AIAA 2006–7121

ADjoint: An Approach for Rapid Development of Discrete Adjoint Solvers

Joaquim R. R. A. Martins, Charles A. Mader
University of Toronto
Toronto, Ontario, Canada, M3H 5T6

Juan J. Alonso
Stanford University
Stanford, CA 94305, U.S.A.

11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference,
6–8 September 2006, Portsmouth, Virginia

ADjoint: An Approach for Rapid Development of Discrete Adjoint Solvers

Joaquim R. R. A. Martins[*], Charles A. Mader[†]
University of Toronto
Toronto, Ontario, Canada, M3H 5T6

Juan J. Alonso[‡]
Stanford University
Stanford, CA 94305, U.S.A.

An automatic differentiation tool, Tapenade, is used to develop the adjoint code for a three-dimensional computational fluid dynamics (CFD) solver. Rather than using automatic differentiation to differentiate the complete source code of the CFD solver (including the iterative loop), we have applied it selectively to produce code that computes both the flux Jacobian matrix and other partial derivatives that are necessary to compute sensitivities using an adjoint method. The resulting linear discrete adjoint system is then solved using the toolkit for parallel scientific computations, PETSc. This approach to deriving and implementing the discrete adjoint equations has the advantages that it is applicable to arbitrary sets of governing equations and cost functions, it is *exactly consistent* with the gradients that would be computed by exact numerical differentiation of the original solver, and it is automatic, thus avoiding the lengthy development times required to develop discrete adjoint solvers for PDEs. All of these significant advantages come at the cost of increased memory requirements for the discrete adjoint solver. However, given typical use patterns in parallel computers, this disadvantage is rather small when compared with the very significant advantages that can be realized. Sensitivities of drag and lift coefficients are validated and the overall performance of the method is shown to be comparable to conventional adjoint approaches.

Introduction

Adjoint methods for sensitivity analysis involving partial differential equations (PDE) 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 expanded by Jameson to perform airfoil shape optimization.[2] Since then, the adjoint method has been developed for more complex problems, leading to its application in the design optimization of complete aircraft configurations considering aerodynamics alone,[3,4] as well as aero-structural interactions.[5] The adjoint method has also been generalized for the case of multidisciplinary systems.[6]

The usefulness of the adjoint method lies in the fact that it is an extremely efficient approach to compute the sensitivity of one function of interest with respect to many parameters. When using gradient-based optimization algorithms, sensitivity computations are often the trouble spot, both due to computational inefficiency and inaccuracy.

Given the power of adjoint methods, it seems peculiar that their use in aerodynamics shape optimization has not become more widespread in the last two decades. In fact, while adjoint methods have already found their way into commercial structural analysis packages, they have yet to proceed beyond research 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*.[7] 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 obtained. To use the reverse mode, the solver has to run first and intermediate variables must be stored. These variables are then used by the reverse version of the code to find the sensitivities. The forward mode is virtually the analog of the finite-difference method without problems of step-size sensitivity. The reverse mode is similar to the adjoint method and also is efficient when computing the sensitivity of a function with respect to many parameters.

However, the memory requirements can be pro-

[*]Assistant Professor, AIAA Senior Member
[†]M.A.Sc. Candidate, AIAA Student Member
[‡]Associate Professor, AIAA Member

hibitive in the case of iterative solvers such as those used in CFD because they entail a large number of iterations to achieve convergence and intermediate results may need to be stored. Although efforts have been pursued to minimize the increase in memory requirements arising from iterative solvers,[8] the fact remains that given typical parallel computing resources, it is still very difficult to treat large-scale problems. The reverse mode of automatic differentiation has been applied to iterative PDE solvers by a few researchers.[9–11] As mentioned earlier, the success of this approach has been limited, mainly due to prohibitive memory requirements in 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 pre-existing code to compute the equation residuals — including boundary conditions — and the cost functions.[12] For this reason, in this paper we propose the use of automatic differentiation *only* to compute certain terms of the discrete adjoint equations that, together with standard available techniques for the iterative solution of large linear systems — such as preconditioned GMRES — can be used to carry out sensitivity analysis. The major advantages of this method are that it is:

- Almost entirely automatic: given existing code in the solver (residuals, boundary conditions, objective function computation), it creates the necessary code to compute all the necessary terms in the discrete adjoint formulation.

- Exactly consistent: because the process of automatic differentiation allows us to treat exactly arbitrarily complex expressions for the computation of the residuals, boundary conditions, and cost functions we are able to produce sensitivities 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. Previous work has shown these approximations to result in significant errors of the gradient components (sometimes errors in sign), particularly for viscous flows.

- Generic: it can be quickly applied to new formulations of the governing equations or even new governing equations themselves (such as the governing equations for magnetohydrodynamics).

While our approach does not constitute a fully automatic way of obtaining sensitivities like pure automatic differentiation, it is much faster (in execution time) and drastically reduces the memory requirements, yet it retains a large amount of automation and, in addition, it allows for the flexibility to include modifications of the basic scheme in the discrete adjoint solver. When compared to the conventional 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 greatly reduced probability of programming errors.

In the next section we review the background material that is relevant to our work, namely analytic sensitivity analysis methods and automatic differentiation. We then discuss how these were implemented in our application. In the results we establish the precision of the desired sensitivities and analyze the performance of the proposed adjoint method.

Background

Semi-Analytic Sensitivity Analysis

One of the most popular sensitivity analysis methods is finite differencing. Even though it can potentially lead to very large errors due to subtractive cancellation, it is still widely used because of its straightforward implementation.

Semi-analytic methods are much more accurate: they are usually capable of yielding derivatives with the same precision as the quantity that is being differentiated.

Our intent is to calculate 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 themselves directly, but also on the state of the system that may result from the solution of a a governing equation which may be a PDE. Thus we can write the vector-valued function to be differentiated, I, 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

$$\mathcal{R}\left(x, w\left(x\right)\right) = 0. \tag{2}$$

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

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

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 re-evaluating the function in the numerator with everything else remaining constant. The total derivatives, however, require the solution of the system's governing equations. Thus, all the terms in the total sensitivity equation (3) can be computed with little effort except for $\mathrm{d}w/\,\mathrm{d}x$.

Since the governing equations must always be satisfied, the total derivative of the residuals (2) with respect to any design variable must also be zero. Expanding the total derivative of the governing equations with respect to the design variables we can write,

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

This expression provides the means for computing the total sensitivity of the state variables with respect to the design variables. By rewriting equation (7) as

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

where we have defined the new 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 equations (3,8) can be written as,

$$\frac{\mathrm{d}I}{\mathrm{d}x} = D + GB, \tag{11}$$
$$\text{such that} \quad AB = C, \tag{12}$$

The final sensitivity can be also obtained by solving the dual of this problem.[13] The dual problem can be derived as follows. If we substitute the solution of the linear system $AB = C$ into the total sensitivity equation (3) we obtain

$$\frac{\mathrm{d}I}{\mathrm{d}x} = D + GA^{-1}C. \tag{13}$$

Defining $H = \left(GA^{-1}\right)^T$, whose size is $(n_w \times n_I)$, we can write the problem as

$$\frac{\mathrm{d}I}{\mathrm{d}x} = D + H^T C, \tag{14}$$
$$\text{such that} \quad A^T H = G^T, \tag{15}$$

where H is an $(n_x \times n_I)$ matrix.

The most computationally intensive step for both of these problems is the solution of the respective linear systems. In the case of the original problem — the *direct method* — we have to solve a linear system of n_w equations and unknowns n_x times. For the dual problem — 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.

There are two main ways of obtaining the discrete adjoint equations (15) for a given system of PDEs. The continuous approach forms a continuous adjoint problem from the governing PDEs and then discretizes this problem to solve it numerically. The discrete approach forms an adjoint from the discretized PDEs. Both of these approaches result in a system of linear equations that are different, but that in theory converge to the same result as the mesh is refined to zero size. However, the discrete approach has certain advantages in that the sensitivities are consistent with those produced by the discretized solver and that the it can treat arbitrary cost functions (which is not the case in the continuous approach). Furthermore, it is easier to obtain the appropriate boundary conditions for the adjoint solver in a discrete fashion. In this work, we adopt the discrete approach. Although the program resulting from the continuous approach can have lower memory requirements, all of the advantages mentioned earlier take precedence over the memory requirement.

CFD Adjoint Equations

We will now derive the adjoint equations for the particular case of our flow solver. The governing equations for the three-dimensional Euler equations are,

$$\frac{\partial w}{\partial t} + \frac{\partial f_i}{\partial x_i} = 0, \tag{16}$$

where x_i are the coordinates in the i^{th} direction, and the state and the fluxes for each cell are

$$w = \begin{bmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho u_3 \\ \rho E \end{bmatrix}, \qquad f_i = \begin{bmatrix} \rho u_i \\ \rho u_i u_1 + p\delta_{i1} \\ \rho u_i u_2 + p\delta_{i2} \\ \rho u_i u_3 + p\delta_{i3} \\ \rho u_i H \end{bmatrix} \tag{17}$$

It must be noted that this derivation is presented here for the Euler equations, but since our approach is only based on the existence of code the computes the *residual* of the governing equations, the procedure can be extended to the full Reynolds-averaged Navier–Stokes equations without modification. Note that the code for the residual computation is assumed to include the boundary conditions of the problem (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,

$$K_{ij} = \left[\frac{\partial X_i}{\partial \xi_j}\right], \qquad J = \det(K), \tag{18}$$

$$K_{ij}^{-1} = \left[\frac{\partial \xi_i}{\partial X_j}\right], \qquad S = JK^{-1}, \tag{19}$$

where S represents the areas of the face of the of each cell projected on to each of the physical coordinate directions.

The Euler equations in computational coordinates can then be written as,

$$\frac{\partial Jw}{\partial t} + \frac{\partial F_i}{\partial \xi_i} = 0, \tag{20}$$

where the fluxes in the computational cell faces are given by $F_i = S_{ij} f_j$.

In semi-discrete form the Euler equations can be shown to be,

$$\frac{\mathrm{d}w_{ijk}}{\mathrm{d}t} + \mathcal{R}_{ijk}(w) = 0, \tag{21}$$

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

The adjoint equations (15) can be re-written for this flow solver as,

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = \frac{\partial I}{\partial w}. \tag{22}$$

The total sensitivity (3) in this case is,

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \psi^T \frac{\partial \mathcal{R}}{\partial x}. \tag{23}$$

where ψ is the *adjoint vector.*

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 doing it manually or using finite differences. Where appropriate we will use the *reverse mode* of automatic differentiation.

Automatic Differentiation

Algorithmic differentiation — also known as computational differentiation or automatic 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.

We denote the *independent* variables as $t_1, t_2, \ldots t_n$, which are usually 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, that we are interested in.

We can then write the sequence of operations in the algorithm as

$$t_i = f_i\left(t_1, t_2, \ldots t_{i-1}\right), \quad i = n+1, n+2, \ldots, m. \tag{24}$$

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

$$\frac{\partial t_i}{\partial t_j} = \sum_{k=1}^{i-1} \frac{\partial I_i}{\partial t_k}\frac{\partial t_k}{\partial t_j}, \quad j = 1, 2, \ldots, n \tag{25}$$

Using the forward mode, we chose 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 backwards in the index j all the way to the independent variables.

The two modes are directly related to the direct and adjoint methods, respectively. The counterparts of the state variables in the semi-analytic methods are the intermediate variables, and the residuals are the lines of code that compute those quantities. The analog of the $\partial\mathcal{R}/\partial w$ matrix is then a lower-triangular with unit diagonal entries.

There are numerous software tools for automatic differentiation for different programming languages. There are two main approaches to 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 in 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 the 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.

Adifor,[14] TAF,[15] TAMC[16] and Tapenade[17,18] are some of the tools available for Fortran. Of these, only TAF and Tapenade offer support for Fortran 90, which was a requirement in our case.

Another point worth noting is that the complex-step derivative approximation[19] is equivalent to the forward-mode using operator overloading.

We chose to use Tapenade as it is the only non-commercial tool with support for Fortran 90. Tapenade is the successor of Odyssée[20] and was developed at the INRIA. It uses source transformation and can perform differentiation in both forward and reverse mode. Furthermore, the Tapenade team is actively developing their software and has been very responsive to a number of suggestions to complete their support of the Fortran 90 standard.

Implementation

To compute the desired sensitivities, we need to form the discrete adjoint equations (22), solve them and then use the total sensitivity equation (23). We will use automatic differentiation to generate code that computes the several matrices of partial sensitivities present in these equations.

For purposes of demonstration in this paper, we decided to compute the sensitivity of the drag coefficient with respect to the free stream Mach number, i.e., $I = C_D$ and $x = M_\infty$.

Computation of $\partial R/\partial w$

The flux Jacobian, $\partial R/\partial w$, is independent of the choice of function or design variable: it is simply a function of the governing equations and their discretization and boundary conditions. To compute it we need to consider the routines in our 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 within the context of the SUmb flow solver[21] that has been developed at Stanford University under the sponsorship of the Department of Energy. SUmb is a completely generic, cell-centered multiblock solver for the Reynolds-averaged Navier-Stokes equations (steady, unsteady, and time-spectral) and it has options for a variety of turbulence models with one, two and four equations. The computation of the flux Jacobian using automatic differentiation in SUmb can be summarized as follows,

1. Compute inviscid fluxes: For each of the N_c cells in the domain, compute the influence of the flow variables on the residual in that cell. For our inviscid flux discretization the only flow variables, w, that influence the residual at a cell are the flow variables at that cell and at the six cells that are adjacent to the faces of the cell.

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

3. Compute viscous fluxes (with similar stencil implications: only the cells directly surrounding the cell in question need to be considered).

4. Apply boundary conditions.

Note that to compute the residuals over the domain, three nested loops (in 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, when all contributions have been accounted for.

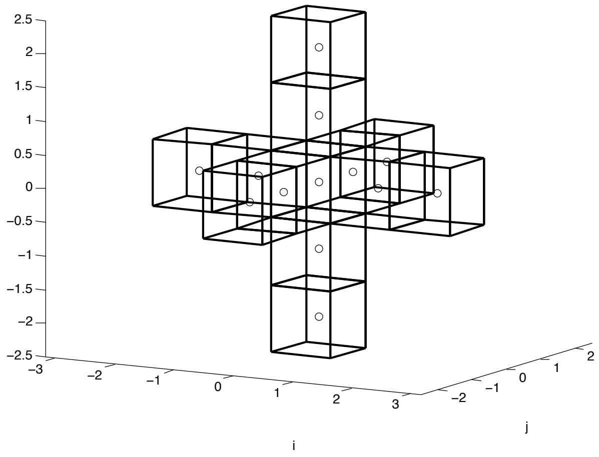

Fig. 1 Stencil for the residual computation

To better understand 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 solve, 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, where considering inviscid and dissipation fluxes, the stencil is as show in Figure 1 and $N_s = 13$.

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

Using an automatic differentiation tool on this set of routines would result in unnecessary inefficiencies. If the forward mode were used, then the cost of computing $\partial R/\partial w$ would be roughly $N_c \times N_w$ times the cost of the above residual computation.

If the reverse mode were used, then there would be a large memory penalty associated with the storage of all the intermediate variables generated by the series of nested loops, which is what we had wanted to avoid. In principle, because $\partial R/\partial w$ is a square matrix, neither mode should have an advantage over the other in terms of computational time.

However, due to the way residuals are computed, the reverse mode is much more efficient. To explain the reason for this, consider a very simplified version of a calculation resembling the residual calculation in CFD shown in Figure 2. This subroutine loops through a two-dimensional domain and computes `r` (the "residual") in the interior of that domain. The residual at any point depends only on the `ws` (the "flow variables") of that point and of the points adjacent to it, thus the stencil of dependence forms a cross with five points.

```
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 type of boundary conditions. However, this simple example is sufficient to demonstrate the computational inefficiencies of a purely automatic approach.

The forward mode of Tapenade produced the subroutine shown in Figure 3. Two new variables are introduced: `wd`, which is the seed vector and `rd`, which is the gradient of all `r`s in the direction specified by the seed vector. For example, if we want 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 the time, although Tapenade can be run in "vectorial" mode to get the whole vector of sensitivities. In this case, an additional loop inside the nested loop is required, as well as additional storage. For our purposes, the differentiated code would have to be called $N_c \times N_w$ times.

The subroutine produced by the reverse mode of Tapenade is shown in Figure 4. This case shows the additional storage requirements: since the `w`s are overwritten, the old values must be stored for later use in the reversed loop.

The overwriting of the flow variables in the first 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 functions (`PUSHREAL` and `POPREAL`) to do this. In this case, we can see that the `w`s are stored before they are modified in the forward sweep, and then retrieved in the reverse sweep.

As with the forward mode, this subroutine would have to be called $N_c \times N_w$. Therefore, to avoid the automatic differentiation of nested loops over the whole computational domain, we decided to create a set of routines that given a cell location, computes the residuals of that cell only. The routine — which is really 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 have not considered the viscous fluxes in

```
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

the present work, only the inviscid fluxes, dissipation fluxes and the boundary conditions. The stencil that affects a given cell when considering these fluxes is shown in Figure 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.

Now we 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 $\partial R/\partial w$ matrix. Each of these rows contains no more than $(N_w \times N_s)$ nonzero entries. Given the ratio of outputs to inputs it is clear that the computation of these sensitivities is performed more efficiently using the reverse mode. In our particular case, $N_w = 5$ and $N_s = 13$, so the ratio of the number of variables is 13 times larger than the number of residuals. For viscous computations the advantages of the reverse mode are even more compelling.

Tapenade was used in the reverse mode to produce adjoint code for this set of routines. In the process we discovered a few minor bugs, which we reported and have now been fixed by the authors of Tapenade. The only major current restriction in using Tapenade is that one cannot have pointer variables in the path of differentiation. This was easy to work around in our case, as we simply copied the relevant variables instead of pointing to them. We did have pointer variables that remained constant within the section of code that is differentiated, and this did not present a problem. The accuracy and computational cost of computing $\partial R/\partial w$ for our test case is presented in the results section.

Computation of $\partial C_D/\partial w$

The right-and side of the adjoint equations (22) is the vector (or matrix, in the case of multiple functions of interest) on the right hand side of the adjoint equation and represents the direct effect that the flow variables have on the function. In the cases shown here, the parameters of interest are C_D and C_L, so the vectors we are interested in are $\partial C_D/\partial w$ and $\partial C_L/\partial w$. As with the above residual equations, a modified version 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 term was split up to allow for a modular approach to computing the derivative. For example, since for the Euler equations C_D and C_L are simple integrations of the pressure over the solid surfaces of the mesh, we know *a priori* that the only changes in C_D and C_L will come through change in the pressure. As a result, the $\partial C_D/\partial w$ 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 follows:

$$\frac{\partial C_D}{\partial w} = \frac{\partial C_D}{\partial C_f}\frac{\partial C_f}{\partial p}\frac{\partial p}{\partial w}. \tag{26}$$

The first two terms in this expression are easily calculated. $\partial C_D/\partial C_f$ is simply a function of the drag direction relative to the coordinate axes of the forces, while $\partial C_f/\partial p$ can be computed via a simple integration of the pressures over the surface of the solid. For our case each of these terms were expressed as a single function and differentiated in forward mode using Tapenade. It is worthy of note that for these two functions in particular, the reverse mode would be more advantageous than the forward mode. Both functions have significantly more input variables than output variables: for $\partial C_D/\partial C_f$ the ratio is 6:1, while for $\partial C_f/\partial p$ the ratio is $(N_x+4)(N_y+4)(N_z+4):6$. However, because both functions are relatively simple, it was not considered to be a priority to complete the reverse mode differentiation. Note: because of the way the pressure was calculated in SUmb, 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. As discussed in the section on $\partial R/\partial w$ above,the computations were done on a stencil by stencil basis. Thus, in this case, for each state all of the derivatives in the stencil can be computed at once. Further, because of the definition of the stencil, any cells outside the stencil will have a derivative of zero relative 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. Perturb the desired state: for each of the $N_w \times N_c$ states in the problem, the seed vector must be set to one. 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 will not be affected by this state.

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

With these three terms calculated, we can compute $\partial C_D/\partial w$ and proceed with the solution of the adjoint equations.

Adjoint Solver

The adjoint equations (22) can be re-written for the flow solver as,

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial C_D}{\partial w}. \tag{27}$$

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).[22–24] 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 built in functions, each of the components in the adjoint equations were setup in sparse format, $\partial R/\partial w$ in sparse matrix format and $-\partial C_D/\partial w$ in sparse vector format. Once the sparse matricies were setup, PETSc's built in Krylov solver was used to compute the adjoint solution. In general, this approach proved to be successful.

Total Sensitivity Equation

The total sensitivity (3) in this case can be written as,

$$\frac{\mathrm{d}C_D}{\mathrm{d}M_\infty} = \frac{\partial C_D}{\partial M_\infty} + \psi^T \frac{\partial \mathcal{R}}{\partial M_\infty}, \tag{28}$$

where we have chosen the free stream Mach number, M_∞, to be the independent variable. Thus, with the adjoint (ψ) computed, there are only two remaining terms required to form the total sensitivity equation, $\partial C_D/\partial M_\infty$ and $\partial \mathcal{R}/\partial M_\infty$. The former term is a very simple dependence. The only direct impact of M_∞ on C_D is through the non-dimensionalization. Thus, this term can be computed analytically. However, in the spirit of this development, the function used for $\partial C_f/\partial p$ earlier was differentiated again, this time with respect to M_∞. This function provided the sensitivity $\partial C_f/\partial M_\infty$, which was then multiplied with $\partial C_D/\partial C_f$ from earlier to achieve the required sensitivity. The differentiation of this function 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, $\partial R/\partial w$, that were previously discussed. Thus much of the same logic regarding the use of a single cell residual calculation still applies. However, in this case, the result is a vector not a matrix and only has one input. Therefore, for this computation, the 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. The term was expressed as follows:

$$\frac{\partial \mathcal{R}}{\partial M_\infty} = \frac{\partial \mathcal{R}}{\partial w_\infty}\frac{\partial w_\infty}{\partial M_\infty} \tag{29}$$

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

1. Set w_∞ seed value: For each of the five values of w_∞ set the seed and compute the dependence of each cell residual.

2. Compute inviscid fluxes: For each of the N_c cells in the domain, compute the influence of the flow variables on the residual in that cell. For our inviscid flux discretization the only flow variables, w, that influence the residual at a cell are the flow variables at that cell and at the six cells that are adjacent to the faces of the cell.

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

4. Compute viscous fluxes (with similar stencil implications: only the cells directly surrounding the cell in question need to be considered).

5. Apply boundary conditions: For each boundary cell compute the effect of the boundary condition on the states and residuals. It is in this computation that w_∞ affects the residuals.

The last term, $\partial w_\infty/\partial M_\infty$, is very straight forward 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 theses derivatives using Tapenade, a new function, which combined all of these dependencies, was created and then differentiated in forward mode. Again, the number of output variables far exceeds the number of input variables (by 5:1).

The results from these computations are presented in the following section.

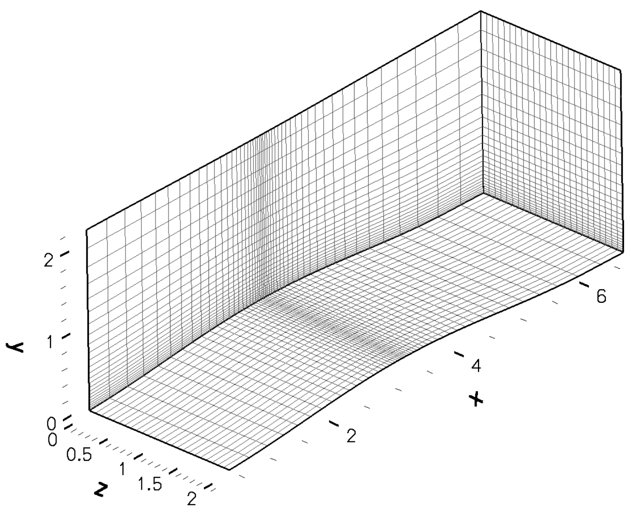

Fig. 5 Computational domain mesh

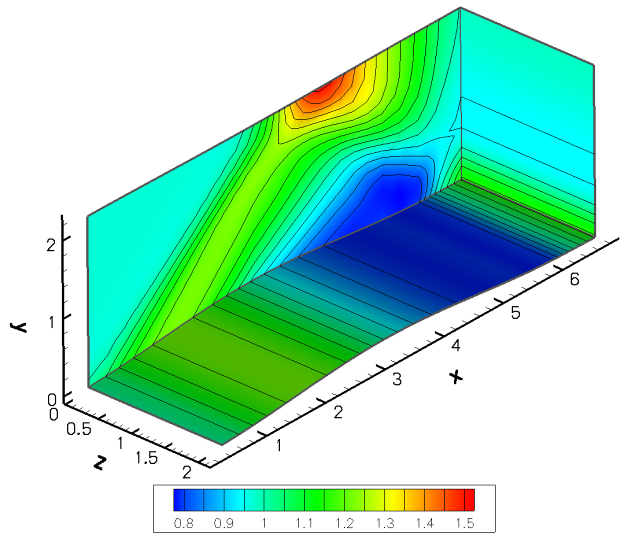

Fig. 6 Contour plot of density

Results

Description of Test Case

The test case we have used to demonstrate our method is a simple bump in a channel flow where the front and back walls of the channel have symmetry boundary conditions imposed on them. The top wall is flat while the bottom wall has been deformed with a sinusoidal bump to create some reasonable variations of the flow. The inflow and outflow faces of the domain have non-reflecting boundary conditions imposed on them. The upper and lower walls use a normal momentum boundary condition to show how easy it is to treat complex boundary conditions with the approach described in this paper.

The mesh for the test case is shown in Figure 5. It is a relatively small mesh ($48 \times 24 \times 24$) and is being used as a proof of concept. Figure 6 shows the density contours for the flow solution. The free stream Mach number is 2.

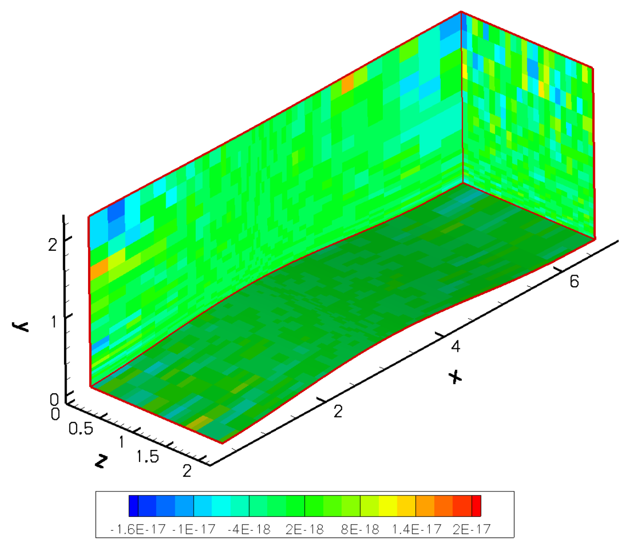

Fig. 7 Relative errors of the residual computations

Flux Jacobian

As mentioned in the implementation section, to reduce the cost of the automatic differentiation procedure, we wrote a set of subroutines that given a cell index computes the residuals of that cell only. This set of subroutines closely resembles the original code used to compute the residuals over the whole domain, except in that it does not loop over the domain. It simply computes the residual at the selected cell. 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 "cut-and-paste", 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 difference between the new and original residuals is shown in Figure 7. The differences are all $\mathcal{O}(10^{-17})$ (less than machine zero) and thus the new calculation is correct.

We differentiated the cell residual calculation routines with Tapenade using the reverse mode, for the reasons we previously mentioned. The residuals were specified as the outputs and the flow variables as the inputs. 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, i.e., all the nonzero terms in the corresponding row of $\partial R/\partial w$. To obtain the full flux Jacobian, a loop over residual index 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 Figure 8. The errors are within an acceptable range, varying between $\mathcal{O}(10^{-10})$ and $\mathcal{O}(10^{-6})$. The finite differences are most likely less precise than the automatic differentiation results, but there is no way of verifying this claim without analytic results. The pattern of the sparsity of the computed flux Jacobian is

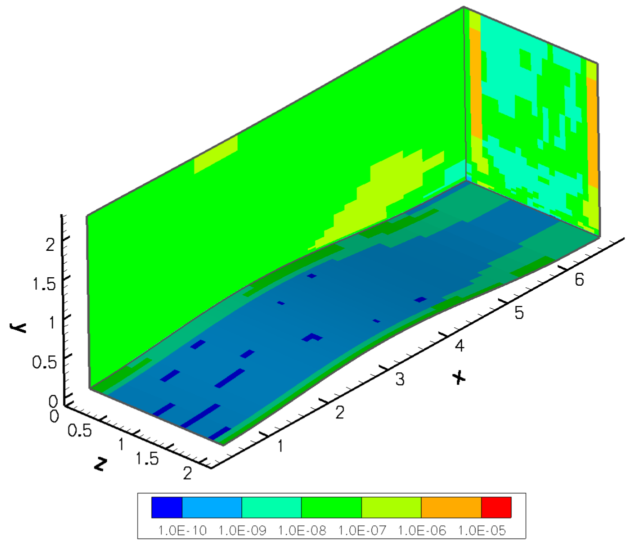

Fig. 8 Relative errors in the sum of each row of $\partial R/\partial w$

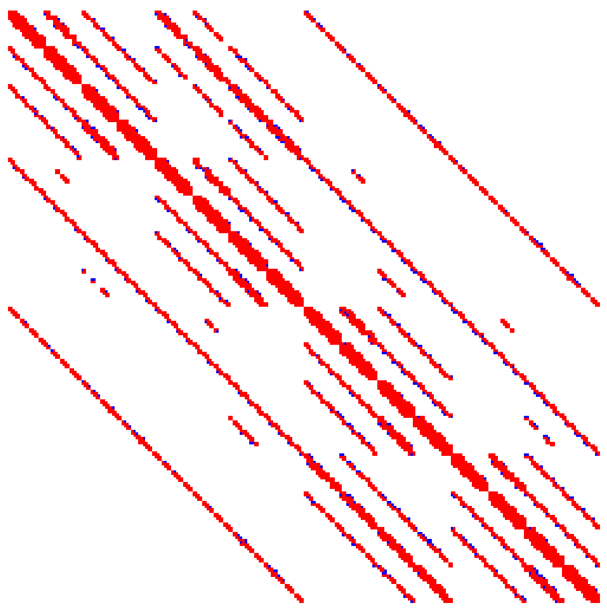

Fig. 9 Sparsity pattern of the flux Jacobian

shown in Figure 9.

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 speculated in the previous section. The reason is that, as explained before, the reverse mode

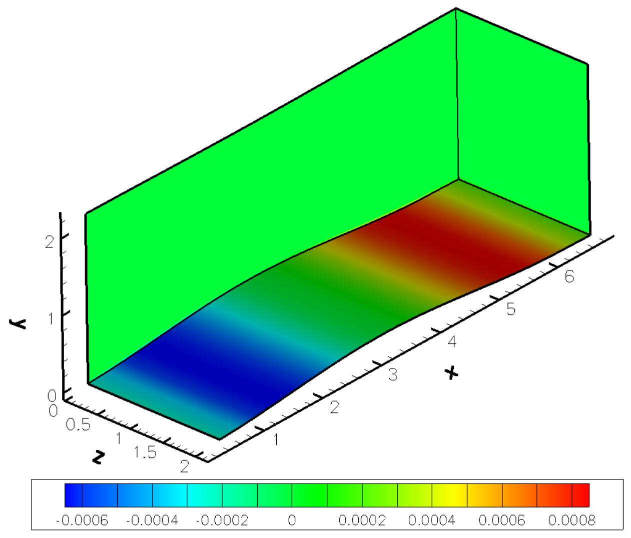

Fig. 10 Partial sensitivity of C_D with respect to density

calculation does not need to be called as many times as the finite-difference residual evaluations.

Table 1 Times and error for flux Jacobian computation

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

Other Partial Derivative Terms

In the process of developing our adjoint solver we visualized the various partial derivative terms that appear in the adjoint and total sensitivity equations. Figure 10 shows the sensitivity of drag coefficient to the density at each cell. The only non-zero terms are located in the two slabs of cells adjacent to the wall ($j = 1, 2$) with the bump, which is what we expected.

The adjoint variables corresponding to the continuity equation are shown in Figure 11. 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 $\partial R/\partial M_\infty$, which appears in the total sensitivity equation (28), is shown in Figure 12. The figure only shows the sensitivity corresponding to the residual of the continuity equation. Since the flow is supersonic, M_∞ affects only the cells near the inflow plane. Three slabs of cells ($i = 1, 2, 3$) are affected by the free stream conditions.

Lift and Drag Coefficient Sensitivities

The benchmark sensitivity results were obtained using the complex-step derivative approximation,[19] which can be considered to be numerically exact, that is, the precision of the sensitivity is of the same order

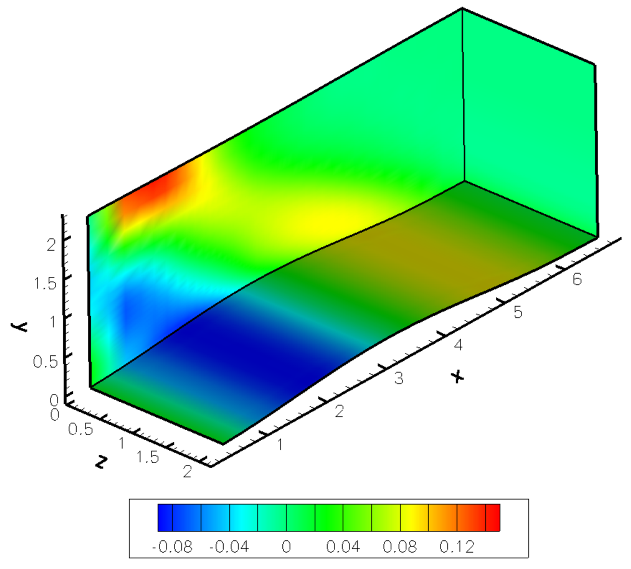

Fig. 11 Adjoint of the continuity equation

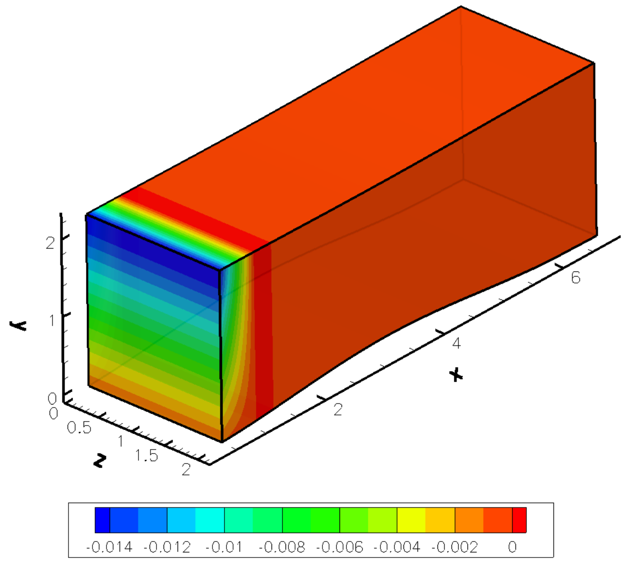

Fig. 12 Partial sensitivity of the continuity residual with respect to M_∞

as the precision of the solution. The derivative in this case is given by,

$$\frac{\mathrm{d}C_D}{\mathrm{d}M_\infty} = \frac{\operatorname{Im}\left[C_D(M_\infty + ih)\right]}{h}, \qquad (30)$$

where h represents the magnitude of the complex step. A value of $h = 10^{-20}$ was used. The sensitivities given by the complex step were spot checked against finite differences by doing a step size study.

In Table 2 we show the sensitivites of both drag and lift coefficients with respect to freestream Mach number for different cases. We consider two types of wall boundary treatment: a simple linear pressure extrapolation (LPE) and a normal momentum (NM) boundary condition. The inflow direction was also varied.

The fine mesh case is the one shown in Figure 5, and the coarse case is a very small mesh ($12 \times 4 \times 4$) of the same geometry that was used for debugging purposes.

We can see that the adjoint sensitivities for the bump cases are extremely accurate, yielding five to nine digits agreement when compared to the complex-step results. This is consistent with the convergence tolerance that was specified in PETSc for the adjoint solution.

In addition to the various bump cases, we also ran the wing shown in Figure 13. This is an NLR LANN wing flown at $M_\infty = 0.621$ and 0.59 degrees of angle of attack. Th sensitivities show and agreement of only three digits. The reason for this discrepancy is the fact that our implementation of the residual calculation that is automatically differentiated does not properly account for the trailing edge cut.

To analyze the performance of the ADjoint solver, several timings were performed. They are shown in Table 3 for two different cases: a very coarse grid with 5,120 flow variables and a finer grid with 203,840 flow variables.

Table 3 ADjoint computational cost breakdown (times in seconds)

	Coarse	Fine
Flow solution	1.559	219.215
ADjoint	0.572	51.959
Breakdown:		
Setup PETSc Variables	0.004	0.011
Compute flux Jacobian	0.218	11.695
Compute RHS	0.124	8.487
Solve the adjoint equations	0.166	28.756
Compute the total sensitivity	0.059	3.010

The total cost of the adjoint solver, including the computation of all the partial derivatives and the solution of the adjoint system is less than one fourth of the flow solution. This is even better than what is usually observed in the case of adjoint solvers developed by conventional means, showing that the ADjoint approach is very efficient.

When it comes to comparing the performance of the various components in the adjoint solver, the fine grid case is more representative, since the coarse grid runs so fast that the timings are much less reliable. We found that most of the time was spent in the solution of the adjoint equations and thus all the automatic differentiation sections performed very well. The costliest of the automatic differentiation routines

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

Mesh	Coefficient	Inflow direction	Wall BC	ADjoint	Complex step
Coarse	C_D	(1,0,0)	LPE	-0.28989664*551*	-0.28989664437
	C_L			-0.267704*80557*	-0.26770446176
Coarse	C_D	(1,0,0)	NM	-0.381246*62362*	-0.38124658590
	C_L			-0.23429*486336*	-0.23429516285
Fine	C_D	(1,0,0)	LPE	-2.79501183*66*	-2.7950118341
	C_L			0.58128604*322*	0.58128604666
Fine	C_D	(1,0,0)	NM	-2.993*9947517*	-2.9937985863
	C_L			0.44*894139444*	0.44904829281
Coarse	C_D	(1,0.05,0)	LPE	-0.2789076*5124*	-0.27890764625
	C_L			-0.262086*66992*	-0.26208631505
Fine	C_D	(1,0.05,0)	LPE	-0.06155986*3813*	-0.061559862874
	C_L			-0.36479675*995*	-0.36479675268
Wing	C_D	(1, 0.0102,0)	LPE	0.00104*33416311*	0.0010419154096
	C_L			-0.272*58951595*	-0.27262420067

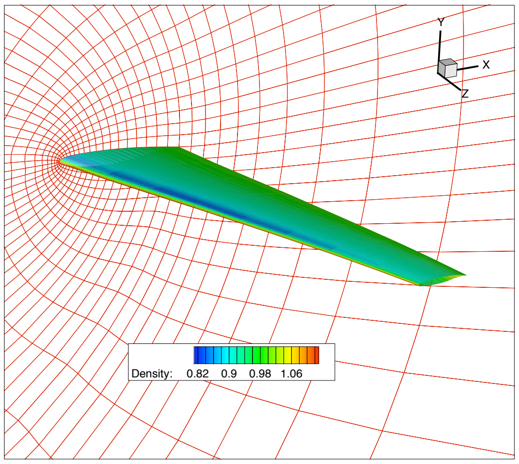

Fig. 13 Mesh and solution for the NLR LANN wing

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 quite impressive.

The computation of the right hand side ($\partial C_D/\partial w$) is not much cheaper than computing the flux Jacobian. We expect to improve the computation of this term significantly, as we have not optimized it yet.

Conclusions

In this paper we have presented an approach to construct discrete adjoint solvers for arbitrary governing equations. The implementation of the ADjoint approach was largely automatic, since no hand differentiation or calculation of adjoint boundary conditions was necessary.

The approach has the advantage that it uses the reverse mode of differentiation on the code that computes the residuals for the governing equations and it is therefore efficient. The timings presented show that the ADjoint approach is much faster than pure automatic differentiation, and even "manual adjoint" solvers, even though our code is at the prototyping stage and has therefore not been fully optimized.

The agreement between the total sensitivities computed with our adjoint implementation and the benchmark results was very good: a five to nine digit accuracy.

We believe that the penalty in storage that this methods incurs is outweighed by the substantial benefits over current methodologies to develop adjoint solvers: currently only highly specialized research groups have been able to develop such adjoint codes. In our opinion, this approach is a much more likely way in which the power of adjoint techniques will be applied to future problems in engineering design.

Acknowledgments

We would like to thank André Marta, Ki Hwan Lee and Edwin van der Weide for their assistance. The first two authors are grateful for the funding provided by the Canada Research Chairs program and the Natural Sciences and Engineering Research Council.

References

[1]Pironneau, O., "On optimum design in fluid mechanics," *Journal of Fluid Mechanics*, Vol. 64, 1974, pp. 97–110.

[2]Jameson, A., "Aerodynamic Design via Control Theory," *Journal of Scientific Computing*, Vol. 3, No. 3, sep 1988, pp. 233–260.

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

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

[5]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.

[6]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.

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

[8]Fagan, M. and Carle, A., "Reducing reverse-mode memory requirements by using profile-driven checkpointing," *Future Generation Comp. Syst.*, Vol. 21, No. 8, 2005, pp. 1380–1390.

[9]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, No. 6-7, 2005, pp. 939–945.

[10]Giering, R., Kaminski, T., and Slawig, T., "Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil." *Future Generation Comp. Syst.*, Vol. 21, No. 8, 2005, pp. 1345–1355.

[11]Heimbach, P., Hill, C., and Giering, R., "An efficient exact adjoint of the parallel MIT General Circulation Model, generated via automatic differentiation." *Future Generation Comp. Syst.*, Vol. 21, No. 8, 2005, pp. 1356–1371.

[12]Martins, J. R. R. A., Alonso, J. J., and van der Weide, E., "An Automated Approach for Developing Discrete Adjoint Solvers," *Proceedings of the 2nd AIAA Multidisciplinary Design Optimization Specialist Conference*, Newport, RI, 2006, AIAA 2006-1608.

[13]Giles, M. B. and Pierce, N. A., "An Introduction to the Adjoint Approach to Design," *Flow, Turbulence and Combustion*, Vol. 65, 2000, pp. 393–415.

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

[15]Giering, R. and Kaminski, T., "Applying TAF to generate efficient derivative code of Fortran 77-95 programs," *Proceedings of GAMM 2002, Augsburg, Germany*, 2002.

[16]Gockenbach, M. S., "Understanding code generated by TAMC," IAAA Paper TR00-29, Department of Computational and Applied Mathematics, Rice University, Texas, USA, 2000.

[17]Hascoët, L. and Pascual, V., "TAPENADE 2.1 User's Guide," Technical report 300, INRIA, 2004.

[18]Pascual, V. and Hascoët, L., "Extension of TAPENADE towards Fortran 95," *Automatic 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, 2005.

[19]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.

[20]Faure, C. and Papegay, Y., *Odyssée Version 1.6. The language reference manual*, INRIA, 1997, Rapport Technique 211.

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

[22]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.

[23]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," Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[24]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 Scientific Computing*, edited by E. Arge, A. M. Bruaset, and H. P. Langtangen, Birkhäuser Press, 1997, pp. 163–202.