AIAA JOURNAL
Vol. 46, No. 4, April 2008

ADjoint: An Approach for the Rapid Development of Discrete Adjoint Solvers

Charles A. Mader* and Joaquim R. R. A. Martins†
University of Toronto, Toronto, Ontario M3H 5T6, Canada
and
Juan J. Alonso‡ and Edwin van der Weide§
Stanford University, Stanford, California 94305

DOI: 10.2514/1.29123

An automatic differentiation tool is used to develop the adjoint code for a three-dimensional computational fluid dynamics solver. Rather than using automatic differentiation to differentiate the entire source code of the computational fluid dynamics solver, we have applied it selectively to produce code that computes the flux Jacobian matrix and the other partial derivatives that are necessary to compute total derivatives using an adjoint method. The resulting linear discrete adjoint system is then solved using the portable, extensible toolkit for scientific computation. This selective application of automatic differentiation is the central idea behind the automatic differentiation adjoint (ADjoint) approach. This approach has the advantage that it is applicable to arbitrary sets of governing equations and cost functions, and that it is exactly consistent with the gradients that would be computed by exact numerical differentiation of the original solver. Furthermore, the approach is largely automatic, thus avoiding the lengthy development times usually required to develop adjoint solvers for partial differential equations. These significant advantages come at the cost of increased memory requirements for the adjoint solver. Derivatives of drag and lift coefficients are validated, and the low computational cost and ease of implementation of the method are shown.

Introduction

ADJOINT methods for sensitivity analysis involving partial differential equations (PDEs) have been known and used for over three decades. They were first applied to solve optimal control problems and thereafter used to perform sensitivity analysis of linear structural finite element models. The first application to fluid dynamics is due to Pironneau [1]. The method was then extended by Jameson to perform airfoil shape optimization [2], and since then it has been used to design laminar flow airfoils [3], and to optimize airfoils suitable for multipoint operation [4].

The adjoint method has been extended to three-dimensional problems, leading to applications such as the aerodynamic shape optimization of complete aircraft configurations [5,6], and shape optimization considering both aerodynamics and structures [7,8]. The adjoint method has since been generalized for multidisciplinary systems [9].

The usefulness of the adjoint method lies in the fact that it is an extremely efficient approach to compute the derivative of one function of interest with respect to many parameters. When using gradient-based optimization algorithms, the efficiency and accuracy of the derivative computations has a significant effect on the overall performance of the optimization. Therefore, efficient and accurate sensitivity analysis is of paramount importance.

Given the power of adjoint methods, it seems peculiar that their use in aerodynamic shape optimization has not become more widespread in the last two decades. Although adjoint methods have already found their way into commercial structural analysis packages, they have yet to proceed beyond research computational fluid dynamics (CFD) solvers. One of the main obstacles is the complexity involved in the development and implementation of adjoint methods for nonlinear PDEs.

The panacea for addressing this problem might just be *automatic differentiation* [10]. This approach relies on a tool that, given the original solver, creates code capable of computing sensitivities. There are two different modes of operation for automatic differentiation: the forward and the reverse modes.

The forward mode propagates the required sensitivity at the same time as the solution is being computed. This is analogous to the finite difference method, but without step-size sensitivity problems.

To use the reverse mode, the solver has to be run to convergence first, with intermediate variable values stored for every iteration. These intermediate variables are then used by a reverse version of the code to find the sensitivities. The reverse mode is analogous to the adjoint method and is also efficient when computing the sensitivity of a function with respect to many parameters. However, the memory requirements of the reverse mode can be prohibitive in the case of iterative solvers, such as those used in CFD, because they require a large number of iterations to achieve convergence, and intermediate results may need to be stored. Although there has been some progress toward minimizing the memory requirements of iterative solvers [11], the fact remains that given typical parallel computing resources, it is still very difficult to apply the reverse mode to large-scale problems. The reverse mode of automatic differentiation has been applied to iterative PDE solvers by a few researchers with some success [12–14]. The main problem in each of these applications was the prohibitive memory requirement for three-dimensional domains.

Our main objective is to make the development of discrete adjoint solvers a routine and quick task that only requires the use of preexisting code to compute the equation residuals (including boundary conditions) and the cost functions [15,16]. To achieve this, we propose the automatic differentiation adjoint (ADjoint) approach, in which we use automatic differentiation to compute only certain terms of the discrete adjoint equations. These terms can then be used, together with standard techniques for the iterative solution of large linear systems—such as the preconditioned generalized minimum residual method (GMRES)—to perform sensitivity analysis. The major advantages of this method are as follows:

Presented as Paper 7121 at the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Portsmouth, Virginia, 6–8 September 2006; received 4 December 2006; accepted for publication 18 October 2007.

*Ph.D. Candidate. Student Member AIAA.
†Assistant Professor. Senior Member AIAA.
‡Associate Professor. AIAA Member
§Engineering Research Associate. Member AIAA.

1) Largely automatic: Given the solver source code, it creates the necessary code to compute all the terms in the discrete adjoint formulation.

2) Exactly consistent: Because the process of automatic differentiation allows us to treat arbitrarily complex expressions for the computation of the residuals, boundary conditions, and cost functions exactly, we are able to compute derivatives that are perfectly consistent with those that would be obtained with an exact numerical differentiation of the original solver. In other words, typical approximations made in the development of adjoint solvers (such as neglecting contributions from the variations resulting from turbulence models, spectral radii, artificial dissipation and upwind formulations) are not made here. The various effects of these approximations are well documented [17–19].

3) Generic: It can be quickly applied to new formulations of the governing equations or even to completely new governing equations (such as the governing equations for magnetohydrodynamics [20]).

Although the ADjoint approach does not constitute a fully automatic way of obtaining sensitivities, as is the case for pure automatic differentiation, it is much faster in terms of execution time and drastically reduces the memory requirements when compared to pure reverse-mode automatic differentiation. Further, when compared to an analytic adjoint method, the proposed approach requires a much shorter implementation time and can be used to develop the discrete adjoint of an arbitrary solver with a much reduced probability of programming errors. An approach similar to the ADjoint has been proposed by Nielsen and Kleb [21]. In this approach, the complex-step derivative approximation [22] is used to compute the partial derivatives instead of automatic differentiation.

In the next section we review the background theory that is relevant to this work, namely, semi-analytic sensitivity analysis methods and automatic differentiation. We then discuss how the ADjoint approach was implemented in this application. In the results, we establish the precision of the desired derivatives and analyze the performance of the ADjoint method.

Background

Semi-Analytic Sensitivity Analysis

One of the most popular sensitivity analysis methods is finite differencing. The widespread use of this method is due to its straightforward implementation. The disadvantages are that its computational cost is high, and its accuracy is low and hard to predict. For gradient-based optimization, both of these characteristics have a negative impact on the performance of the optimization.

Semi-analytic methods remedy both of these issues: They are capable of yielding derivatives with the same precision as the quantity that is being differentiated, and in the case of the adjoint method, they are able to achieve a computational cost that is essentially independent of the number of design variables.

Our intent is to compute the sensitivity of a function (or vector of functions) with respect to a large number of design variables. Such functions depend not only on the design variables, but also on the state of the system that results from the solution of a governing equation, which may be a PDE. Thus we can write the vector-valued function to be differentiated as

$$I = I(x, w) \tag{1}$$

where x represents the vector of design variables and w is the state variable vector.

For a given vector x, the solution of the governing equations of the system yields a vector w, thus establishing the dependence of the state of the system on the design variables. We denote these governing equations by

$$\mathcal{R}[x, w(x)] = 0 \tag{2}$$

As a first step toward obtaining the derivatives that we ultimately want to compute, we use the chain rule to write the total derivative of the vector-valued function I as

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \frac{\partial I}{\partial w}\frac{\mathrm{d}w}{\mathrm{d}x} \Leftrightarrow \frac{\mathrm{d}I}{\mathrm{d}x} = D + GB \tag{3}$$

where the sizes of the sensitivity matrices are

$$D = \frac{\partial I}{\partial x} \qquad (N_I \times N_x) \tag{4}$$

$$G = \frac{\partial I}{\partial w} \qquad (N_I \times N_w) \tag{5}$$

$$B = \frac{\mathrm{d}w}{\mathrm{d}x} \qquad (N_w \times N_x) \tag{6}$$

It is important to distinguish the total and partial derivatives in these equations. The partial derivatives can be directly evaluated by varying the denominator and reevaluating the function in the numerator with everything else remaining constant. The total derivatives, however, require the solution of the governing equations. Thus, all the terms in the total sensitivity equation (3) can be computed with relatively little effort except for $\mathrm{d}w/\mathrm{d}x$.

Because the governing equations must always be satisfied, the total derivative of the residuals (2) with respect to any design variable must also be zero, that is,

$$\frac{\mathrm{d}\mathcal{R}}{\mathrm{d}x} = \frac{\partial \mathcal{R}}{\partial x} + \frac{\partial \mathcal{R}}{\partial w}\frac{\mathrm{d}w}{\mathrm{d}x} = 0 \tag{7}$$

This expression provides the means for computing the total derivative of the state variables with respect to the design variables. To this end, we rewrite this equation as

$$\frac{\partial \mathcal{R}}{\partial w}\frac{\mathrm{d}w}{\mathrm{d}x} = -\frac{\partial \mathcal{R}}{\partial x} \Leftrightarrow AB = C \tag{8}$$

where we have defined the following sensitivity matrices:

$$A = \frac{\partial \mathcal{R}}{\partial w} \qquad (N_w \times N_w) \tag{9}$$

$$C = -\frac{\partial \mathcal{R}}{\partial x} \qquad (N_w \times N_x) \tag{10}$$

Thus the sensitivity analysis problem given by Eqs. (3) and (8) can be written as

$$\frac{\mathrm{d}I}{\mathrm{d}x} = D + GB, \quad \text{such that } AB = C \tag{11}$$

The final derivative can also be obtained by solving the dual of this problem [23], which is derived as follows. If we substitute the solution of the linear system $AB = C$ into the total sensitivity equation (3) we obtain

$$\frac{\mathrm{d}I}{\mathrm{d}x} = D + GA^{-1}C \tag{12}$$

Defining $H = (GA^{-1})^T$, which is of size $(N_w \times N_I)$, we can write the problem as

$$\frac{\mathrm{d}I}{\mathrm{d}x} = D + H^T C, \quad \text{such that } A^T H = G^T \tag{13}$$

The most computationally intensive step in both of these problems is the solution of the respective linear systems. In the case of problem (11)—the *direct method*—we have to solve a linear system of N_w equations N_x times. For the dual problem (13)—the *adjoint method*—we solve a linear system of the same size N_I times. Thus, the choice of which of these methods to use depends largely on how the number of design variables N_x compares to the number of functions of interest N_I.

When it comes to implementation, there are two ways of obtaining the discrete adjoint equations (13) for a given system of PDEs. The *continuous adjoint approach* forms a continuous adjoint problem from the governing PDEs and then discretizes this problem to solve it numerically. The *discrete adjoint approach* forms an adjoint from the discretized PDEs. Each of these approaches results in a different system of linear equations, but in theory they converge to the same result as the mesh is refined [23–26].

The discrete approach has the advantage that the sensitivities are consistent with those produced by the discretized solver. Furthermore, it is easier to obtain the appropriate boundary conditions for the adjoint solver in a discrete fashion. It has also been shown that the discrete adjoint formulation has the same asymptotic convergence rate as the original code [27]. In this work, we adopt the discrete approach. Although this approach usually requires more memory than the continuous adjoint, it is our opinion that the advantages we just mentioned outweigh this disadvantage.

CFD Adjoint Equations

We now derive the adjoint equations for the particular case of the flow solver used in this work. The governing PDEs for the three-dimensional Euler equations are

$$\frac{\partial w}{\partial t} + \frac{\partial f_i}{\partial x_i} = 0 \tag{14}$$

where x_i are the coordinates in the ith direction, and the state and the fluxes for each cell are

$$w = \begin{bmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho u_3 \\ \rho E \end{bmatrix}, \qquad f_i = \begin{bmatrix} \rho u_i \\ \rho u_i u_1 + p\delta_{i1} \\ \rho u_i u_2 + p\delta_{i2} \\ \rho u_i u_3 + p\delta_{i3} \\ \rho u_i H \end{bmatrix} \tag{15}$$

The derivation presented here is for the Euler equations. The ADjoint approach assumes the existence of code that computes the residual of the governing equations, but does not make any assumptions about the content of that code. Therefore, the procedure described herein can be extended to the full Reynolds-averaged Navier–Stokes equations without modification. Note that the code for the residual computation must include the application of the required boundary conditions (however complex they may be) and any artificial dissipation terms that may need to be added for numerical stability.

A coordinate transformation to computational coordinates (ξ_1, ξ_2, ξ_3) is used. This transformation is defined by the following metrics:

$$K_{ij} = \left[\frac{\partial X_i}{\partial \xi_j}\right], \qquad J = \det(K) \tag{16}$$

$$K^{-1}_{ij} = \left[\frac{\partial \xi_i}{\partial X_j}\right], \qquad S = JK^{-1} \tag{17}$$

where S represents the areas of the face of each cell, projected on to each of the physical coordinate directions. The Euler equations in computational coordinates can then be written as

$$\frac{\partial Jw}{\partial t} + \frac{\partial F_i}{\partial \xi_i} = 0 \tag{18}$$

where the fluxes in the computational cell faces are given by $F_i = S_{ij} f_j$. In semidiscrete form the Euler equations are

$$\frac{\mathrm{d}w_{ijk}}{\mathrm{d}t} + \mathcal{R}_{ijk}(w) = 0 \tag{19}$$

where $\mathcal{R}$ is the residual described earlier with all of its components (fluxes, boundary conditions, artificial dissipation, etc.).

In this specific case, the adjoint equations (13) can be rewritten as

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial I}{\partial w} \tag{20}$$

where ψ is the *adjoint vector*. The total derivative (3) in this case is

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \psi^T \frac{\partial \mathcal{R}}{\partial x} \tag{21}$$

We propose to compute the partial derivative matrices $\partial\mathcal{R}/\partial w$, $\partial I/\partial w$, $\partial I/\partial x$, and $\partial\mathcal{R}/\partial x$ using automatic differentiation instead of manual differentiation or finite differences. Where appropriate, we use the reverse mode of automatic differentiation.

Automatic Differentiation

Automatic differentiation, also known as computational differentiation or algorithmic differentiation, is a well-known method based on the systematic application of the chain rule of differentiation to computer programs. The method relies on tools that automatically produce a program that computes user specified derivatives based on the original program [10].

We denote the *independent* variables as $t_1, t_2, \ldots, t_n$, which for our purposes are the same as the design variables, x. We also need to consider the *dependent* variables, which we write as $t_{n+1}, t_{n+2}, \ldots, t_m$. These are all the intermediate variables in the algorithm, including the outputs I, in which we are interested. We can then write the sequence of operations in any algorithm as

$$t_i = f_i(t_1, t_2, \ldots, t_{i-1}), \qquad i = n+1, n+2, \ldots, m \tag{22}$$

The chain rule can be applied to each of these operations and is written as

$$\frac{\partial t_i}{\partial t_j} = \sum_{k=1}^{i-1} \frac{\partial f_i}{\partial t_k}\frac{\partial t_k}{\partial t_j}, \quad \text{for any } i > j \tag{23}$$

Using the forward mode, we choose one j and keep it fixed. We then work our way forward in the index i until we get the desired derivative. The reverse mode, on the other hand, works by fixing i, the desired quantity we want to differentiate, and working our way backward in the index j all the way down to the independent variables.

The forward and reverse modes are analogous to the direct and adjoint methods, respectively. The counterparts of the state variables in semi-analytic methods are the intermediate variables, and the counterparts of the residual computations are the lines of code that compute the respective quantities.

There are two main ways of implementing automatic differentiation: source code transformation and operator overloading. Tools that use source code transformation add new statements to the original source code that compute the derivatives of the original statements. The operator overloading approach consists of defining a new user-defined type that is used instead of real numbers. This new type includes not only the value of the original variable, but its derivative as well. All the intrinsic operations and functions have to be redefined (overloaded) in order for the derivative to be computed together with the original computations. The operator overloading approach results in fewer changes to the original code, but is usually less efficient [10,28].

There are automatic differentiation tools available for a variety of programming languages including FORTRAN, C/C++, and MATLAB. ADIFOR [29], TAF [30], TAMC [31], and Tapenade [32,33] are some of the tools available for FORTRAN. Of these, only TAF and Tapenade currently support FORTRAN 90, which was a requirement in our case.

We chose to use Tapenade as it is the only noncommercial tool with support for FORTRAN 90. Tapenade is the successor of Odyssée [34], and was developed at the Institut National de Recherche en Informatique et en Automatique (INRIA). It uses source transformation and can perform differentiation in either forward or reverse mode.

To verify the results given by the ADjoint approach we decided to use the complex-step derivative approximation [22,35] as the benchmark. Unlike finite differences, this method is not subject to subtractive cancellation and enables us to make much more conclusive comparisons when it comes to accuracy. The complex-step formula is given by

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\mathrm{Im}[I(x + ih)]}{h} \tag{24}$$

where the design variable is perturbed by a small pure complex step, which is typically less than $\mathcal{O}(10^{-20})$. A point worth noting is that the complex-step method is equivalent to the forward mode of automatic differentiation implemented with operator overloading [36].

Implementation

To compute the desired sensitivities, we need to form the discrete adjoint equation (20), solve it, and then use the total sensitivity equation (21). We use automatic differentiation to generate code that computes the matrices of partial sensitivities present in these equations. In this paper, we compute the sensitivity of the drag coefficient with respect to the freestream Mach number, that is, $I = C_D$ and $x = M_\infty$.

Computation of $\partial\mathcal{R}/\partial w$

The flux Jacobian, $\partial\mathcal{R}/\partial w$, is independent of the choice of function or design variable: It is simply a function of the governing equations, their discretization, and the problem boundary conditions. To compute it we need to consider the routines in the flow solver that, for each iteration, compute the residuals based on the flow variables w. In the following discussion we note that the residual computations are carried out by the SUmb flow solver [37] that was developed at Stanford University under the sponsorship of the Department of Energy. SUmb is a finite volume, cell-centered, multiblock solver for the Reynolds-averaged Navier–Stokes equations (steady, unsteady, and time spectral) and it provides options for a variety of turbulence models with one, two, or four equations [37]. The computation of the residual in SUmb can be summarized as follows:

1) Compute inviscid fluxes: For the inviscid flux discretization we use, the only flow variables w that influence the residual at a cell are the flow variables in that cell and in the six cells that are adjacent to the faces of the cell.

2) Compute dissipation fluxes: For each of the cells in the domain, compute the contributions of the flow variables on the residual in that cell. For this portion of the residual, the flow variables in the current cell and in the 12 adjacent cells in each of the three directions need to be considered.

3) Compute viscous fluxes: This has a similar stencil to the above computation because only the cells adjacent to the current cell need to be considered.

4) Apply boundary conditions.

Note that to compute the residuals over the domain, three nested loops (one for each of the three directions) are used and that the correct value of the residual for any given cell is only obtained at the end of all three loops, when all contributions have been added.

To better illustrate the choices made in the mode of differentiation, as well as the effect of these choices in the general case, we define the following numbers:

N_c: The number of cells in the domain. For three-dimensional domains, where the Navier–Stokes equations are solved, this can be $\mathcal{O}(10^6)$.

N_s: The number of cells in the stencil whose variables affect the residual of a given cell. In our case, we consider inviscid and dissipation fluxes, so the stencil is as shown in Fig. 1 and $N_s = 13$.

N_w: The number of flow variables (and also residuals) for each cell. In our case $N_w = 5$.

Consider the simple subroutine shown in Fig. 2, which resembles a CFD residual calculation. This subroutine loops through a two-dimensional domain and computes `r` (the "residual") in the interior of that domain. The residual at any cell depends only on the `ws` (the "flow variables") at that cell and at the cells immediately adjacent to it; thus the stencil of dependence forms a cross with five cells.

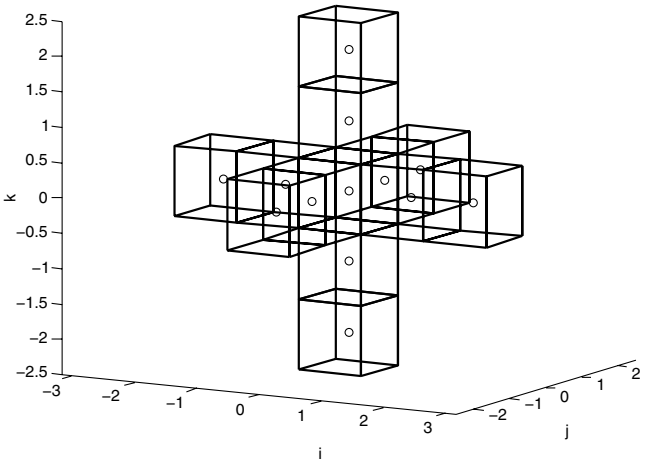

Fig. 1 Stencil for the residual computation.

```
SUBROUTINE RESIDUAL(w, r, ni, nj)
  IMPLICIT NONE
  INTEGER :: ni, nj
  REAL :: r(ni, nj), w(0:ni+1, 0:nj+1)
  INTEGER :: i, j
  INTRINSIC SQRT
  DO i = 1, ni
    DO j = 1, nj
      w(i, j) = w(i, j) + SQRT(w(i, j-1)*w(i, j+1))
      r(i, j) = w(i, j)*w(i-1, j) + SQRT(w(i+1, j)) + &
&               w(i, j-1)*w(i, j+1)
    END DO
  END DO
END SUBROUTINE RESIDUAL
```

Fig. 2 Simplified subroutine for residual calculation.

The residual computation in our three-dimensional CFD solver is obviously much more complicated: It involves multiple subroutines, a larger stencil, the computation of the different fluxes, and applies many different types of boundary conditions. However, this simple example is sufficient to demonstrate the computational inefficiencies of a purely automatic approach.

Forward-mode automatic differentiation was applied to this simple subroutine to produce the subroutine shown in Fig. 3. Two new variables are introduced: `wd`, which is the seed vector, and `rd`, which is the gradient of all `rs` in the direction specified by the seed vector. For example, if we wanted the derivative with respect to `w(1, 1)`, we would set `wd(1, 1) = 1` and all other `wd`s to zero. One can only choose one direction at a time, although Tapenade can be run in vectorial mode to get the whole vector of sensitivities. In vectorial mode, another loop is inserted within the nested loop and additional storage is required.

The corresponding subroutine produced by reverse-mode differentiation is shown in Fig. 4. The code shows the additional storage requirements that are typical of reverse-mode automatic differentiation. Because the `ws` are overwritten, the old values must be stored for later use in the reversed loop. This overwriting of the flow variables in the nested loops of the original subroutine is characteristic of iterative solvers. Whenever overwriting is present, the reverse mode needs to store the time history of the intermediate variables. Tapenade provides the functions `PUSHREAL` and `POPREAL` to do this. In this case, we can see that the `ws` are stored before they are modified in the forward sweep, and then retrieved in the reverse sweep.

In principle, because $\partial\mathcal{R}/\partial w$ is a square matrix, neither mode should have an advantage over the other in terms of computational time. For the simple routine shown above this is true: Both the forward and reverse modes would require $N_c \times N_w$ calls to the residual computation. The reverse mode is slower in practice, due to the additional operations involved. However, using an automatic

```
SUBROUTINE RESIDUAL_D(w, wd, r, rd, ni, nj)
  IMPLICIT NONE
  INTEGER :: ni, nj
  REAL :: r(ni, nj), rd(ni, nj)
  REAL :: w(0:ni+1, 0:nj+1), wd(0:ni+1, 0:nj+1)
  INTEGER :: i, j
  REAL :: arg1, arg1d, result1, result1d
  INTRINSIC SQRT

  rd(1:ni, 1:nj) = 0.0
  DO i = 1, ni
    DO j = 1, nj
      arg1d = wd(i, j-1)*w(i, j+1) + w(i, j-1)*wd(i, j+1)
      arg1 = w(i, j-1)*w(i, j+1)
      IF (arg1d .EQ. 0.0 .OR. arg1 .EQ. 0.0) THEN
        result1d = 0.0
      ELSE
        result1d = arg1d/(2.0*SQRT(arg1))
      END IF
      result1 = SQRT(arg1)
      wd(i, j) = wd(i, j) + result1d
      w(i, j) = w(i, j) + result1
      IF (wd(i+1, j) .EQ. 0.0 .OR. w(i+1, j) .EQ. 0.0) THEN
        result1d = 0.0
      ELSE
        result1d = wd(i+1, j)/(2.0*SQRT(w(i+1, j)))
      END IF
      result1 = SQRT(w(i+1, j))
      rd(i, j) = wd(i, j)*w(i-1, j) + w(i, j)*wd(i-1, j) + &
&                result1d + wd(i, j-1)*w(i, j+1) + &
&                w(i, j-1)*wd(i, j+1)
      r(i, j) = w(i, j)*w(i-1, j) + result1 + &
&               w(i, j-1)*w(i, j+1)
    END DO
  END DO
END SUBROUTINE RESIDUAL_D
```

Fig. 3 Subroutine differentiated using the forward mode.

```
SUBROUTINE RESIDUAL_B(w, wb, r, rb, ni, nj)
  IMPLICIT NONE
  INTEGER :: ni, nj
  REAL :: r(ni, nj), rb(ni, nj)
  REAL :: w(0:ni+1, 0:nj+1), wb(0:ni+1, 0:nj+1)
  INTEGER :: i, j
  REAL :: tempb
  INTRINSIC SQRT
  DO i = 1, ni
    DO j = 1, nj
      CALL PUSHREAL4(w(i, j))
      w(i, j) = w(i, j) + SQRT(w(i, j-1)*w(i, j+1))
    END DO
  END DO
  wb(0:ni+1, 0:nj+1) = 0.0
  DO i = ni, 1, -1
    DO j = nj, 1, -1
      wb(i, j) = wb(i, j) + w(i-1, j)*rb(i, j)
      wb(i-1, j) = wb(i-1, j) + w(i, j)*rb(i, j)
      wb(i+1, j) = wb(i+1, j) + rb(i, j)/(2.0*SQRT(w(i+1, j)))
      wb(i, j-1) = wb(i, j-1) + w(i, j+1)*rb(i, j)
      wb(i, j+1) = wb(i, j+1) + w(i, j-1)*rb(i, j)
      rb(i, j) = 0.0
      CALL POPREAL4(w(i, j))
      tempb = wb(i, j)/(2.0*SQRT(w(i, j-1)*w(i, j+1)))
      wb(i, j-1) = wb(i, j-1) + w(i, j+1)*tempb
      wb(i, j+1) = wb(i, j+1) + w(i, j-1)*tempb
    END DO
  END DO
END SUBROUTINE RESIDUAL_B
```

Fig. 4 Subroutine differentiated using the reverse mode.

differentiation tool on the entire residual routine results in unnecessary computations. For CFD solvers, the residual is only dependent on a stencil of cells adjacent to it, and thus the flux Jacobian is very sparse. Automatic differentiation of the entire routine does not take this sparsity into account, and much effort would be wasted in the computation of zeros. When using the reverse mode, there would also be the added waste of storing the intermediate variables that are required for these superfluous computations. Therefore, to avoid the automatic differentiation of nested loops over the whole computational domain, and to take advantage of the sparsity in the Jacobian, we created a routine that computes the residuals for a single cell at a time. The routine, which is actually a set of routines due to the complexity of the computation, mimics the original computation of residuals exactly, but without the nested loops over the domain.

We do not consider the viscous fluxes in the present work, only the inviscid fluxes, dissipation fluxes, and the boundary conditions. The

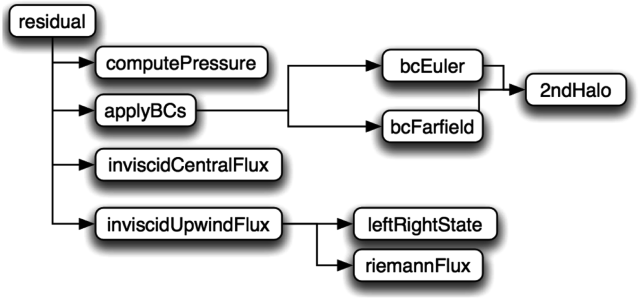

Fig. 5 Call graph for the original residual calculation.

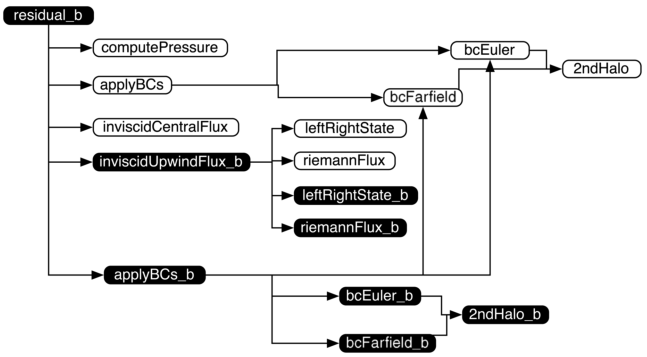

Fig. 6 Call graph for the differentiated residual calculation. Shaded boxes indicate differentiated routines.

stencil that affects a given cell when considering these fluxes is shown in Fig. 1. Note that the code to compute the residual in a given cell in the domain is easily constructed from the original residual evaluation routines in the flow solver by removing the loops over all the cells in the domain and making necessary adjustments so that the appropriate boundary conditions are called for every cell in the stencil.

We now have a routine that computes N_w residuals in a given cell. These residuals get contributions from all $(N_w \times N_s)$ flow variables in the stencil. Thus there are $N_w \times (N_w \times N_s)$ sensitivities to be computed for each cell, corresponding to N_w rows in the $\partial\mathcal{R}/\partial w$ matrix. Each of these rows contains no more than $(N_w \times N_s)$ nonzero entries. In the reverse mode, this routine is called once and computes all $N_w \times (N_w \times N_s)$ terms in the stencil of this cell.

The analog in the forward mode would be to consider all of the residuals affected by the states in a single cell. Theoretically, one could compute the derivatives of the $N_w \times N_s$ residuals affected by a single state, thus for a single cell, one would again obtain $N_w \times (N_w \times N_s)$ derivatives. However, due to the one-way dependence of the residual on the states, this does not prove to be the case. In the reverse mode, all of the derivatives in the stencil can be calculated from one residual calculation. All of the information for that calculation is contained in a single stencil. For the forward mode, one would need to calculate all of the $N_w \times N_s$ residuals in the inverse stencil. This requires the addition of all of the states in those stencils as well. Thus, rather than having a single calculation with $N_w \times N_s$ states involved to get $N_w \times (N_w \times N_s)$ derivative values, one requires N_s residual calculations and many additional states to get the same number of derivative components.

Given these differences, the reverse mode is the more efficient approach to differentiating the single cell residual routine. The call graphs for the original and the differentiated routines are shown in Figs. 5 and 6, respectively.

We should also note that, with advanced automatic differentiation tools, it is also possible to get similar improvements in efficiency for forward-mode computations [21]. These tools use seed variable coloring schemes and other techniques to reduce the computational cost of computing sparse Jacobians.

Computation of $\partial C_D/\partial w$

The right-hand side vector of the adjoint equation (20)—or matrix, in the case of multiple functions of interest—represents the direct effect of the flow variables on the function of interest. Normally, to generate this vector using the ADjoint method, we would identify the code that computes C_D as a function of the state variables. We would then differentiate this code using the reverse mode. However, because we know the details of the computation of C_D, we have made a more specific implementation, which is discussed next.

Because there may be several functions of interest that need to be considered, the derivatives have been generated in a modular fashion. This allows different portions of the formulation to be interchanged as needed for various functions of interest. In the cases shown here, the functions of interest are C_D and C_L, so the derivatives we need are $\partial C_D/\partial w$ and $\partial C_L/\partial w$. As with the residual equations, modified versions of the original functions were used to compute the derivatives. However, rather than differentiating the entire path from w to C_D and C_L at once, the terms were split up to allow for the modular approach previously mentioned. In this example, because we have inviscid flow, C_D and C_L are just integrations of the pressure over the solid surfaces of the mesh. Thus, we know a priori that the only changes in C_D and C_L come through changes in the pressure. As a result, $\partial C_D/\partial w$ can be separated via the chain rule into three terms to simplify the automatic differentiation and increase the flexibility of the code. To this end, the right-hand side of the adjoint was expressed as

$$\frac{\partial C_D}{\partial w} = \frac{\partial C_D}{\partial C_f}\frac{\partial C_f}{\partial p}\frac{\partial p}{\partial w} \tag{25}$$

where C_f represents the x, y, and z components of the forces on the body. Using this approach, changing from $\partial C_D/\partial w$ to $\partial C_L/\partial w$ only requires recomputing the first term.

The first two terms in this expression are easily calculated. The first one is simply a function of the drag direction relative to the coordinate axes of the forces, while the second can be computed by reintegrating the pressures over the surface of the solid. In our case, each of these terms was expressed as a single function and differentiated in forward mode. It is worthwhile to note that for these two functions in particular, the reverse mode would be more advantageous than the forward mode. Both functions have fewer output variables than input variables: For $\partial C_D/\partial C_f$ the ratio is 1:3, whereas for $\partial C_f/\partial p$ the ratio is approximately $3{:}N_c$. However, because both functions are relatively simple, we did not make it a priority to use the reverse mode of differentiation. Note that because of the way the pressure is calculated in our flow solver, the halo cells need to be included in these pressure derivatives.

The computation of the final term, $\partial p/\partial w$, is slightly more involved than the previous two. The functions used in this computation were again differentiated in forward mode. In this case, because we need the derivative of the pressure in the halo cells with respect to the internal states, the forward mode is actually better suited than the reverse mode, because there are more output variables than input variables.

As discussed previously, the residual computations are done on a stencil-by-stencil basis. However, unlike the residual, there is a one-to-one cell relationship between the flow states and the pressure. Thus, for each state, all of the derivatives in the stencil can be computed at once, even in forward mode. Furthermore, because of the definition of the stencil, any cells outside the stencil have a derivative of zero with respect to that state. Thus, for a single stencil computation an entire row of the $\partial p/\partial w$ matrix can be generated. Nevertheless, a set of four nested loops is required to generate the derivatives with respect to all of the states. With this in mind, the required process to compute the derivative $\partial p/\partial w$ is as follows:

1) Seed the desired state: For each of the $N_w \times N_c$ states in the problem, the seed vector must be set to unity. For each of these cases an entire vector of pressure derivatives is generated.

2) Recalculate pressures: For each cell in the stencil around w, recompute the pressure. Pressures outside this stencil are not affected by this state.

3) Apply boundary conditions: These are applied to each cell in the stencil and they modify both the states and the pressures in the halo cells, which are required to compute the correct derivative.

At the end of this process, the complete $\partial p/\partial w$ matrix is available from the differentiated code. With these three terms, we can compute $\partial C_D/\partial w$, and proceed with the solution of the adjoint equations.

Adjoint Solver

The adjoint equation (20) can be rewritten for this specific case as

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial C_D}{\partial w} \tag{26}$$

As we have pointed out, both the flux Jacobian and the right-hand side in this system of equations are very sparse. To solve this system efficiently, and having in mind that we want to have a parallel adjoint solver, we decided to use PETSc (portable, extensible toolkit for scientific computation) [38–40]. PETSc is a suite of data structures and routines for the scalable, parallel solution of scientific applications modeled by PDEs. It employs the message passing interface (MPI) standard for all interprocessor communication. Using PETSc's data structures, $\partial \mathcal{R}/\partial w$ and $-\partial C_D/\partial w$ were stored as sparse entities. Once the sparse matrices are filled, one of PETSc's Krylov solvers is used to compute the adjoint solution.

Total Derivative Equation

The total derivative (21) for this case is

$$\frac{\mathrm{d}C_D}{\mathrm{d}M_\infty} = \frac{\partial C_D}{\partial M_\infty} + \psi^T \frac{\partial \mathcal{R}}{\partial M_\infty} \tag{27}$$

where we have chosen the freestream Mach number, M_∞, to be the independent variable. There are only two remaining terms required to form the total derivative equation: $\partial C_D/\partial M_\infty$ and $\partial \mathcal{R}/\partial M_\infty$. The former term represents a very simple dependence. The only direct impact of M_∞ on C_D is through the nondimensionalization of the term, and thus it is possible to compute it analytically. However, in the spirit of the ADjoint method, the functions used for $\partial C_D/\partial w$ earlier were again differentiated automatically, this time with respect to M_∞, thus providing $\partial C_D/\partial M_\infty$. The differentiation of this term was done in forward mode as there is only one input.

The final term, $\partial \mathcal{R}/\partial M_\infty$, shares many components with the flux Jacobian, $\partial \mathcal{R}/\partial w$, which we discussed previously. Thus, much of the same logic regarding the use of a single cell residual calculation applies here. In this case, however, the result is a vector as opposed to a matrix, so the residual routines were differentiated in forward mode. As with the right-hand side of the adjoint equations, a modular approach was used to compute this term, which we write as

$$\frac{\partial \mathcal{R}}{\partial M_\infty} = \frac{\partial \mathcal{R}}{\partial w_\infty}\frac{\partial w_\infty}{\partial M_\infty} \tag{28}$$

To compute the first term, $\partial \mathcal{R}/\partial w_\infty$, the routines for $\partial \mathcal{R}/\partial w$ were reused. However, in this case, because there are only five components in w_∞, the forward mode of differentiation is far more efficient. Thus, the routines were again automatically differentiated, this time in forward mode with respect to w_∞ instead of w.

The last term, $\partial w_\infty/\partial M_\infty$, is very straightforward and can be verified analytically. Of the five states, only four depend directly on M_∞. The second to fourth states are velocities and vary linearly with M_∞, while the last state is related to total energy and is proportional to M_∞^2. To compute these derivatives, a new function that combined all of these dependencies was created and then differentiated in forward mode. Again, this is because the number of output variables far exceeds the number of input variables (by 5:1).

The implementation we describe is only for $\partial C_D/\partial M_\infty$. The real benefits of this method arise when there are large numbers of design variables, as is usually the case for aerodynamic shape optimization. In such problems, the sensitivity of C_D with respect to all of the shape variables can be found from a single ADjoint solution. The basic implementation for such a case would not change significantly,

because the adjoint equations remain exactly the same. The modifications to accommodate shape variables would come from two terms in the total sensitivity equation, $\partial\mathcal{R}/\partial x$ and $\partial C_D/\partial x$. These two partial derivatives would have to be modified to include the geometry related procedures: shape parameterization, mesh movement, and the spatial metrics in the CFD calculation.

Results

Description of Test Cases

Two test cases have been used to demonstrate the ADjoint approach: a channel flow over a bump, and a subsonic wing. For the bump case, the front and back walls of the channel have symmetry boundary conditions imposed on them, while the top wall is flat and the bottom wall was deformed with a sinusoidal bump to create a reasonable variation in the flow. The inflow and outflow faces of the domain have nonreflecting boundary conditions imposed on them and the upper and lower walls use a linear pressure extrapolation boundary condition. The freestream Mach number is 2. The mesh for this test case is shown in Fig. 7 and its size is $48 \times 24 \times 24$. Figure 8 shows the surface density distribution.

The second case is the Lockheed–Air Force–NASA–NLR (LANN) wing [41], which is a supercritical transonic transport wing. A symmetry boundary condition is used at the root and a linear pressure extrapolation boundary condition is used on the wing surface. The freestream Mach number is 0.621. The mesh for this test case is shown in Fig. 9 and its size is $64 \times 16 \times 12$. Figure 10 shows the surface density distribution.

Flux Jacobian

As mentioned in the Implementation section, to reduce the cost of the automatic differentiation procedure, we wrote a set of subroutines that compute the residual of a single cell at a time. This set of subroutines closely resembles the original code used to compute the residuals over the whole domain, except that it does not loop over the domain.

The residual of the chosen cell depends on the flow variables in the stencil of dependence of that cell. Although this new code is essentially produced by duplicating the original code, it was still necessary to verify that the residuals were the same as the original code for debugging purposes. To this end, the l^2 norm of the difference between the new and original residuals was computed and found to be $\mathcal{O}(10^{-17})$ (i.e., machine zero), thus proving the validity of the new calculations.

As previously discussed, we differentiated the single cell residual calculation routines using the reverse mode of automatic differentiation. The resulting code computes the sensitivity of one residual of the chosen cell with respect to all the flow variables in the stencil of that cell, that is, all the nonzero terms in the corresponding row of $\partial\mathcal{R}/\partial w$. To obtain the full flux Jacobian, a loop over each of the residual and cell indices in each of the three coordinate directions was necessary.

To verify the flux Jacobian obtained with the differentiated code, the relative error between the automatic differentiation and the finite difference results is shown in Fig. 11. The quantity considered in this figure is the sum of the derivative for each residual with respect to all of the states that affect it. Figure 11 shows the error in this cumulative quantity. The errors are within an acceptable range, varying between $\mathcal{O}(10^{-10})$ and $\mathcal{O}(10^{-6})$. In this case, however, it is evident that the finite difference results incur the larger error, because we later show that the automatic differentiation adjoint yields an $\mathcal{O}(10^{-14})$ error in the final sensitivities.

Comparing the time for the computation of the residuals of one cell with the time for running the reverse mode of the same computation, we found that the reverse mode was equivalent to 4.5 residual computations. A factor of this order is typical of reverse-mode codes. In spite of this penalty, our automatic differentiation approach is much more efficient in computing the flux Jacobian, as shown in Table 1. Here we compare the times needed for computing the full flux Jacobian. We can see that the automatic differentiation approach was 29 times faster than finite differencing, confirming what we

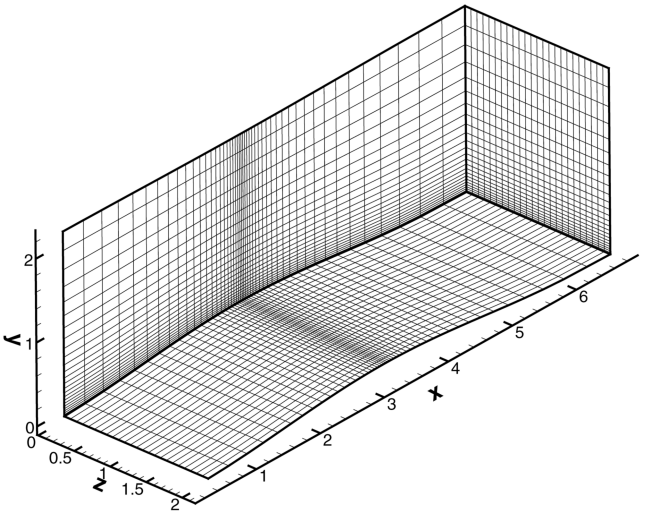

Fig. 7 Computational mesh for the bump case.

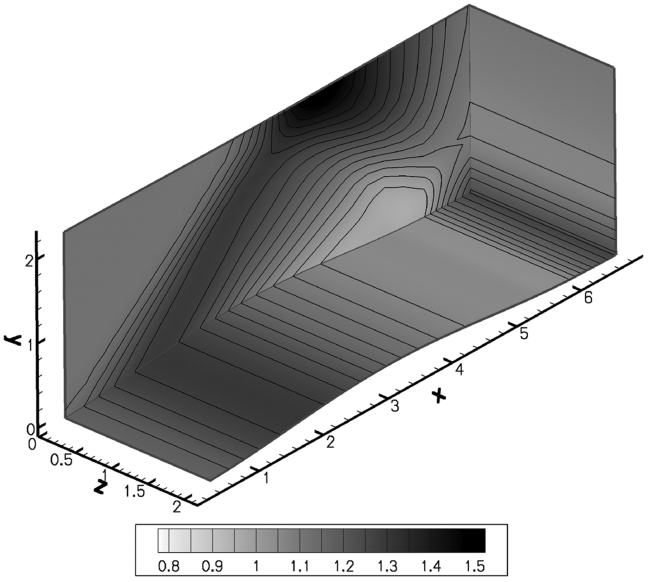

Fig. 8 Surface density distribution for the bump case.

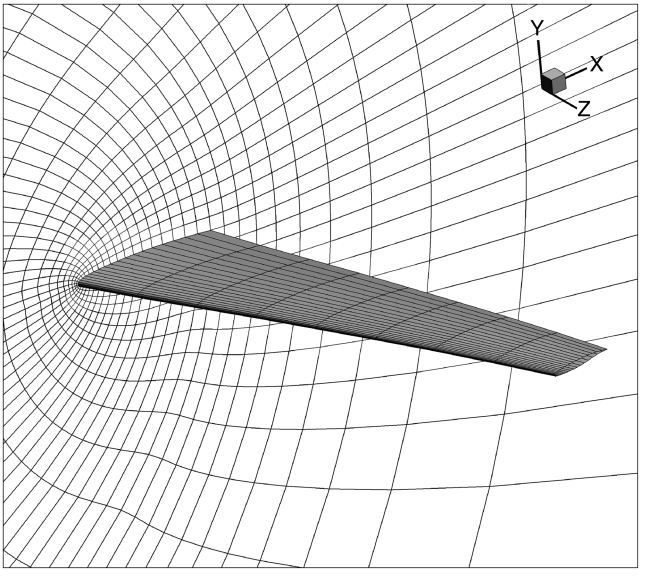

Fig. 9 Computational mesh for the wing case.

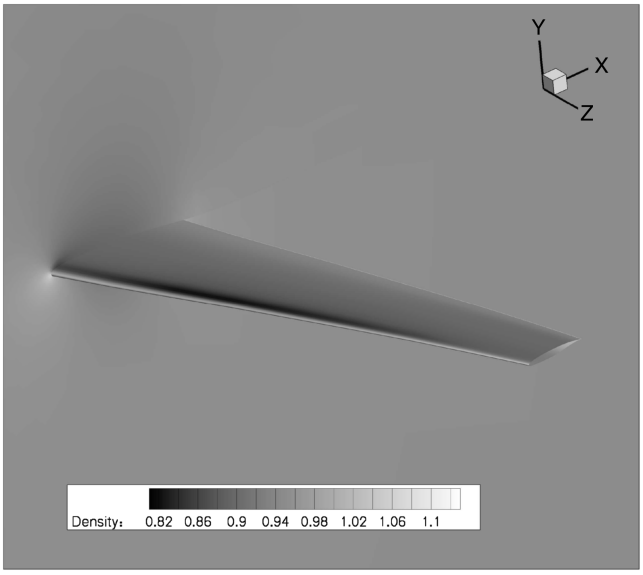

Fig. 10 Surface density distribution for the wing case.

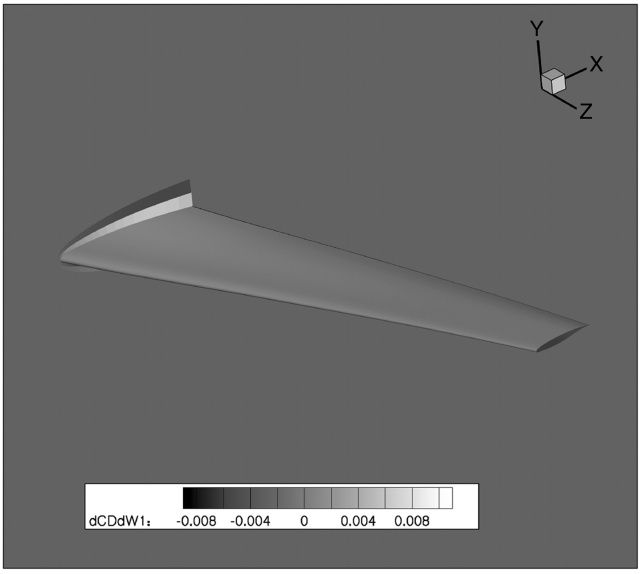

Fig. 12 Partial sensitivity of C_D with respect to density.

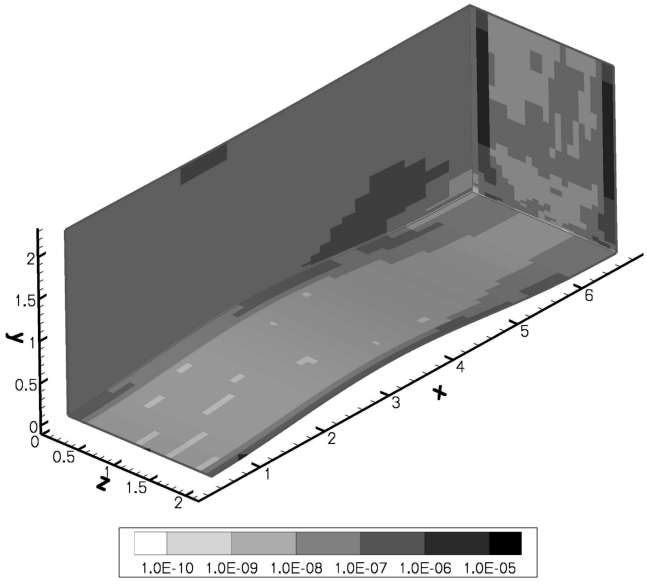

Fig. 11 Relative errors in the sum of each row of $\partial\mathcal{R}/\partial w$.

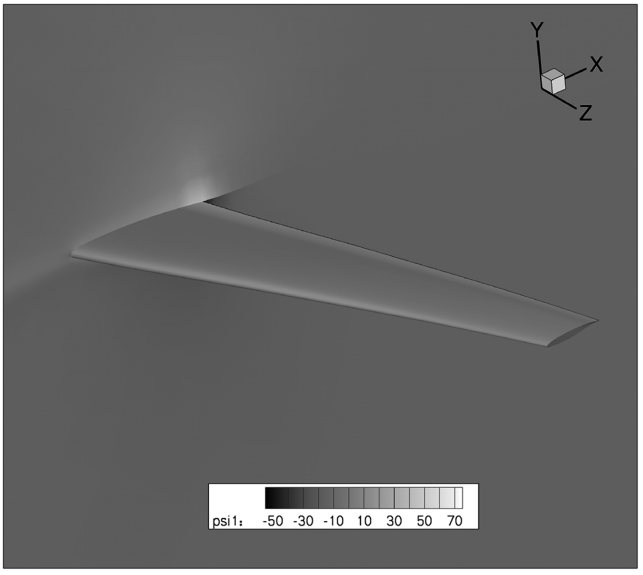

Fig. 13 Adjoint of the continuity equation.

speculated in the previous section. The reason is that the reverse-mode calculation does not need to be called as many times as the finite difference residual evaluations.

Other Partial Derivative Terms

In the process of developing the ADjoint solver, we visualized the various partial derivative terms that appear in the adjoint and total derivative equations. Figure 12 shows the derivative of drag coefficient with respect to the density at each cell. The only nonzero terms are located in the two layers of cells adjacent to the wing surface, which is what we expected.

The adjoint variables corresponding to the continuity equation are shown in Fig. 13. The adjoint variables in this case represent the sensitivity of C_D with respect to the residual of the continuity equation at a given cell.

The term $\partial\mathcal{R}/\partial M_\infty$, which is needed for the total sensitivity equation (27), is shown in Fig. 14. The figure only shows the sensitivity corresponding to the residual of the continuity equation. Because the flow is subsonic, M_∞ affects the cells near the inflow and outflow planes. Because of the shape of the mesh and the nature of the flow, the entire outer surface is affected by the freestream conditions.

Table 1 Times and error for flux Jacobian computation

Finite differences	425.4 s
Automatic differentiation	14.6 s
Speed increase	29×
l^2 norm of error	4.25×10^{-7}

Lift and Drag Coefficient Sensitivities

The benchmark sensitivity results were obtained using the complex-step derivative approximation (24), which is numerically exact, that is, the precision of the sensitivity is of the same order as the precision of the solution. The derivative in this case is given by

$$\frac{\mathrm{d}C_D}{\mathrm{d}M_\infty} = \frac{\mathrm{Im}[C_D(M_\infty + ih)]}{h} \tag{29}$$

where h represents the magnitude of the complex step, for which a value of $h = 10^{-20}$ was used. The sensitivities given by the complex

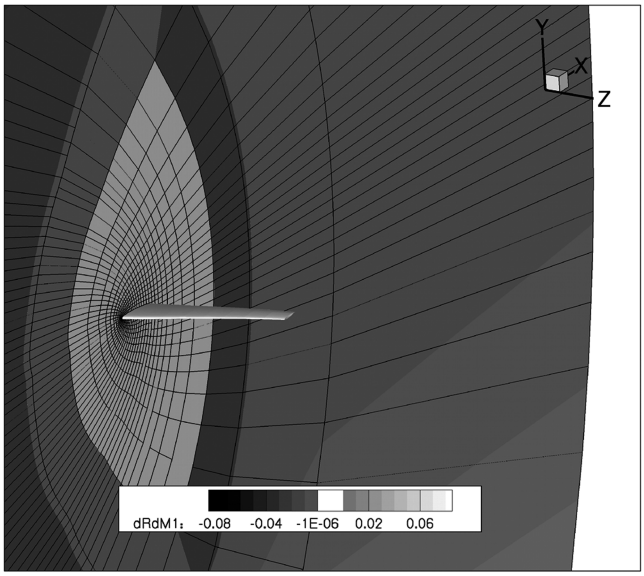

Fig. 14 Partial sensitivity of the continuity residual with respect to M_∞.

step were spot checked against finite differences by doing a step-size study.

In Table 2 we show the derivatives of both drag and lift coefficients with respect to freestream Mach number for different cases. The wing mesh and the fine mesh cases are the two test cases described earlier (Figs. 7 and 9). The coarse case is a very small mesh ($12 \times 4 \times 4$) with the same geometry as the fine bump case.

We can see that the adjoint sensitivities for these cases are extremely accurate, yielding an agreement of 12–14 digits when compared to the complex-step results. This is consistent with the convergence tolerance that was specified in PETSc for the adjoint solution.

To analyze the performance of the ADjoint solver, several timings were performed. They are shown in Table 3 for the two larger cases mentioned above. The fine grid has 203,840 flow variables and the wing grid has 108,800 flow variables.

The total cost of the adjoint solver, including the computation of all the partial derivatives and the solution of the adjoint system, is 24% of the cost for a flow solution for the fine bump case, and 11% for the wing case.

We found that most of the time was spent in the solution of the adjoint equations, thus indicating that the automatic differentiation sections are very efficient. The costliest of the automatic differentiation routines was the computation of the flux Jacobian. When one takes into consideration the number of terms in this matrix, spending only 5% of the flow solution time in this computation is very reasonable.

However, the computation of the right-hand side ($\partial C_D/\partial w$) is not much cheaper than computing the flux Jacobian. The poor performance in the computation of this term is primarily due to the use of the forward mode. The use of reverse-mode differentiation would allow the cost of this term to be independent of the grid size.

Also, although we did not perform a rigorous measurement of the memory requirements of this code, our observations indicate that the memory required for the ADjoint code is approximately 10 times that required for the original flow solver. Given the pattern of usage on current parallel CFD computations, this is acceptable in many cases.

Discussion

In the Results section of this paper, we presented accuracy and timing results for the ADjoint approach. These results conclusively show that the ADjoint approach can produce accurate derivatives in a computationally efficient manner. However, there are a few points about this approach that deserve discussion.

In this implementation, the elements of the flux Jacobian are computed once, using the reverse mode of automatic differentiation, and stored in memory for the remainder of the adjoint solution. This allows for the use of powerful preconditioned linear solvers such as those provided by PETSc. It is this storage of the flux Jacobian matrix that causes the bulk of the memory increase described in the results. Similar memory increases would be observed if the flux Jacobian matrix were computed using analytic methods instead of reverse-mode automatic differentiation. The actual memory increase caused by the reverse mode is minimal compared to the other memory use in the code because of the small stencil.

A similar solution scheme could be implemented using analytic derivatives, where the only difference in performance would come in the time required to compute the partial derivatives. Because those partial derivatives are only computed once in the solution method described, the potential computational savings available from analytic derivatives would not be very significant.

An alternative to this approach would be to use a matrix free solution algorithm. Although this would lower the memory usage, it would also result in higher computational costs. This is because the partial derivatives would need to be recomputed multiple times. In such an approach, the cost savings resulting from an analytic implementation would be much more significant.

The most significant advantages of the ADjoint approach are that it is easily applicable to a broad range of governing equations, and that it significantly reduces the implementation time required relative to an analytic adjoint method. Given the fact that many CFD runs performed on modern parallel computers tend to underutilize the available memory, the authors feel that the benefits achieved in terms of reduced implementation time outweigh this memory penalty.

Conclusions

We have presented an approach for developing discrete adjoint solvers for arbitrary governing equations using automatic differentiation. The implementation of the ADjoint approach was largely automatic, because no hand differentiation or calculation of adjoint boundary conditions was necessary. Thanks to the use of automatic differentiation, the implementation time was greatly reduced and the resulting derivatives were numerically exact. The agreement between the total derivatives computed with the ADjoint method and the benchmark results ranged from 12 to 14 digits.

The stencil-based approach, in conjunction with the reverse mode of automatic differentiation resulted in a very efficient flux Jacobian

Table 2 Derivatives of drag and lift coefficients with respect to M_∞

Mesh	Coefficient	Inflow direction	ADjoint	Complex step
Coarse	C_D	(1,0,0)	−0.289896632731764	−0.289896632731759
	C_L		−0.26770445536666714	−0.26770445536668683
Fine	C_D	(1,0,0)	−0.0279501183024705	−0.0279501183024709
	C_L		0.58128604734707	0.58128604734708
Coarse	C_D	(1,0.05,0)	−0.278907645833786	−0.278907645833792
	C_L		−0.262086315233911	−0.262086315233875
Fine	C_D	(1,0.05,0)	−0.0615598631060438	−0.0615598631060444
	C_L		−0.364796754652787	−0.364796754652797
Wing	C_D	(1, 0.0102,0)	0.00942875710535217	0.00942875710535312
	C_L		0.26788212595474	0.26788212595468

Table 3 ADjoint computational cost breakdown (times in seconds)

	Fine	Wing
Flow solution	219.215	182.653
ADjoint	51.959	20.843
Breakdown:		
Setup PETSc variables	0.011	0.004
Compute flux Jacobian	11.695	5.870
Compute RHS	8.487	2.232
Solve the adjoint equations	28.756	11.213
Compute the total sensitivity	3.010	1.523

computation. In the case presented herein, the finite difference approach to computing the flux Jacobian was shown to be 29 times longer than our approach.

The overall computational cost of the ADjoint method exceeded our own expectations, with the complete adjoint computation ranging between one-eighth and one-fourth of the flow solution.

Although there is a significant memory penalty associated with the ADjoint approach, we believe that in most cases this is outweighed by the substantial benefits over current methodologies used to develop adjoint solvers.

Currently, only highly specialized research groups have been able to develop adjoint codes for aerodynamic shape optimization. The ADjoint approach is likely to facilitate the development of adjoint codes, both in academia and industry, and thus contribute to a more widespread use of this powerful sensitivity analysis approach.

Acknowledgments

C. A. Mader and J. R. R. A. Martins are grateful for the funding provided by the Canada Research Chairs program and the Natural Sciences and Engineering Research Council. All of the authors would like to thank Valerie Pascual and Laurent Hascoët of the Tapenade development team for being so responsive to our questions and suggestions. The authors would also like to thank André Marta and Ki Hwan Lee for their assistance with the flow solver.

References

[1] Pironneau, O., "On Optimum Design in Fluid Mechanics," *Journal of Fluid Mechanics*, Vol. 64, No. 1, June 1974, pp. 97–110. doi:10.1017/S0022112074002023

[2] Jameson, A., "Aerodynamic Design via Control Theory," *Journal of Scientific Computing*, Vol. 3, No. 3, Sept. 1988, pp. 233–260. doi:10.1007/BF01061285

[3] Driver, J., and Zingg, D. W., "Numerical Aerodynamic Optimization Incorporating Laminar-Turbulent Transition Prediction," *AIAA Journal*, Vol. 45, No. 8, 2007, pp. 1810–1818. doi:10.2514/1.23569

[4] Nemec, M., Zingg, D. W, and Pulliam, T. H., "Multipoint and Multi-Objective Aerodynamic Shape Optimization," *AIAA Journal*, Vol. 42, No. 6, June 2004, pp. 1057–1065.

[5] Reuther, J. J., Jameson, A., Alonso, J. J., Rimlinger, M. J., and Saunders, D., "Constrained Multipoint Aerodynamic Shape Optimization Using an Adjoint Formulation and Parallel Computers, Part 1," *Journal of Aircraft*, Vol. 36, No. 1, 1999, pp. 51–60.

[6] Reuther, J. J., Jameson, A., Alonso, J. J., Rimlinger, M. J., and Saunders, D., "Constrained Multipoint Aerodynamic Shape Optimization Using an Adjoint Formulation and Parallel Computers, Part 2," *Journal of Aircraft*, Vol. 36, No. 1, 1999, pp. 61–74.

[7] Martins, J. R. R. A., Alonso, J. J., and Reuther, J. J., "High-Fidelity Aerostructural Design Optimization of a Supersonic Business Jet," *Journal of Aircraft*, Vol. 41, No. 3, 2004, pp. 523–530.

[8] Maute, K., Nikbay, M., and Farhat, C., "Coupled Analytical Sensitivity Analysis and Optimization of Three-Dimensional Nonlinear Aeroelastic Systems," *AIAA Journal*, Vol. 39, No. 11, 2001, pp. 2051–2061.

[9] Martins, J. R. R. A., Alonso, J. J., and Reuther, J. J., "A Coupled-Adjoint Sensitivity Analysis Method for High-Fidelity Aero-Structural Design," *Optimization and Engineering*, Vol. 6, No. 1, March 2005, pp. 33–62. doi:10.1023/B:OPTE.0000048536.47956.62

[10] Griewank, A., *Evaluating Derivatives*, SIAM, Philadelphia, 2000.

[11] Fagan, M., and Carle, A., "Reducing Reverse-Mode Memory Requirements by Using Prole-Driven Checkpointing," *Future Generation Computer Systems*, Vol. 21, No. 8, 2005, pp. 1380–1390. doi:10.1016/j.future.2004.11.005

[12] Cusdin, P., and Müller, J.-D., "On the Performance of Discrete Adjoint CFD Codes Using Automatic Differentiation," *International Journal of Numerical Methods in Fluids*, Vol. 47, Nos. 8–9, 2005, pp. 939–945.

[13] Giering, R., Kaminski, T., and Slawig, T., "Generating Efficient Derivative Code with TAF: Adjoint and Tangent Linear Euler Flow Around an Airfoil." *Future Generation Computer Systems*, Vol. 21, No. 8, 2005, pp. 1345–1355. doi:10.1016/j.future.2004.11.003

[14] Heimbach, P., Hill, C., and Giering, R., "An Efficient Exact Adjoint of the Parallel MIT General Circulation Model, Generated via Automatic Differentiation," *Future Generation Computer Systems*, Vol. 21, No. 8, 2005, pp. 1356–1371. doi:10.1016/j.future.2004.11.010

[15] Martins, J. R. R. A., Alonso, J. J., and van der Weide, E., "An Automated Approach for Developing Discrete Adjoint Solvers," AIAA Paper 2006-1608, 2006.

[16] Martins, J. R. R. A., Mader, C. A., and Alonso, J. J., "ADjoint: An Approach for Rapid Development of Discrete Adjoint Solvers," AIAA Paper 2006-7121, 2006.

[17] Dwight, R. P., and Brezillion, J., "Effect of Approximations of the Discrete Adjoint on Gradient-Based Optimization," *AIAA Journal*, Vol. 44, No. 12, 2006, pp. 3022–3031. doi:10.2514/1.21744

[18] Kim, C. S., Kim, C., and Rho, O. H., "Sensitivity Analysis for the N-S Equations with Two-Equation Turbulence Models," *AIAA Journal*, Vol. 39, No. 5, 2001, pp. 838–845.

[19] Nielsen, E. J., and Anderson, W. K., "Aerodynamic Design Optimization on Unstructured Meshes Using the Navier-Stokes Equations," *AIAA Journal*, Vol. 37, No. 11, 1999, pp. 1411–1419.

[20] Marta, A. C., and Alonso, J. J., "High-Speed MHD Flow Control Using Adjoint-Based Sensitivities," AIAA Paper 2006-8009, 2006.

[21] Nielsen, E. J., and Kleb, W. L., "Efficient Construction of Discrete Adjoint Operators on Unstructured Grids Using Complex Variables," *AIAA Journal*, Vol. 44, No. 4, 2006, pp. 827–836.

[22] Martins, J. R. R. A., Sturdza, P., and Alonso, J. J., "The Complex-Step Derivative Approximation," *ACM Transactions on Mathematical Software*, Vol. 29, No. 3, 2003, pp. 245–262. doi:10.1145/838250.838251

[23] Giles, M. B., and Pierce, N. A., "An Introduction to the Adjoint Approach to Design," *Flow, Turbulence and Combustion*, Vol. 65, Nos. 3–4, Dec. 2000, pp. 393–415. doi:10.1023/A:1011430410075

[24] Newman, J. C., III, Taylor, A. C., III, Barnwell, R. W., Newman, P. A., and Hou, G. J.-W., "Overview of Sensitivity Analysis and Shape Optimization for Complex Aerodynamic Congurations," *Journal of Aircraft*, Vol. 36, No. 1, 1999, pp. 87–96.

[25] Anderson, W. K., and Venkatakrishnan, V., "Aerodynamic Design Optimization on Unstructured Grids with a Continuous Adjoint Formulation," *Computers and Fluids*, Vol. 28, No. 4, 1999, pp. 443–480. doi:10.1016/S0045-7930(98)00041-3

[26] Nadarajah, S., and Jameson, A., "A Comparison of the Continuous and Discrete Adjoint Approach to Automatic Aerodynamic Optimization," AIAA Paper 2000-0667, 2000.

[27] Giles, M. B., Duta, M. C., Müller, J.-D., and Pierce, N. A., "Algorithm Developments for Discrete Adjoint Methods," *AIAA Journal*, Vol. 41, No. 2, Feb. 2003, pp. 198–205.

[28] Pryce, J. D., and Reid, J. K., "AD01, A Fortran 90 Code for Automatic Dierentiation," Rutherford Appleton Laboratory, Oxfordshire, U.K., Rept. RAL-TR-1998-057, 1998.

[29] Carle, A., and Fagan, M., "ADIFOR 3.0 Overview," Rice University, TR CAAM-TR-00-02, 2000.

[30] Giering, R., Kaminski, T., and Slawig, T., "Generating Efficient Derivative Code with TAF: Adjoint and Tangent Linear Euler Flow Around an Airfoil," *Future Generation Computer Systems*, Vol. 21, No. 8, 2005, pp. 1345–1355. doi:10.1016/j.future.2004.11.003

[31] Gockenbach, M. S., "Understanding Code Generated by TAMC," IAAA Paper TR00-29, Department of Computational and Applied Mathematics, Rice University, TX, 2000.

[32] Hascoët, L., and Pascual, V., "TAPENADE 2.1 User's Guide," INRIA, TR 300, 2004.

[33] Pascual, V., and Hascoët, L., "Extension of TAPENADE Towards Fortran 95," *Auto-Matic Differentiation: Applications, Theory, and Tools*, edited by H. M. Bücker, G. Corliss, P. Hovland, U. Naumann,

and B. Norris, Lecture Notes in Computational Science and Engineering, Springer, New York, 2005.

[34] Faure, C., and Papegay, Y., *Odyssée Ver. 1.6: The Language Reference Manual*, INRIA, Rapport Technique 211, 1997.

[35] Squire, W., and Trapp, G., "Using Complex Variables to Estimate Derivatives of Real Functions," *SIAM Review*, Vol. 40, No. 1, 1998, pp. 110–112. doi:10.1137/S003614459631241X

[36] Martins, J. R. R. A., Sturdza, P., and Alonso, J. J., "The Connection Between the Complex-Step Derivative Approximation and Algorithmic Differentiation," AIAA, 2001-0921, Jan. 2001.

[37] van der Weide, E., Kalitzin, G., Schluter, J., and Alonso, J. J., "Unsteady Turbomachinery Computations Using Massively Parallel Platforms," AIAA, 2006-0421, 2006.

[38] Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H., "PETSc Web page," 2001, http://www.mcs.anl.gov/petsc.

[39] Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H., "PETSc Users Manual," Argonne National Laboratory, TR ANL-95/11-Revision 2.1.5, 2004.

[40] Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F., "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," *Modern Software Tools in Scientic Computing*, edited by E. Arge, A. M. Bruaset, and H. P. Langtangen, Birkhäuser Press, Boston, MA, 1997, pp. 163–202.

[41] Ruo, S. Y., Malone, J. B., Horsten, J. J., and Houwink, R., "The LANN Program—An Experimental and Theoretical Study of Steady and Unsteady Transonic Airloads on a Supercritical Wing," AIAA Paper 1983-1686, July 1983.

N. Alexandrov
Associate Editor