  as well as  $U$ . In the forward linear analysis, the only modification this introduces is to the final calculation,

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

while in the reverse adjoint calculation the equation for  $\overline{X}$  is

$$\overline{X} = \left( \frac{\partial N}{\partial X} \right)^T \overline{N} + \left( \frac{\partial J}{\partial X} \right)^T.$$

### 3 Automatic Differentiation

The previous section outlined the mathematics of the discrete adjoint approach, and tried to substantiate the first two of the key points presented in the Introduction, that there is a clear prescriptive process to constructing the discrete adjoint equations and solution procedure, and in most cases it is guaranteed to converge at exactly the same rate as the terminal convergence of the original nonlinear solver. This means that the adjoint solver benefits immediately from all of the research and hard work that has gone into the rapid solution of the nonlinear discrete flow equations.

This section now addresses the third point, how the process of creating the adjoint program can be simplified through the use of Automatic Differentiation. To do this, we begin by looking at the mathematics of adjoint sensitivity calculation at the lowest possible level to understand how automatic differentiation works.

Consider a computer program which starts with a number of input variables  $u_i, i = 1, \dots, I$  which can be represented collectively as an input vector  $\mathbf{u}^0$ . Each step in the execution of the computer program computes a new value as a function of two previous values; unitary functions such as  $\exp(x)$  can be viewed as a binary function with no dependence on the second parameter. Appending this new value to the vector of active variables, the  $n^{th}$  execution step can be expressed as

$$\mathbf{u}^n = \mathbf{f}^n(\mathbf{u}^{n-1}) \equiv \left( \frac{\mathbf{u}^{n-1}}{f_n(\mathbf{u}^{n-1})} \right), \quad (3.1)$$

where  $f_n$  is a scalar function of two of the elements of  $\mathbf{u}^{n-1}$ . The result of the complete  $N$  steps of the computer program can then be expressed as the composition of these individual functions to give

$$\mathbf{u}^N = \mathbf{f}^N \circ \mathbf{f}^{N-1} \circ \dots \circ \mathbf{f}^2 \circ \mathbf{f}^1(\mathbf{u}^0). \quad (3.2)$$

Defining  $\dot{\mathbf{u}}^n$  to be the derivative of the vector  $\mathbf{u}^n$  with respect to one particular element of  $\mathbf{u}^0$ , differentiating (3.1) gives

$$\dot{\mathbf{u}}^n = L^n \dot{\mathbf{u}}^{n-1}, \quad L^n = \left( \frac{I^{n-1}}{\partial f_n / \partial \mathbf{u}^{n-1}} \right), \quad (3.3)$$

with  $I^{n-1}$  being the identity matrix with dimension equal to the length of the vector  $\mathbf{u}^{n-1}$ . The derivative of (3.2) then gives

$$\dot{\mathbf{u}}^N = L^N L^{N-1} \dots L^2 L^1 \dot{\mathbf{u}}^0, \quad (3.4)$$

which gives the sensitivity of the entire output vector to the change in one particular element of the input vector. The elements of the initial vector  $\dot{\mathbf{u}}^0$  are all zero except for a unit value for the particular element of interest. If one is interested in the sensitivity to  $N_I$  different input elements, then (3.4) must be evaluated for each one, at a cost which is proportional to  $N_I$ .

The above description is of the forward mode of AD sensitivity calculation, which is intuitively quite natural. The reverse, or adjoint, mode is computationally much more efficient when one is interested in the sensitivity of a small number of output quantities with respect to a large number of input parameters. Defining the column vector  $\bar{\mathbf{u}}^n$  to be the derivative of a particular element of the output vector  $u_i^N$  with respect to the elements of  $\mathbf{u}^n$ , then through the chain rule of differentiation we obtain

$$\begin{aligned} (\bar{\mathbf{u}}^{n-1})^T &= \frac{\partial u_i^N}{\partial \mathbf{u}^{n-1}} = \frac{\partial u_i^N}{\partial \mathbf{u}^n} \frac{\partial \mathbf{u}^n}{\partial \mathbf{u}^{n-1}} = (\bar{\mathbf{u}}^n)^T L^n, \\ \implies \bar{\mathbf{u}}^{n-1} &= (L^n)^T \bar{\mathbf{u}}^n. \end{aligned} \quad (3.5)$$

Hence, the sensitivity of the particular output element to all of the elements of the input vector is given by

$$\bar{\mathbf{u}}^0 = (L^1)^T (L^2)^T \dots (L^{N-1})^T (L^N)^T \bar{\mathbf{u}}^N. \quad (3.6)$$

Note that the reverse mode calculation proceeds backwards from  $n=N$  to  $n=1$ . Therefore, it is necessary to first perform the original calculation forwards from  $n=1$  to  $n=N$ , storing all of the partial derivatives needed for  $L^n$ , before then doing the reverse mode calculation.

If one is interested in the sensitivity of  $N_O$  different output elements, then (3.6) must be evaluated for each one, at a cost which is proportional to  $N_O$ . Thus the reverse mode is computationally much more efficient than the forward mode when  $N_O \ll N_I$ .

Looking in more detail at what is involved in (3.3) and (3.5), suppose that the  $n^{th}$  step of the original program involves the computation

$$c = f(a, b).$$

The corresponding forward mode step will be

$$\dot{c} = \frac{\partial f}{\partial a} \dot{a} + \frac{\partial f}{\partial b} \dot{b}$$

at a computational cost which is no more than a factor 3 greater than the original nonlinear calculation. Looking at the structure of  $(L^n)^T$ , one finds that the corresponding reverse mode step consists of two calculations:

$$\begin{aligned} \bar{a} &= \bar{a} + \frac{\partial f}{\partial a} \bar{c} \\ \bar{b} &= \bar{b} + \frac{\partial f}{\partial b} \bar{c}. \end{aligned}$$

At worst, this has a cost which is a factor 4 greater than the original nonlinear calculation.

The above description outlines a clear algorithmic approach to the reverse mode calculation of sensitivity information. However, the programming implementation can be tedious and error-prone. Fortunately, tools have been developed to automate this process, either through operator overloading involving a process known as ‘‘taping’’ which records all of the partial derivatives in the nonlinear calculation then performs



the reverse mode calculations [12], or through source code transformation which takes as an input the original program and generates a new program to perform the necessary calculations [6]. Further information about AD tools and publications is available from the AD community website [www.autodiff.org](http://www.autodiff.org) which includes links to all of the major groups working in this field.

In the present work, we are using Tapenade, developed by Hascoët and Pascual at INRIA [4, 13]. The software is written in Java, and it applies source transformation to codes written in FORTRAN77; work is in progress to extend the software to C and C++ but there are technical difficulties because of their use of pointers.

## 4 Example application

The example application is a 2D inviscid airfoil code using an unstructured grid. The full source code is available [10], including the files generated by Tapenade.

The nonlinear code uses a cell-centred discretisation together with a simple predictor/corrector timemarching, using local timesteps which are also involved in the definition of a simple first order accurate numerical smoothing. The schematic of the nonlinear flow code is given in Figure 1. There are four key nonlinear routines which are called from within loops over cell or faces:

- **TIME\_CELL:**  
computes the local area/timestep for a single cell
- **FLUX\_FACE:**  
computes the flux through a single regular face
- **FLUX\_WALL:**  
computes the flux for a single airfoil wall face
- **LIFT\_WALL:**  
computes the lift contribution from a single airfoil wall face

Figure 2 shows the schematic of the linear sensitivity flow code. The part of the code before the main time-marching loop computes the quantity  $\dot{N}$ ; since this remains constant throughout the time-marching it is much more efficient to compute it just once rather than re-evaluate it at each iteration. The time-marching loop then iterates to compute  $\dot{U}$ , and the final section then evaluates  $\dot{J}$ . The seven subroutines called by the linear flow code are all linearisations of the four nonlinear routines, and are generated automatically by Tapenade.

```

define grid and initialise flow field

begin predictor/corrector time-marching loop
  loop over cells to calculate timestep
    call TIME_CELL
  loop over regular faces to calculate flux
    call FLUX_FACE
  loop over airfoil faces to calculate flux
    call FLUX_WALL
  loop over cells to update solution
end time-marching loop

calculate lift
  loop over boundary faces
    call LIFT_WALL

```

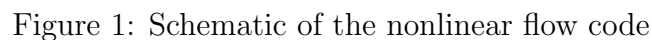


Figure 1: Schematic of the nonlinear flow code

```

define grid and initialise flow field
define grid perturbation

loop over cells -- perturbed timestep
  call TIME_CELL_DX
loop over regular faces -- perturbed flux
  call FLUX_FACE_DX
loop over airfoil faces -- perturbed flux
  call FLUX_WALL_DX

begin predictor/corrector time-marching loop
  loop over cells to calculate timestep
    call TIME_CELL_D
  loop over regular faces to calculate flux
    call FLUX_FACE_D
  loop over airfoil faces to calculate flux
    call FLUX_WALL_D
  loop over cells to update solution
end time-marching loop

calculate lift
  loop over boundary faces -- perturbed lift
    call LIFT_WALL_DX

```

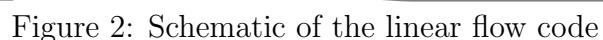


Figure 2: Schematic of the linear flow code

```

FLUX_WALL(x1,x2,q,res)

FLUX_WALL_D(x1,x2,q,qd,res,resd)

FLUX_WALL_DX(x1,x1d,x2,x2d,q,qd,res,resd)

FLUX_WALL_B(x1,x2,q,qb,res,resb)

FLUX_WALL_BX(x1,x1b,x2,x2b,q,qb,res,resb)

```

Figure 3: The arguments of the nonlinear subroutine FLUX\_WALL and its Tapenade-generated derivatives

```

define grid and initialise flow field

calculate adjoint lift sensitivity
  loop over boundary faces
    call LIFT_WALL_BX

begin predictor/corrector time-marching loop
  loop over airfoil faces -- adjoint flux
    call FLUX_WALL_B
  loop over regular faces -- adjoint flux
    call FLUX_FACE_B
  loop over cells -- adjoint timestep calc
    call TIME_CELL_B
  loop over cells to update solution
end time-marching loop

loop over airfoil faces -- adjoint flux
  call FLUX_WALL_BX
loop over regular faces -- adjoint flux
  call FLUX_FACE_BX
loop over cells -- adjoint timestep calc
  call TIME_CELL_BX

loop over nodes to evaluate lift sensitivity

```

Figure 4: Schematic of the adjoint flow code

As an example, Figure 3 shows the arguments for the nonlinear routine `FLUX_WALL`; the inputs are the coordinates of the two nodes at either end of the wall face, and the flow variables in the cell next to the face, and the output is an increment to the flux residual for that same cell. The routines `FLUX_WALL_D` and `FLUX_WALL_DX` are the two forward mode linearisations of `FLUX_WALL`. In the case of `FLUX_WALL_D`, the extra input is `qd` which represents  $\dot{U}$  in the mathematical formulation, and the extra output is `resd` which is the consequential perturbation to `res`. The suffix “D” in the function name and the suffix “d” in the variable names both stand for “dot”. `FLUX_WALL_DX` is similar but also includes perturbations to each of the coordinates. Which variables are considered “active”, and which are treated as being fixed, is controlled by the command line arguments supplied to Tapenade when it creates the new routines.

The forward mode linearisation is relatively natural, but the reverse mode adjoint code in Figure 4 is much less intuitive. The first thing to note is that the order in which the different routines is called is the opposite to the linear code. The first section has the calls necessary to compute  $\bar{U}$ , the time-marching loop iterates to find the value of  $\bar{U}$ , and then the final section computes  $\bar{X}$  and then evaluates the product  $\dot{J} = \bar{X}^T \dot{X}$ .

Within the time-marching loop, the reversal of the order in which the flux and timestep routines are called can be explained by noting that  $L$  can be viewed as the product of two matrices  $FA$ , where  $A$  is the effect of the timestep calculation which determines a timestep perturbation in addition to carrying over the flow perturbations, and then  $F$  is the effect of the linearised flux calculation. The adjoint operator is then  $L^T = (FA)^T = A^T F^T$ . Thus the fact that the transpose of a product of matrices is equal to the product of the transposed matrices in the opposite order, leads to the reversal in the order in which the adjoint routines are called. A similar behaviour is seen in adjoint viscous codes in which a flow gradient is first computed in one loop over edges, and then the viscous flux is computed in a second loop over edges [9].

Looking now in detail at the adjoint versions of `FLUX_WALL` in Figure 3, these too operate in reverse. While the linear version `FLUX_WALL_D` computes

$$\text{res} = \text{res} + K \dot{q},$$

for some matrix  $K$ , the adjoint version `FLUX_WALL_B` computes

$$\bar{q} = \bar{q} + K^T \bar{\text{res}}.$$

The suffix “B” in the function name and the suffix “b” in the variable names `qb` and `resb` all stand for “bar”. Thus `resb` is an unchanged input to `FLUX_WALL_B`, and `qb` is incremented appropriately by the routine. `FLUX_WALL_BX` is the version which includes variations in the coordinates; this is used to evaluate  $(\partial N / \partial X)^T \bar{N}$  as part of the calculation of  $\bar{X}$ .

The generation of the linear and adjoint versions of the four nonlinear routines is handled automatically by the Unix Makefile used to create the executables. For example, when the object file `flux_wall_bx.o` is needed for the adjoint executable `air_adj`, the Makefile runs Tapenade with an appropriate set of command line arguments, compiles the file `flux_wall_bx.f` which it generates, and then deletes the FORTRAN source file;

in practice it is rarely helpful to look at the Tapenade-generated source except to better understand how Tapenade works.

The trickiest part of using Tapenade is correctly specifying the status of active variables:

- input only – it is never used again after the routine is finished, so its output value is irrelevant;
- output only – it is assigned a value within the routine, so its input value is irrelevant;
- input and output – in practice the most common case, since “input” variables such as the flow variables will be used again later, and “output” variables such as the flux residual are in fact increments added to pre-existing values.

Figure 5 shows the part of the Makefile which uses Tapenade to generate the four linear and adjoint routines which are derived from the routine `FLUX_WALL` within the file `routines.F`. In each case, there is an initial step which uses the C preprocessor to produce pure FORTRAN code. This is processed by Tapenade to produce the desired output code, which is then compiled, and finally all unwanted intermediate files are deleted. The Tapenade flags are fairly obvious; `-forward` and `-reverse` specify whether to generate linear or adjoint code, `-vars` and `-outvars` specify the input and output active variables, and `-difffuncname` gives the function suffix to be used in place of the default `"_d"` or `"_b"`, depending on the AD mode. For more information, see the documentation [13].

```

flux_wall_d.o: routines.F
    ${GCC} -E -C -P routines.F > routines.f;
    ${TPN} -forward \
        -head      flux_wall \
        -output     flux_wall \
        -vars       "q res" \
        -outvars    "q res" \
        routines.f;
    ${FC} ${FFLAGS} -c flux_wall_d.f;
    /bin/rm routines.f flux_wall_d.f *.msg

flux_wall_dx.o: routines.F
    ${GCC} -E -C -P routines.F > routines.f;
    ${TPN} -forward \
        -head      flux_wall \
        -output     flux_wall \
        -vars       "x1 x2 q res" \
        -outvars    "x1 x2 q res" \
        -difffuncname "_dx" \
        routines.f;
    ${FC} ${FFLAGS} -c flux_wall_dx.f;
    /bin/rm routines.f flux_wall_dx.f *.msg

flux_wall_b.o: routines.F
    ${GCC} -E -C -P routines.F > routines.f;
    ${TPN} -backward \
        -head      flux_wall \
        -output     flux_wall \
        -vars       "q res" \
        -outvars    "q res" \
        routines.f;
    ${FC} ${FFLAGS} -c flux_wall_b.f;
    /bin/rm routines.f flux_wall_b.f *.msg

flux_wall_bx.o: routines.F
    ${GCC} -E -C -P routines.F > routines.f;
    ${TPN} -backward \
        -head      flux_wall \
        -output     flux_wall \
        -vars       "x1 x2 q res" \
        -outvars    "x1 x2 q res" \
        -difffuncname "_bx" \
        routines.f;
    ${FC} ${FFLAGS} -c flux_wall_bx.f;
    /bin/rm routines.f flux_wall_bx.f *.msg

```

Figure 5: Part of Makefile with Tapenade instructions for generating linear and adjoint versions of FLUX\_WALL; GCC, TPN, FC and FFLAGS correspond to the Gnu C compiler, Tapenade, the FORTRAN compiler and its compilation flags. respectively.

## 5 Validation checks

Although we have complete confidence in the correctness of the linear and adjoint code produced by Tapenade, it is nevertheless good programming practice to implement validation checks to ensure the software is performing correctly; in the past, this has helped to identify cases in which we have incorrectly specified the status of active variables when using Tapenade. This is particularly important in much larger applications than the model one considered in this paper.

### 5.1 Individual routines

One set of checks can be performed at the level of the individual routines generated by Tapenade. In concept, we have

- a nonlinear routine which computes a vector output  $f(u)$  given a vector input  $u$ ;
- a linear routine which computes  $\left(\frac{\partial f}{\partial u}\right) \dot{u}$ ;
- and an adjoint routine which computes  $\left(\frac{\partial f}{\partial u}\right)^T \bar{f}$ .

By calling the linear and adjoint routines for a sequence of different unit vectors  $\dot{u}$  and  $\bar{f}$ , each one zero except for a unit value for one element, it is possible to construct the matrix  $\partial f / \partial u$  which each is effectively using, and check that these are in agreement.

Checking for consistency against the original nonlinear code is possible by using the complex Taylor series expansion method [20, 2]. If  $f(u)$  is assumed to be a complex analytic function, then a Taylor series expansion gives

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

In this equation, the notation  $\mathcal{I}\{\dots\}$  denotes the imaginary part of a complex quantity. The convergence to the limiting value is second order in  $\epsilon$  so numerical evaluation with  $\epsilon < 10^{-8}$  yields double precision accuracy. In practice, we use  $\epsilon = 10^{-20}$ . Unlike the usual finite difference approximation of a linear sensitivity, there is no cancellation effect from the subtraction of two quantities of similar magnitude, and therefore no unacceptable loss of accuracy due to machine rounding error. Applying this technique to a FORTRAN code requires little more than replacing all `REAL*8` declarations by `COMPLEX*16`, and defining appropriate complex analytic versions of the intrinsic functions `min`, `max`, `abs`.

The source code [10] includes a test program `testlinadj` which performs all of these checks for all four of the nonlinear routines and their associated linear and adjoint derivatives.

## 5.2 Complete codes

A second set of checks can be applied to the complete codes,

- a nonlinear code which computes  $J(\alpha)$ ;
- a linear code which computes  $\dot{J}$ ;
- and an adjoint code which also computes  $\dot{J}$ .

The first thing is to check that the linear and adjoint codes produce the same value for  $\dot{J}$ . Not only should the values agree once both codes have fully converged, they should also agree after performing the same number of iterations if they use an iterative solution method of the type described earlier in the mathematical overview. This is very helpful when debugging very large codes which may run for a long time; checking for full agreement down to machine accuracy after 10 iterations takes much less CPU time than checking after 10,000 iterations.

It is usually impractical to apply the complex Taylor series method to the entire nonlinear code because it requires too much intervention in the code, so instead the final test is to check the linear sensitivity  $\dot{J}$  against the finite difference approximation

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

Typical results from such a check are shown in Figure 6. For extremely small values of  $\Delta\alpha$ , the dominant error is due to machine accuracy in computing  $J(\alpha)$  and has a magnitude which is  $O(\epsilon/\Delta\alpha)$  where  $\epsilon$  here represents the level of machine accuracy. For larger values of  $\Delta\alpha$ , the dominant error is  $O(\Delta\alpha^2)$  due to the second order accuracy of central finite differences.

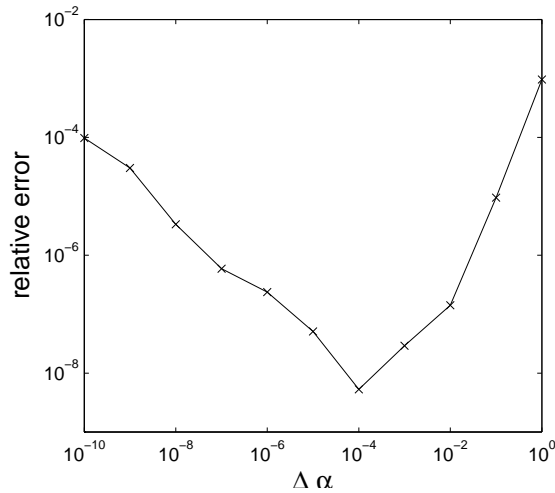


Figure 6: Relative error in finite difference sensitivity as a function of the step size  $\Delta\alpha$ .



## 6 Conclusions

Although the example application in this paper is very simple, the same approach is also being used for a set of turbomachinery codes called HYDRA. HYDRA includes a nonlinear analysis code [17], a linear code for the analysis of flutter and unsteady forced response [3], and an adjoint code for design optimisation [9]. The use of Tapenade is essential in keeping the linear and adjoint codes consistent with the latest changes to the nonlinear code, and Tapenade handles without any difficulty the complicated nonlinearities and conditional branching in the turbulence modelling and the characteristic-based numerical smoothing. As in the model application, an extensive set of validation checks is employed to verify the correctness of the linear and adjoint routines and codes.

It is our hope that the mathematical overview and software development process outlined in this paper will help and encourage those who are interested in developing industrial scale adjoint codes for design optimisation. It is strongly recommended that those who are interested in this should download the source code and Makefile [10] and obtain a copy of Tapenade [13] in order to try out the codes and run through the whole development process and validation checks.

## 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 very grateful to Laurent Hascoët for making Tapenade available to us, and for being so responsive to our queries.

## References

- [1] W.K. Anderson and D.L. Bonhaus. Airfoil design on unstructured grids for turbulent flows. *AIAA J.*, 37(2):185–191, 1999.
- [2] 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.
- [3] M.S. Campobasso and M.B. Giles. Effect of flow instabilities on the linear analysis of turbomachinery aeroelasticity. *AIAA J. Propulsion and Power*, 19(2), 2003.
- [4] F. Courty, A. Dervieux, B. Koobus, and L. Hascoet. Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. *Opt. Meth. and Software*, 18(5):615–627, 2003.
- [5] J. Elliott and J. Peraire. Practical 3D aerodynamic design and optimization using unstructured meshes. *AIAA J.*, 35(9):1479–1485, 1997.
- [6] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Trans. Math. Software*, 24(4):437–474, 1998.

- [7] M.B. Giles. On the use of Runge-Kutta time-marching and multigrid for the solution of steady adjoint equations. Technical Report NA00/10, Oxford University Computing Laboratory, 2000.
- [8] 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*, pages 145–152. Springer-Verlag, 2001.
- [9] M.B. Giles, M.C. Duta, J.-D. Müller, and N.A. Pierce. Algorithm developments for discrete adjoint methods. *AIAA J.*, 42(2), 2003.
- [10] M.B. Giles and D. Ghate. Source code for airfoil testcase for forward and reverse mode automatic differentiation using Tapenade, 2005. <http://www.comlab.ox.ac.uk/mike.giles/airfoil/>.
- [11] A. Griewank. *Evaluating derivatives : principles and techniques of algorithmic differentiation*. SIAM, 2000.
- [12] 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.
- [13] L. Hascoët and V. Pascual. Tapenade 2.1 user’s guide. Technical Report 0300, INRIA, 2004 <http://www-sop.inria.fr/tropics/>.
- [14] A. Jameson. Aerodynamic design via control theory. *J. Sci. Comput.*, 3:233–260, 1988.
- [15] A. Jameson, N. Pierce, and L. Martinelli. Optimum aerodynamic design using the Navier-Stokes equations. *J. Theor. Comp. Fluid Mech.*, 10:213–237, 1998.
- [16] 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.
- [17] P. Moinier, J.-D. Müller, and M.B. Giles. Edge-based multigrid and preconditioning for hybrid grids. *AIAA J.*, 40(10):1954–1960, 2002.
- [18] O. Pironneau. On optimum design in fluid mechanics. *J. Fluid Mech.*, 64:97–110, 1974.
- [19] J. Reuther, A. Jameson, J.J. Alonso, M.J. Remlinger, and D. Saunders. Constrained multipoint aerodynamic shape optimisation using an adjoint formulation and parallel computers, parts 1 and 2. *J. Aircraft*, 36(1):51–74, 1999.
- [20] W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Rev.*, 10(1):110–112, 1998.