Journal of Aircraft
Vol. 45, No. 1, January–February 2008

# Development of the Discrete Adjoint for a Three-Dimensional Unstructured Euler Solver

Giampietro Carpentieri,* Barry Koren,† and Michel J. L. van Tooren‡
*Delft University of Technology,*
*2629 HS Delft, The Netherlands*

DOI: 10.2514/1.32871

**The discrete adjoint of a reconstruction-based unstructured finite volume formulation for the Euler equations is derived and implemented. The matrix-vector products required to solve the adjoint equations are computed on-the-fly by means of an efficient two-pass assembly. The adjoint equations are solved with the same solution scheme adopted for the flow equations. The scheme is modified to efficiently account for the simultaneous solution of several adjoint equations. The implementation is demonstrated on wing and wing–body configurations.**

## Nomenclature

| | | |
|---|---|---|
| $\mathbf{A}$ | = | inviscid flux Jacobian |
| $\mathbf{D}$ | = | matrix of control volumes |
| $\mathbf{D}_M$ | = | diagonal of the matrix |
| $E$ | = | number of edges in the mesh |
| $e_t$ | = | total specific energy |
| $\mathbf{F}$ | = | inviscid flux |
| $\mathbf{G}$ | = | gradients vector |
| $\mathbf{H}$ | = | vector of numerical fluxes |
| $J$ | = | functional |
| $\mathbf{L}_M$ | = | strictly lower part of the matrix |
| $\mathbf{M}$ | = | linear system matrix |
| $N$ | = | number of nodes in the mesh |
| $\mathbf{n}$ | = | normal vector |
| $\mathbf{P}_M$ | = | preconditioner |
| $p$ | = | pressure |
| $\mathbf{R}$ | = | residuals vector |
| $\mathbf{r}$ | = | residual |
| $\mathbf{U}$ | = | conservative variables vector |
| $\mathbf{U}_L$, $\mathbf{U}_R$ | = | vectors of left and right states |
| $\mathbf{U}_M$ | = | strictly upper part of the matrix |
| $\mathbf{u}$ | = | conservative variables |
| $V$ | = | control volume |
| $\mathbf{v}$ | = | primitive variables |
| $\mathbf{w}$ | = | velocity |
| $\alpha$ | = | parameter |
| $\gamma$ | = | ratio of specific heats, $\gamma = 1.4$ |
| $\theta$ | = | angle of attack |
| $\mathbf{\Lambda}_J$ | = | adjoint variables vector |
| $\rho$ | = | density |
| $\sigma$ | = | slope limiter |
| $\mathbf{\Phi}$ | = | numerical flux |

*Subscripts*

| | | |
|---|---|---|
| bc | = | boundary condition |
| $i, j, k$ | = | node number |
| $w$ | = | wall |
| $\hat{}$ | = | linear reconstruction |
| $\sim$ | = | approximation |
| $\infty$ | = | freestream |

## I. Introduction

GRADIENT-BASED aerodynamic shape optimization often involves a large number of design variables and a limited number of functionals (e.g., lift, drag, or pitching moment), the evaluation of which requires expensive numerical solution procedures. Therefore, to perform the optimization at reasonable cost, it is crucial to compute the gradient efficiently.

Given the large number of design variables, methods that compute the gradient at a cost proportional to this number (e.g., finite differences, complex variables, or linearized approaches) are clearly inefficient. The adjoint method computes the gradient at an effort that is independent of the number of design variables and proportional to the number of aerodynamic functionals [1]. Thus, for aerodynamic shape optimization problems, the method becomes very attractive.

The discrete adjoint approach is widespread [2–8]. It consists of developing the adjoint on top of the discretized flow equations. In practice, the code must be differentiated to derive the Jacobians. The latter have to be transposed and multiplied by vectors. Preferably, for memory reasons, the matrix-vector products should be performed on-the-fly, avoiding matrix storage. These operations are not easily hand-coded, especially not in the case of an unstructured finite volume solver. Among several difficulties, the transposition is a major one because it inverts the operations in the differentiated code in a counterintuitive way.

Some researchers proposed automatic differentiation (AD) as an alternative to hand-coding [2,4,6]. AD tools relieve the developer of the aforementioned difficulties. However, this occurs at the price of a lower control on the code. The risk of generating inefficient code is present, in terms of both memory and execution time. In contrast, the more demanding hand-coding approach allows the developer to exploit his/her knowledge of the implementation to the maximum extent; thus, he/she can generate very efficient adjoint code [5,7,8].

The solution process for the adjoint equations also poses some difficulties. The linearity of the adjoint equations suggests the use of a linear system solver. However, the poor diagonal dominance of the matrix makes it unlikely for the procedure to succeed. A robust solution method is obtained by simply treating the adjoint problem as a nonlinear one and using the same solution procedure as for the flow equations [4,5].

Presented as Paper 3954 at the 18th AIAA Computational Fluid Dynamics Conference, Miami, FL, 25–28 June 2007; received 17 June 2007; revision received 27 August 2007; accepted for publication 1 September 2007. Copyright © 2007 by Giampietro Carpentieri. Published by the American Institute of Aeronautics and Astronautics, Inc., with permission. Copies of this paper may be made for personal or internal use, on condition that the copier pay the $10.00 per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923; include the code 0021-8669/08 $10.00 in correspondence with the CCC.

*Ph.D. Student, Faculty of Aerospace Engineering; G.Carpentieri@tudelft.nl. Member AIAA.

†Full Professor, Faculty of Aerospace Engineering; B.Koren@tudelft.nl. Senior Member AIAA.

‡Full Professor, Faculty of Aerospace Engineering; M.J.L.vanTooren@tudelft.nl. Member AIAA.

The present work is aimed at developing an aerodynamic shape optimization framework, which includes an unstructured Euler solver. A 2D version of the framework was completed and demonstrated [9]. There, several optimization test cases involving transonic and supersonic flows were successfully addressed. Here, the 3D flow and adjoint solvers are presented, which are, in part, the extension of the 2D solvers described previously [9].

The adjoint presented here is hand-coded and is efficiently assembled by means of a two-pass construction. The assembly is matrix-free and does not require any data structure in addition to that already available from the flow solver. Some simplifying approximations in the differentiation were incorporated in the implementation to limit the amount of differentiated code required. Several parts of the differentiated code are already available from the implicit scheme of the flow solver. These approximations were introduced only after numerical experiments have shown that the approximate code is as effective as the exactly differentiated code in terms of optimization results [9].

The adjoint equations are solved with an implicit scheme that uses a symmetric Gauss–Seidel solution procedure to solve the linear system of equations arising at each time step. Because shape optimization often requires some constraints to be enforced on the aerodynamic functionals, it is required to solve more than one adjoint equation. The equations have the same Jacobian matrix in common. For this reason, it appeared convenient to modify the solution method to efficiently solve multiple adjoint equations simultaneously, saving time on the computation of the expensive matrix elements. The benefit obtained by the simultaneous solution varies according to the type of storage adopted for the solution of the linear system. In fact, the symmetric Gauss–Seidel preconditioning procedure can also be performed without storing the matrix elements (i.e., matrix-free preconditioning).

The implementation is demonstrated on two configurations: the ONERA-M6 wing and the DLR-F6 wing–body.

## II. Finite Volume Formulation

The integral form of the Euler equations written for a volume $V$ contained in a domain $\Omega$ reads

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_V \mathbf{u}\,\mathrm{d}\Omega + \oint_{\partial V} \mathbf{F}(\mathbf{u})\cdot\mathbf{n}\,\mathrm{d}\Gamma = \mathbf{0} \tag{1}$$

where the conservative variables and the inviscid flux are defined, respectively, as

$$\mathbf{u} = \begin{bmatrix} \rho \\ \rho\mathbf{w} \\ \rho e_t \end{bmatrix}, \qquad \mathbf{F}(\mathbf{u}) = \begin{bmatrix} \rho\mathbf{w}^T \\ \rho\mathbf{w}\mathbf{w}^T + p\mathbf{I} \\ (p+\rho e_t)\mathbf{w}^T \end{bmatrix} \tag{2}$$

The perfect-gas equation $p = (\gamma - 1)\rho(e_t - \mathbf{w}\cdot\mathbf{w}/2)$ provides closure of the system. Equation (1) is discretized using a finite volume method on unstructured median-dual meshes. The median-dual mesh is obtained by processing the original mesh [10,11]. Each element of the latter is divided into a number of parts that are equal to the number of its vertices. Specifically, the element is divided by surfaces that are constructed by the union of the lines connecting the element's center of gravity to its edge midpoints and to the centers of its faces. Each part is then added to the corresponding node to create, once all the elements that share the node are visited, a control volume around the node itself.

For the $i$th control volume, the discretization of Eq. (1) reads

$$V_i\frac{\mathrm{d}\mathbf{u}_i}{\mathrm{d}t} + \sum_{j=1,N_i}\boldsymbol{\Phi}(\hat{\mathbf{u}}_i,\hat{\mathbf{u}}_j,\mathbf{n}_{ij}) + \boldsymbol{\Phi}^{\mathrm{bc}}(\mathbf{u}_i,\mathbf{n}_i) = \mathbf{0} \tag{3}$$

where $N_i$ is the number of distance-1 neighbors of node $i$; the numerical fluxes $\boldsymbol{\Phi}$ and $\boldsymbol{\Phi}^{\mathrm{bc}}$ replace the integrated inviscid flux; $\mathbf{n}_{ij}$ is the integrated normal along the edge that connects the nodes $i$ and $j$, for example,

$$\mathbf{n}_{ij} \equiv \int_{\partial V_{ij}} \mathbf{n}\,\mathrm{d}\Gamma$$

and $\mathbf{n}_i$ indicates the integrated normal for the boundary node $i$.

The edges intersect the interfaces between neighboring control volumes of the median-dual mesh. The intersections are located exactly at the midpoint of each edge. An edge-based data structure is used to loop on the edges of the mesh to collect the numerical fluxes.

As the hat on the conservative variables in Eq. (3) suggests, the fluxes are not evaluated with the averaged variables. They are evaluated with the variables obtained by a MUSCL-like reconstruction procedure [10], which aims at achieving second-order accuracy. The procedure is applied to the primitive variables, which are reconstructed on each edge midpoint. Across the edge $ij$, the reconstruction reads

$$\hat{v}_i = v_i + \frac{\sigma_i}{2}\nabla v_i^T\Delta\mathbf{x}_{ij}, \qquad \hat{v}_j = v_j - \frac{\sigma_j}{2}\nabla v_j^T\Delta\mathbf{x}_{ij} \tag{4}$$

where $\Delta\mathbf{x}_{ij}$ represents the distance between the coordinates of the two nodes, $\sigma$ is the slope limiter of Venkatakrishnan [12], and $\nabla v$ is the gradient of the variable. The latter may be computed using a Green–Gauss or a least-squares formulation [10].

The numerical flux $\boldsymbol{\Phi}$ in Eq. (3) is evaluated using Roe's approximate Riemann solver [13]

$$\begin{aligned}\boldsymbol{\Phi}_{ij} &= \boldsymbol{\Phi}(\mathbf{u}_i,\mathbf{u}_j,\mathbf{n}_{ij}) \\ &= \tfrac{1}{2}[\mathbf{F}(\mathbf{u}_i)+\mathbf{F}(\mathbf{u}_j)]\cdot\mathbf{n}_{ij} - \tfrac{1}{2}\left|\mathbf{A}\left(\mathbf{u}_{ij}^r,\mathbf{n}_{ij}\right)\right|(\mathbf{u}_j-\mathbf{u}_i)\end{aligned} \tag{5}$$

which uses the Roe averages $\mathbf{u}^r$ to compute the absolute value of the inviscid flux Jacobian $\mathbf{A} = \mathrm{d}(\mathbf{F}\cdot\mathbf{n})/\mathrm{d}\mathbf{u}$. The numerical flux $\boldsymbol{\Phi}^{\mathrm{bc}}$ in Eq. (3), which is obviously present only when the node $i$ is lying on the boundary, depends on whether the boundary type is external, wall, or symmetry. In the case of an external boundary, flux vector splitting [14] is used, which evaluates the numerical flux as

$$\boldsymbol{\Phi}_i^{\mathrm{bc}} = \boldsymbol{\Phi}^{\infty}(\mathbf{u}_i,\mathbf{n}_i) = \mathbf{A}^{-}(\mathbf{u}_i,\mathbf{n}_i)\mathbf{u}_{\infty} + \mathbf{A}^{+}(\mathbf{u}_i,\mathbf{n}_i)\mathbf{u}_i \tag{6}$$

with $\mathbf{A}^{+} = (\mathbf{A}+|\mathbf{A}|)/2$ and $\mathbf{A}^{-} = (\mathbf{A}-|\mathbf{A}|)/2$ being the plus and the minus splitting of the flux Jacobian, respectively. For a wall boundary, the vanishing of the normal velocity, $\mathbf{w}\cdot\mathbf{n} = \mathbf{0}$, is enforced in the normal projection of the inviscid flux. Therefore, one has

$$\boldsymbol{\Phi}_i^{\mathrm{bc}} = \boldsymbol{\Phi}^{w}(\mathbf{u}_i,\mathbf{n}_i) = [0, p\mathbf{n}_i, 0]^T \tag{7}$$

This flux is also used for the symmetry type of boundaries.

The residual of each node is defined as the sum of the numerical fluxes; that is,

$$\mathbf{r}_i \equiv \sum_{i=1}^{N_i}\hat{\boldsymbol{\Phi}}_{ij} + \boldsymbol{\Phi}_i^{\mathrm{bc}}$$

If one introduces the conservative variables vector $\mathbf{U} = [\mathbf{u}_1,\mathbf{u}_2,\ldots,\mathbf{u}_N]^T$ and the residual vector $\mathbf{R} = [\mathbf{r}_1,\mathbf{r}_2,\ldots,\mathbf{r}_N]^T$, it is then possible to write Eq. (3) in the more compact form:

$$\mathbf{D}\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t} + \mathbf{R} = \mathbf{0} \tag{8}$$

where $\mathbf{D}$ is a diagonal matrix containing the control volumes.

## III. Discrete Adjoint for the MUSCL Scheme

The discrete adjoint method [15] computes the sensitivity of a functional $J$ with respect to a generic parameter $\alpha$ as

$$\frac{\mathrm{d}J}{\mathrm{d}\alpha} = \frac{\partial J}{\partial\alpha} - \boldsymbol{\Lambda}_J^T\frac{\partial\mathbf{R}}{\partial\alpha} \tag{9}$$

where the vector $\boldsymbol{\Lambda}_J$ contains the adjoint variables, which are found by the solution of the adjoint equation:

$$\frac{\partial \mathbf{R}^T}{\partial \mathbf{U}} \boldsymbol{\Lambda}_J = \frac{\partial J^T}{\partial \mathbf{U}} \tag{10}$$

The practical implementation of the transposed Jacobian represents a major difficulty of the method, especially if one is interested directly in the computation of the matrix-vector products $[\partial \mathbf{R}/\partial \mathbf{U}]^T \boldsymbol{\Lambda}_J$, which is the case in the present work. At least for 3D applications, the computation of the matrix only, with subsequent transposition, may be too demanding in terms of memory.

### A. Exact and Approximate Discrete Adjoint

An exact discrete adjoint for the 2D Euler equations has been implemented previously [9]. The terms appearing in the adjoint equation were derived exactly; that is, no approximations were made in the differentiation. Exactness was demonstrated indirectly, employing the differentiated Jacobian in Newton iterations. The iterations converged quadratically. This is known to be possible only when the Jacobian is exact.

However, the exact differentiation of the code requires a lot of human work. The question arose whether all the different contributions to the differentiation were necessary to obtain an effective code. For this reason, an investigation was carried out [9] to assess the effect of simplifying approximations. Rather than looking at the error generated in the sensitivity [see Eq. (9)], the effect was considered directly on the results obtained for some optimization test cases. The cases involved subsonic as well as supersonic flows and were driven by two different optimization algorithms. From the optimization results, it appeared that some approximations produce effective code. In fact, results obtained using these approximations in the code were comparable with those of the exact adjoint code. Similar results have been found by other authors [16].

The approximations were incorporated in the 3D adjoint code presented here to ease the development phase and to produce the simplest possible code. They consist of neglecting the Jacobian in the differentiation of the numerical fluxes of Eqs. (5) and (6); that is,

$$\begin{aligned}\frac{\partial \boldsymbol{\Phi}_{ij}}{\partial \mathbf{u}_i} &\approx \frac{1}{2}(\mathbf{A}(\mathbf{u}_i, \mathbf{n}_{ij}) + |\mathbf{A}(\mathbf{u}_{ij}^r, \mathbf{n}_{ij})|) \\ \frac{\partial \boldsymbol{\Phi}_{ij}}{\partial \mathbf{u}_j} &\approx \frac{1}{2}(\mathbf{A}(\mathbf{u}_j, \mathbf{n}_{ij}) - |\mathbf{A}(\mathbf{u}_{ij}^r, \mathbf{n}_{ij})|)\end{aligned} \tag{11}$$

for the Roe flux and

$$\frac{\partial \boldsymbol{\Phi}_i^\infty}{\partial \mathbf{u}_i} \approx \mathbf{A}^+(\mathbf{u}_i, \mathbf{n}_i) \tag{12}$$

for the numerical boundary flux. Also, the limiter in the reconstruction of Eq. (4) is considered as constant in the differentiation (i.e., $\partial \hat{v}_i/\partial \sigma_i = 0$).

### B. Derivation of the Discrete Adjoint

To derive a convenient expression for the (transposed) Jacobian, it is useful to introduce some vectors: the vector of second-order fluxes $\mathbf{H} = [\hat{\boldsymbol{\Phi}}_1, \hat{\boldsymbol{\Phi}}_2, \ldots, \hat{\boldsymbol{\Phi}}_E]^T$; the vector of reconstructed left states $\mathbf{U}_L = [\hat{\mathbf{u}}_{L1}, \hat{\mathbf{u}}_{L2}, \ldots, \hat{\mathbf{u}}_{LE}]^T$; the vector of reconstructed right states $\mathbf{U}_R = [\hat{\mathbf{u}}_{R1}, \hat{\mathbf{u}}_{R2}, \ldots, \hat{\mathbf{u}}_{RE}]^T$; and the vector of primitive variables gradient $\mathbf{G} = [\nabla \mathbf{v}_1, \nabla \mathbf{v}_2, \ldots, \nabla \mathbf{v}_N]^T$. The first three vectors have length $E$ and the gradient has length $N$, equal to the number of edges and nodes in the mesh, respectively.

For the MUSCL-scheme described in the previous section, the dependency of the residual vector upon the conservative variables can be indicated in abstract form as

$$\mathbf{R} = \mathbf{R}[\mathbf{H}(\mathbf{U}_L[\mathbf{U}, \mathbf{G}(\mathbf{U})], \mathbf{U}_R[\mathbf{U}, \mathbf{G}(\mathbf{U})])] \tag{13}$$

where the dependence of the reconstructed states on the limiters was neglected according to the approximation described earlier. By means of chain rule, transposition, and rearrangement of some terms, the left-hand side of Eq. (10) can be expressed as

$$\frac{\partial \mathbf{R}^T}{\partial \mathbf{U}} \boldsymbol{\Lambda}_J = \mathbf{P}_1 \boldsymbol{\Lambda}_J + \frac{\partial \mathbf{G}^T}{\partial \mathbf{U}} \mathbf{P}_2 \boldsymbol{\Lambda}_J \tag{14}$$

where the matrices $\mathbf{P}_1$ and $\mathbf{P}_2$ are defined as

$$\begin{aligned}\mathbf{P}_1 &= \left(\frac{\partial \mathbf{U}_L^T}{\partial \mathbf{U}} \frac{\partial \mathbf{H}^T}{\partial \mathbf{U}_L} + \frac{\partial \mathbf{U}_R^T}{\partial \mathbf{U}} \frac{\partial \mathbf{H}^T}{\partial \mathbf{U}_R}\right) \frac{\partial \mathbf{R}^T}{\partial \mathbf{H}} \\ \mathbf{P}_2 &= \left(\frac{\partial \mathbf{U}_L^T}{\partial \mathbf{G}} \frac{\partial \mathbf{H}^T}{\partial \mathbf{U}_L} + \frac{\partial \mathbf{U}_R^T}{\partial \mathbf{G}} \frac{\partial \mathbf{H}^T}{\partial \mathbf{U}_R}\right) \frac{\partial \mathbf{R}^T}{\partial \mathbf{H}}\end{aligned} \tag{15}$$

Before discussing the assembly of Eq. (14), a description of the preceding matrices is necessary. The matrices $\mathbf{P}_1$ and $\mathbf{P}_2$ are sparse. The gradient operator $\partial \mathbf{G}/\partial \mathbf{U}$ is also sparse, with a sparsity pattern that corresponds to the graph of the mesh. Specifically,

1) Matrix $\partial \mathbf{R}/\partial \mathbf{H}$ is of size $E \times N$. Thus, $[\partial \mathbf{R}/\partial \mathbf{H}]^T \boldsymbol{\Lambda}_J$ is a vector of length $E$. Each element of this vector, which can be thought to correspond to an edge of the mesh, represents the difference between the adjoint variables of the two nodes of the edge. For example, for the edge that connects the nodes $i$ and $j$, the corresponding element of the vector has the value $\boldsymbol{\Lambda}_{Ji} - \boldsymbol{\Lambda}_{Ji}$.

2) Diagonal matrices $\partial \mathbf{H}/\partial \mathbf{U}_L$ and $\partial \mathbf{H}/\partial \mathbf{U}_R$ are of size $E \times E$. They represent the differentiation of the numerical flux formulation. For the line corresponding to the edge $ij$, the numerical flux Jacobians $\partial \hat{\boldsymbol{\Phi}}_{ij}/\partial \hat{\mathbf{u}}_i$ are the elements of $\partial \mathbf{H}/\partial \mathbf{U}_L$, and $\partial \hat{\boldsymbol{\Phi}}_{ij}/\partial \hat{\mathbf{u}}_j$ are the elements of $\partial \mathbf{H}/\partial \mathbf{U}_R$. When the node $i$ is lying on the boundary, the numerical boundary fluxes $\partial \boldsymbol{\Phi}_i^{\text{bc}}/\partial \mathbf{u}_i$ are also present. The same holds for node $j$. As the hat on the numerical flux Jacobian indicates, in the case of second-order accuracy these fluxes are evaluated with reconstructed variables $\hat{\mathbf{u}}_i$ and $\hat{\mathbf{u}}_j$.

3) Matrices $\partial \mathbf{U}_L/\partial \mathbf{U}$ and $\partial \mathbf{U}_R/\partial \mathbf{U}$ are of size $N \times E$. The elements of $\partial \mathbf{U}_L/\partial \mathbf{U}$ are $\partial \hat{\mathbf{u}}_i/\partial \mathbf{u}_i$. Introducing the transformation matrix between conservative and primitive variables, one has

$$\frac{\partial \hat{\mathbf{u}}_i}{\partial \mathbf{u}_i} = \frac{\partial \hat{\mathbf{u}}_i}{\partial \hat{\mathbf{v}}_i} \frac{\partial \hat{\mathbf{v}}_i}{\partial \mathbf{v}_i} \frac{\partial \mathbf{v}_i}{\partial \mathbf{u}_i}$$

The middle matrix is the identity because for each reconstructed primitive variable, $\partial \hat{v}_i/\partial v_i = 1$ [see Eq. (4)]. Therefore, one is left with the product of the transformation matrix evaluated with $\hat{\mathbf{v}}_i$ and its inverse evaluated with $\mathbf{v}_i$. Note that this product would result in the identity only in the case of first-order spatial accuracy (i.e., $\hat{\mathbf{v}}_i = \mathbf{v}_i$). The elements of $\partial \mathbf{U}_R/\partial \mathbf{U}$ are simply obtained by exchanging the $i$ with the $j$.

4) Matrices $\partial \mathbf{U}_L/\partial \mathbf{G}$ and $\partial \mathbf{U}_R/\partial \mathbf{G}$ are of size $N \times E$ for each component of the gradient $\mathbf{G}$. These matrices, together with $\partial \mathbf{U}_L/\partial \mathbf{U}$ and $\partial \mathbf{U}_R/\partial \mathbf{U}$, represent the differentiation of the reconstruction formulation [i.e., the MUSCL-like reconstruction of Eq. (4)]. The elements of the left matrix are $\partial \hat{\mathbf{u}}_i/\partial \nabla \mathbf{v}_i$. Introducing the transformation matrix, one has

$$\frac{\partial \hat{\mathbf{u}}_i}{\partial \nabla \mathbf{v}_i} = \frac{\partial \hat{\mathbf{u}}_i}{\partial \hat{\mathbf{v}}_i} \frac{\partial \hat{\mathbf{v}}_i}{\partial \nabla \mathbf{v}_i}$$

As can be seen from Eq. (4),

$$\frac{\partial \hat{v}_i}{\partial \nabla v_i} = \frac{\sigma_i \Delta \mathbf{x}_{ij}}{2}$$

In the case of $\partial \mathbf{U}_R/\partial \mathbf{G}$, $i$ is exchanged with $j$ and

$$\frac{\partial \hat{v}_j}{\partial \nabla v_j} = \frac{-\sigma_j \Delta \mathbf{x}_{ij}}{2}$$

5) The differentiation of the gradient formulation [9] is $\partial \mathbf{G}/\partial \mathbf{U}$. If transposition is not applied, then the differentiated gradient routine would take a vector as input and give a multicomponent vector as output. Clearly, the number of components must be equal to that of the gradient (i.e., equal to the number of space dimensions). When transposition is applied, the situation is inverted. Therefore, the differentiated gradient routine has to take a multicomponent vector as input and give a vector as output. In practice, the derivation of the

gradient operator is much less intuitive than that of the previously described operators.

Note that as already mentioned, the Roe Jacobians of Eq. (11) are evaluated with reconstructed variables. It means that these Jacobians are different from the ones required by an implicit solution scheme (see Sec. IV). The latter, in fact, requires these Jacobians to be evaluated with averaged variables (i.e., first-order Jacobian). It means that for reconstruction schemes, the Jacobians available from the implicit solution scheme cannot be reused to assemble the adjoint. However, in the case of an artificial dissipation scheme, the reuse of the Jacobians would be possible [8].

### C. Two-Pass Matrix-Free Assembly

The aforementioned matrices, which are very useful for explanatory reasons, are not formed at all. The assembly is, in fact, completely matrix-free; that is, the elements of the matrices are computed on-the-fly every time the assembly has to be performed. The matrix-free assembly reflects the need to keep the memory requirements to the same level as that of the flow solver. Especially with an implicit solution scheme, as the one used in this work, additional storage of the matrices may not be affordable.

The matrix-vector product in Eq. (14) is carried out in two steps:

1) First, the differentiated residual routine is run, which computes the two vectors $\mathbf{P}_1\mathbf{\Lambda}_J$ and $\mathbf{P}_2\mathbf{\Lambda}_J$, given the adjoint vector $\mathbf{\Lambda}_J$ as input. Note that in the case of first-order spatial accuracy only the vector $\mathbf{P}_1\mathbf{\Lambda}_J$ needs to be computed. The multicomponent vector $\mathbf{P}_2\mathbf{\Lambda}_J$ can be thought of as a kind of gradient that, due to the transposition, is an output, rather than an input, of the differentiated residual routine;

2) Once $\mathbf{P}_1\mathbf{\Lambda}_J$ and $\mathbf{P}_2\mathbf{\Lambda}_J$ are computed, the differentiated gradient routine is run. The routine takes the multicomponent vector $\mathbf{P}_2\mathbf{\Lambda}_J$ as input and gives the vector $[\partial\mathbf{G}/\partial\mathbf{U}^T]\mathbf{P}_2\mathbf{\Lambda}_J$ as output, which is added to $\mathbf{P}_1\mathbf{\Lambda}_J$, thus completing the assembly.

In previous work [9], for the 2D Euler equations, the dependency of the states upon the gradients was not made explicit, as in this case. The result was that a one-pass assembly was derived directly. Such an assembly allows an easy inclusion of the differentiation of the limiters. However, it requires additional data structure and storage, making the assembly not efficient in terms of both memory and operations. For the 3D case, the two-pass assembly presented here seems more suitable because it uses the same data structure of the flow solver and avoids extra storage.

### D. Verification of the Adjoint Implementation

There is a systematic way of verifying the correctness of the transposed Jacobians [15]. It is known that the relation $\mathbf{V}^T\mathbf{M}\mathbf{U} = \mathbf{U}^T\mathbf{M}^T\mathbf{V}$ holds for the two vectors $\mathbf{V}$ and $\mathbf{U}$ and for the matrix $\mathbf{M}$. Therefore, each routine of the code is differentiated in two versions: one applying the Jacobian and the other one applying the transposed Jacobian. In the present work, after the accuracy of the Jacobians was verified, it was checked that each routine satisfies the preceding relation up to machine accuracy.

The accuracy of the Jacobians is verified easily. For most of the Jacobians that appear in Eq. (15), analytical expressions for their elements are already available from the flow solver or exist in the literature. The gradient Jacobian can also be verified easily because this operator is a linear function of the variables [i.e., $\mathbf{G}(\mathbf{V}) = (\partial\mathbf{G}/\partial\mathbf{V})\mathbf{V}$]. Finite differences, although not exact, are also used to check the Jacobians.

Finally, the accuracy of the computed adjoint sensitivity is verified against that obtained by the linearized model, for which the sensitivity and the equations to solve are, respectively,

$$\frac{\partial\mathbf{R}}{\partial\mathbf{U}}\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}\alpha} = -\frac{\partial\mathbf{R}}{\partial\alpha}, \qquad \frac{\mathrm{d}J}{\mathrm{d}\alpha} = \frac{\partial J}{\partial\alpha} + \frac{\partial J}{\partial\mathbf{U}}\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}\alpha} \tag{16}$$

where $\partial\mathbf{U}/\partial\alpha$ is the flow sensitivity. The linearized equation is solved with a procedure similar to that for the adjoint equation. At convergence, it is verified that the two sensitivities agree up to machine precision.

## IV. Time Marching of Flow and Adjoint Equations

Both the flow and the adjoint equations are advanced in time using an implicit time-stepping scheme. The scheme is essentially the same for both solvers. At each time step, a system of linear equations arises, which is solved iteratively to the required level of accuracy.

### A. Implicit Pseudo-Time-Stepping Method

An implicit pseudo-time-stepping method is used to solve the semidiscrete system in Eq. (8). Formally, the method is a defect-correction [17] approach in pseudotime. In practice, some algebra shows that it coincides with an Euler scheme that uses an approximate residual Jacobian [18].

The time derivative in Eq. (8) is discretized using a forward approximation:

$$\mathbf{D}\left(\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right) \approx \mathbf{D}_t(\mathbf{U}^{n+1} - \mathbf{U}^n)$$

where the diagonal matrix $\mathbf{D}_t$ contains the control volumes divided by their local time steps, $V_i/\Delta t_i$. The residual $\tilde{\mathbf{R}}$ of a low-order discretization is introduced. According to the defect-correction approach, a solution to Eq. (8) is found iteratively as

$$\mathbf{D}_t(\mathbf{U}^{n+1} - \mathbf{U}^n) + \tilde{\mathbf{R}}(\mathbf{U}^{n+1}) = \tilde{\mathbf{R}}(\mathbf{U}^n) - \mathbf{R}(\mathbf{U}^n) \tag{17}$$

where the low-order residual terms vanish once the solution is converged. If the low-order residual is expanded linearly,

$$\tilde{\mathbf{R}}(\mathbf{U}^{n+1}) \approx \tilde{\mathbf{R}}(\mathbf{U}^n) + \frac{\partial\tilde{\mathbf{R}}}{\partial\mathbf{U}^n}(\mathbf{U}^{n+1} - \mathbf{U}^n)$$

and substituted into Eq. (17), one obtains

$$\left(\mathbf{D}_t + \frac{\partial\tilde{\mathbf{R}}}{\partial\mathbf{U}}\right)^n(\mathbf{U}^{n+1} - \mathbf{U}^n) = -\mathbf{R}^n \tag{18}$$

In the present work, the Jacobian $\partial\tilde{\mathbf{R}}/\partial\mathbf{U}$ is first-order accurate because the reconstruction contribution is neglected, and it is approximate because the Jacobians are frozen in the differentiation of the numerical fluxes, as in Eqs. (5) and (6). At each iteration, the local time steps $\Delta t_i$ are computed using a CFL-number update of the type

$$\mathrm{CFL}^n = \beta\mathrm{CFL}^{n-1}\frac{L_2(\mathbf{R}^{n-2})}{L_2(\mathbf{R}^{n-1})} \tag{19}$$

where $L_2(\mathbf{R})$ is a discrete norm of the residual vector and $\beta$ a suitable parameter.

The same solution scheme is used for the adjoint of Eq. (10) in spite of the linearity of this system of equations. Hence, the problem is treated as a nonlinear one. The reason for adopting this solution procedure is that the off-diagonal contribution arising from the reconstruction contribution, $[\partial\mathbf{G}/\partial\mathbf{U}^T]\mathbf{P}_2$, makes the matrix in Eq. (10) poorly diagonally dominant. The consequence is that attempts to solve the system by means of an iterative linear solver may result in the solution failing to converge.

Applying the implicit pseudo-time-stepping method described earlier to the adjoint system of Eq. (10) gives

$$\left(\mathbf{D}_t + \frac{\partial\tilde{\mathbf{R}}}{\partial\mathbf{U}}\right)^n\left(\mathbf{\Lambda}_J^{n+1} - \mathbf{\Lambda}_J^n\right) = -\left(\frac{\partial\mathbf{R}^T}{\partial\mathbf{U}}\mathbf{\Lambda}_J^n - \frac{\partial J^T}{\partial\mathbf{U}}\right) \tag{20}$$

This nonlinear solution procedure results in a robust solver because it shifts the poorly diagonally dominant matrix to the right-hand side of the equation and uses a diagonally dominant matrix at the left-hand side to drive the solution process.

Constrained shape optimization problems may require Eq. (20) to be solved as many times as the number of functionals; that is, one has to compute the adjoint $\mathbf{\Lambda}_J$ of each functional $J$. The matrix terms appearing on the right-hand side of Eq. (20) are expensive to compute. However, because they are identical for all functionals, it makes sense to perform more matrix-vector products simultaneously once these terms are available. In practice, each time the assembly in Eq. (14) is performed, different adjoints are assembled together.

A similar reasoning is also valid for the left-hand side of Eq. (20). However, the benefit that has to be expected for this part depends on whether the linear system procedure stores the matrix. This aspect is clarified later. In general, numerical experiments show that simultaneous time stepping gives appreciable time savings compared with the sequential solution.

### B. Linear System of Equations

At each time step, Eq. (18) implies the solution of a linear system $\mathbf{Mz} = \mathbf{b}$, where

$$\mathbf{M} = \left(\mathbf{D}_t + \frac{\partial\tilde{\mathbf{R}}}{\partial\mathbf{U}}\right)^n, \qquad \mathbf{z} = \mathbf{U}^{n+1} - \mathbf{U}^n, \qquad \mathbf{b} = -\mathbf{R}^n \tag{21}$$

A simple iterative procedure is employed that computes corrections of the type

$$\mathbf{z}^{k+1} = \mathbf{z}^k + \Delta\mathbf{z}, \qquad \Delta\mathbf{z} = \mathbf{P}_M^{-1}\mathbf{R}_L^k, \qquad \mathbf{R}_L^k = \mathbf{b} - \mathbf{M}\mathbf{z}^k \tag{22}$$

where $\mathbf{R}_L$ is the residual of the linear system, and $\mathbf{P}_M$ the preconditioner computed from $\mathbf{M}$. The procedure is iterated until the discrete norm of the linear residual is less than a fraction of that of the nonlinear residual; that is

$$L_2\left(\mathbf{R}_L^k\right) \leq cL_2(\mathbf{R}^n) \tag{23}$$

Numerical experiments show that $c = 0.1$ is a safe value for obtaining convergence, although for some conditions a lower value may be needed. In general, for $c = 0.1$, the average number of iterations that are needed at each pseudotime step is $k \approx 4$–8. Very low values of tolerance are not useful because no improvement in the convergence rate can be expected due to the approximations in the Jacobian.

For the adjoint equation, where $\mathbf{M}$, $\mathbf{z}$, and $\mathbf{b}$ are defined according to Eq. (20), the procedure is essentially the same. If storage is allowed (described later), then there are savings in terms of operations, due to the fact that the matrix $\mathbf{M}$ and its preconditioner $\mathbf{P}_M$ only need to be computed at the first nonlinear iteration. In fact, these matrices are dependent on the conservative variables and not on the adjoint variables.

### C. Preconditioning

The preconditioner should be a good approximation of the original matrix ($\mathbf{P}_M \approx \mathbf{M}$) and, moreover, it should be relatively simple to invert. A symmetric Gauss–Seidel preconditioner was implemented, which can be expressed as

$$\mathbf{P}_M = (\mathbf{D}_M + \mathbf{L}_M)\mathbf{D}_M^{-1}(\mathbf{D}_M + \mathbf{U}_M) \tag{24}$$

This preconditioner can be inverted by means of a forward solve followed by a backward solve. To perform the sweeps, a distance-1 topology is needed; that is, for each node, stencil $\mathcal{N}_i$, which contains the node $i$ and all its distance-1 neighbors, must be available. These data are all that is needed to perform the sweeps, because for a first-order Jacobian each edge produces four nonzero entries: two on the diagonal and two off-diagonal [18]. Therefore, the nonzero entries for each node (a Jacobian row) are only the nodes in the stencil.

The forward solve $(\mathbf{D}_M + \mathbf{L}_M)\Delta\mathbf{z}^* = \mathbf{R}_L^k$ is performed with the first sweep on the nodes:

$$\Delta\mathbf{z}_i^* = \mathbf{D}_{M_i}^{-1}\left(\mathbf{R}_{Li}^k - \sum_{j\in\mathcal{L}_i}\mathbf{L}_{M_{ij}}\Delta\mathbf{z}_j^*\right), \qquad (i = 1, N) \tag{25}$$

whereas the backward solve $(\mathbf{I} + \mathbf{D}_M^{-1}\mathbf{U}_M)\Delta\mathbf{z} = \Delta\mathbf{z}^*$ is performed with the second sweep on the nodes:

$$\Delta\mathbf{z}_i = \Delta\mathbf{z}_i^* - \mathbf{D}_{M_i}^{-1}\sum_{j\in\mathcal{U}_i}\mathbf{U}_{M_{ij}}\Delta\mathbf{z}_j, \qquad (i = N, 1) \tag{26}$$

$\mathcal{L}_i$ and $\mathcal{U}_i$ are subsets of the stencil $\mathcal{N}_i$ ($\forall\ j \in \mathcal{L}_i$: $j < i$ and $\forall\ j \in \mathcal{U}_i$: $j > i$). The elements of $\mathbf{D}_M$, $\mathbf{L}_M$, and $\mathbf{U}_M$ are

$$\begin{aligned} \mathbf{D}_{M_i} &= \frac{V_i}{\Delta t_i}\mathbf{I} + \sum_{j=1}^{N_i}\frac{\partial\mathbf{\Phi}_{ij}}{\partial\mathbf{u}_i} + \frac{\partial\mathbf{\Phi}_i^{\mathrm{bc}}}{\partial\mathbf{u}_i} \\ \mathbf{L}_{M_{ij}} &= -\frac{\partial\mathbf{\Phi}_{ij}}{\partial\mathbf{u}_i}, \qquad \mathbf{U}_{M_{ij}} = \frac{\partial\mathbf{\Phi}_{ij}}{\partial\mathbf{u}_j} \end{aligned} \tag{27}$$

where the boundary flux Jacobian is only present when the node $i$ is lying on the boundary.

For the adjoint solution, for which the elements of the preconditioner are transposed, the matrices $\mathbf{D}_{M_i}^T$, $\mathbf{L}_{M_{ij}}^T$, and $\mathbf{U}_{M_{ij}}^T$ are employed. Moreover, $\mathbf{U}_{M_{ij}}^T$ should be used in Eq. (25) instead of Eq. (26) and vice versa. In the case of multiple right-hand sides, there are a number of linear residuals that must be preconditioned, one for each functional $J$. For instance, consider Eq. (25). There, $\Delta\mathbf{z}_i^*$ is computed given $\mathbf{R}_{L_i}$ for each functional, with $\mathbf{D}_{M_i}^{-T}$ and $\mathbf{U}_{M_{ij}}^T$ being equal for all functionals. This practice consists of simultaneously solving several linear systems that have the same matrix in common.

### D. Storage vs the Matrix-Free Approach

Storage of the matrix $\mathbf{M}$ and of its preconditioner $\mathbf{P}_M$ is beneficial in terms of CPU time. However, depending on the size of the mesh, the amount of memory involved may be excessive. The preconditioning strategy implemented in this work has the feature of not requiring any preparatory work for the preconditioner. In fact, as shown in Eqs. (25) and (26), only the inversion of the diagonal submatrices is required. Consequently, there is no need to store both the matrix and the preconditioner. It is enough to store only $\mathbf{D}_M$, $\mathbf{L}_M$, and $\mathbf{U}_M$, thus halving the storage. With these elements available one can 1) perform the matrix-vector product to compute the linear residual [see Eq. (22)] and 2) apply the preconditioner to the latter [see Eqs. (25) and (26)]. In practice, $\mathbf{D}_M$, $\mathbf{L}_M$, and $\mathbf{U}_M$ are precomputed and stored at each nonlinear iteration, at least for the flow equations. In fact, as already mentioned, the adjoint solution only requires them to be computed once.

The approach just described can be referred to as the storage approach. In contrast, there is a matrix-free approach, which can be applied in this case because the preconditioner does not require preparatory work. Looking at the definition of the lower and the upper matrices [see Eq. (27)], one understands that the sweeps of Eqs. (25) and (26) only require $\mathbf{D}_M$ to be stored and allow $\mathbf{L}_M$ and $\mathbf{U}_M$ to be computed on-the-fly. Also, for the matrix-vector product that is required to compute the linear system residual $\mathbf{R}_L$ [see Eq. (22)], the elements can be computed on-the-fly. The matrix-free approach is very beneficial in terms of memory because it requires an amount of storage similar to that of an explicit scheme. However, recomputing $\mathbf{L}_M$ and $\mathbf{U}_M$ on-the-fly gives a penalty in terms of CPU time.

## V. Numerical Results

Two configurations are considered: the ONERA-M6 wing [19] and the DLR-F6 wing–body [20]. The unstructured meshes of tetrahedral elements used for the computations are depicted in Figs. 1 and 2, respectively. The pressure coefficients obtained by the present solver are compared with those obtained by FOI's Edge [21] flow solver on the same meshes. The Edge flow solver is an unstructured solver based upon an artificial dissipation scheme.[§] The comparison is shown in Figs. 3 and 4. Comparison of the pressure coefficients and other quantities at different spanwise sections can be found in the technical report that describes the present flow solver in more detail [22].

Figure 5 shows the convergence histories of the flow, linearized, and adjoint solvers for the ONERA-M6 wing. These are represented in terms of nonlinear iterations. As can be seen from Fig. 5a, the scaled residual of the flow solver, which is defined as

[§]Documentation available online at http://www.foi.se/edge [retrieved 23 October 2007].

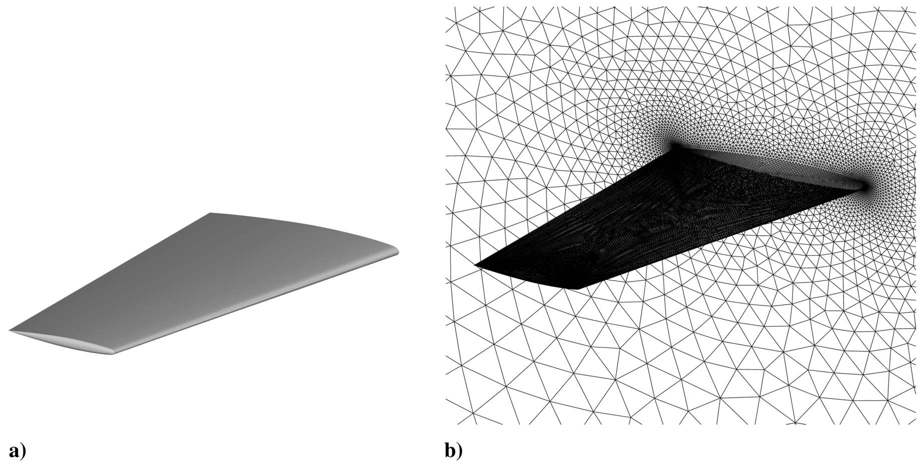


**Fig. 1 Wing configuration: a) ONERA-M6 wing and b) unstructured mesh around the wing.**

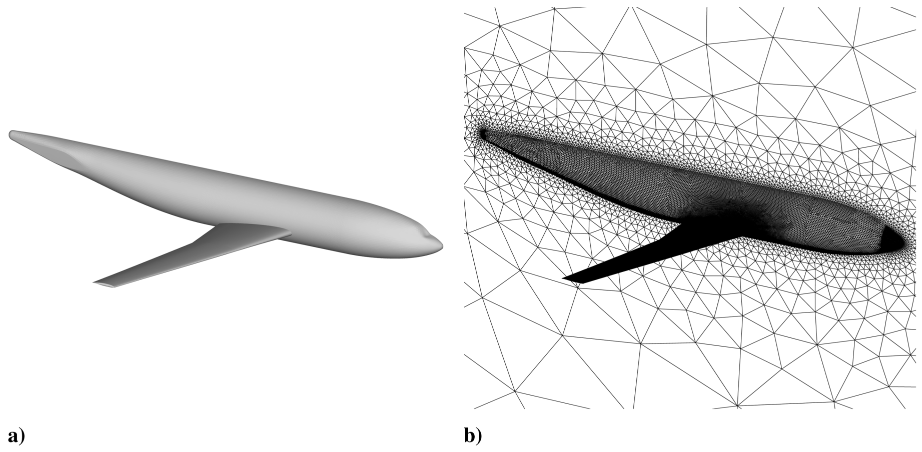


**Fig. 2 Wing–body configuration: a) DLR-F6 wing–body and b) unstructured mesh around the wing–body.**

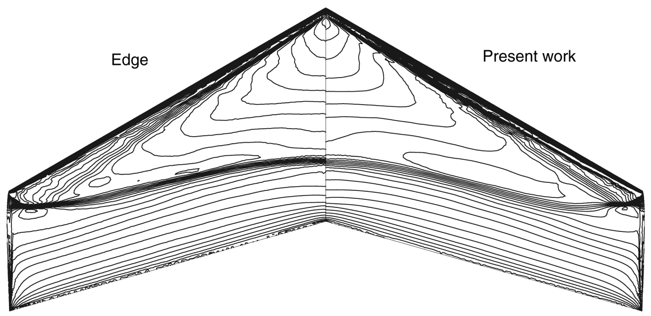


**Fig. 3 ONERA-M6 wing at $M_\infty = 0.84$ and $\theta = 3.06\,\mathrm{deg}$; contours of the pressure coefficient obtained by the present solver (right) and by the Edge solver (left).**

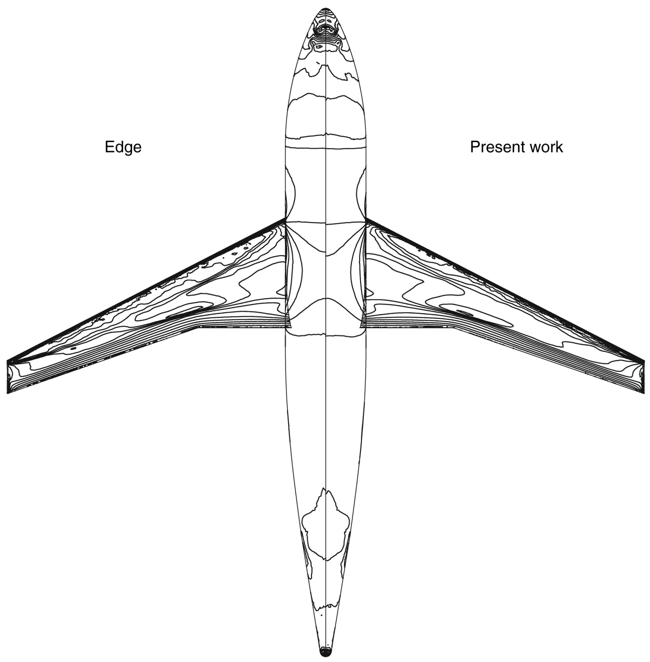


**Fig. 4 DLR-F6 at $M_\infty = 0.75$ and $\theta = 0.5\,\mathrm{deg}$; contours of the pressure coefficient obtained by the present solver (right) and by the Edge solver (left).**

$L_2(\mathbf{R}^n)/L_2(\mathbf{R}^0)$, is reduced by 10 orders of magnitude. The lift coefficient (see Fig. 5c) is fully converged at less than half of the total number of iterations. In terms of linear iterations (see Fig. 5b), an average of six of these are required for each nonlinear iteration. A maximum of 11 linear iterations is required around the 30th nonlinear iteration. In general, especially for 3D computations, the startup phase of the implicit solution procedure appears to be a critical one. A safe choice of the solution parameters has to be made to obtain convergence. For instance, the parameter $\beta$ in Eq. (19) is set equal to 1–2, whereas in 2D computations it can be safely set to values that are 10 times larger.

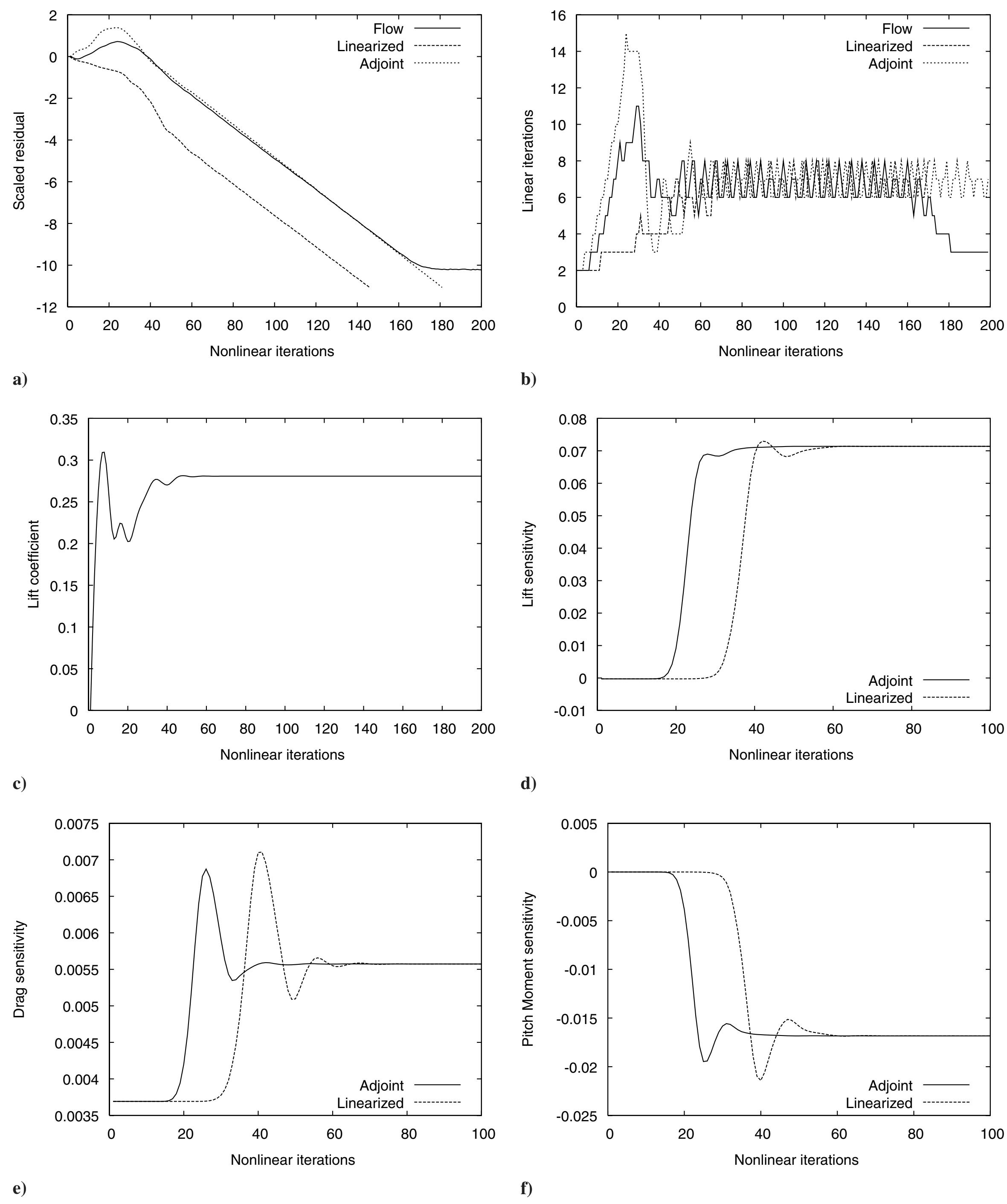
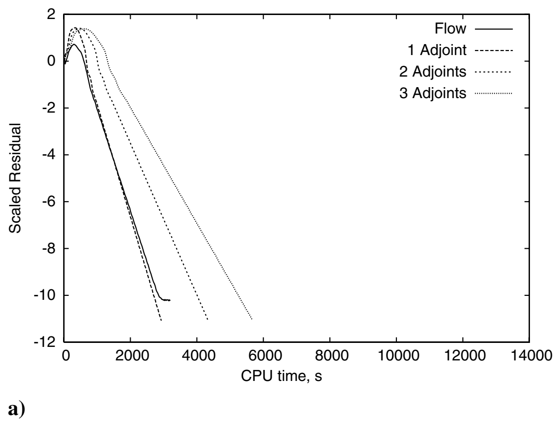


**Fig. 5 ONERA-M6 wing convergence histories of the flow, adjoint, and linearized solvers: a) residual, b) linear vs nonlinear iterations, c) lift coefficient, d) lift sensitivity, e) drag sensitivity, and f) pitching-moment sensitivity.**

Convergence histories of the flow and the adjoint solvers for the two configurations are shown in Figs. 6 and 7, respectively. To compare the efficiency of the adjoint solver with that of the flow solver, these convergence histories are plotted in terms of CPU time. For the convergence histories of Figs. 6a and 7a, the elements $\mathbf{L}_M$, $\mathbf{U}_M$, and $\mathbf{D}_M$ of the matrix $\mathbf{M}$ were stored. It appears that the adjoint solver residual overlaps with the flow solver residual; that is, one adjoint solution requires the same amount of time as one flow solution. This positive result can be explained as follows. The larger amount of CPU time required to assemble the right-hand side of Eq. (20), compared with Eq. (18), is cancelled out by the time saved on the left-hand side for the computation of the matrix elements. In fact, as already mentioned, these elements for the adjoint are computed only once. In the case of multiple adjoint solutions, looking at these pictures, it appears that the simultaneous solution of two adjoints saves 25% of CPU time compared with two sequential solutions. In the case of three adjoint solutions, the CPU time saving rises to 33%.

The convergence histories of Figs. 6b and 7b were produced using the matrix-free option; that is, $\mathbf{L}_M$ and $\mathbf{U}_M$ are always computed on-the-fly for both matrix-vector products and preconditioning. Clearly, there is a penalty in terms of time. As can be seen, the CPU time required to converge the flow is around three times more than in the storage case of Figs. 6a and 7a. A single adjoint solution required in this case around 10% more CPU time than the flow solution. In fact, for the adjoint solution, on-the-fly computation means that the advantage of having the same $\mathbf{L}_M$ and $\mathbf{U}_M$ for all iterations is lost in this case. Nevertheless, on-the-fly computations allow maximal exploitation of the advantages given by simultaneous adjoint solutions. As can be seen from Figs. 6b and 7b, two simultaneous



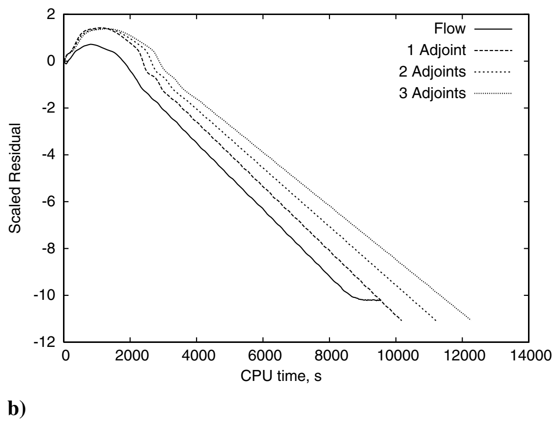
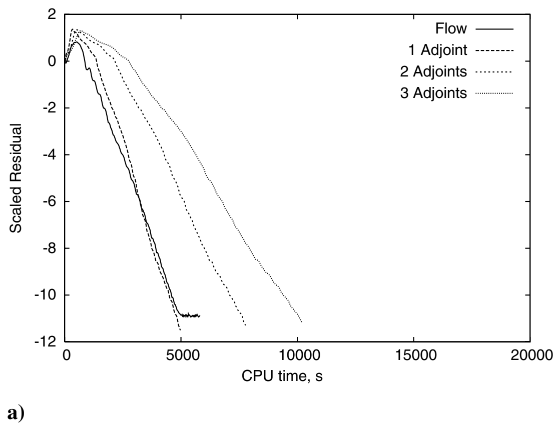


**Fig. 6 ONERA-M6 wing flow and adjoint convergence histories: a) storage of the preconditioner and b) matrix-free preconditioning.**



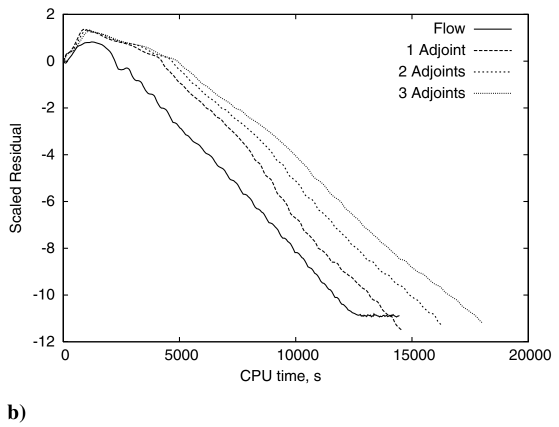


**Fig. 7 DLR-F6 wing–body flow and adjoint convergence histories: a) storage of the preconditioner and b) matrix-free preconditioning.**

adjoint solutions give around 45% time savings compared with sequential solutions. In the case of three adjoint solutions, the time saving rises to 60%.

In terms of memory requirements, the adjoint requires an amount of memory that is of the same order as that required by the flow solver. Figure 8 shows these requirements for storage and matrix-free computations. Because the picture is valid for both configurations, the memory is normalized by the memory required by the flow solver for matrix-free computations. As can be seen, the flow solver requires 2.7 times more memory when the matrix elements are stored than in the case of matrix-free computations. When the matrix elements are stored, the adjoint takes only 11% more memory than the flow solver. This memory increase rises to 40% in the case of three simultaneous adjoint solutions. In the case of matrix-free computations, the difference in memory use between the flow solver and the adjoint solver is clearly larger. For instance, three adjoints simultaneously require slightly more than two times the memory required by the flow solver.

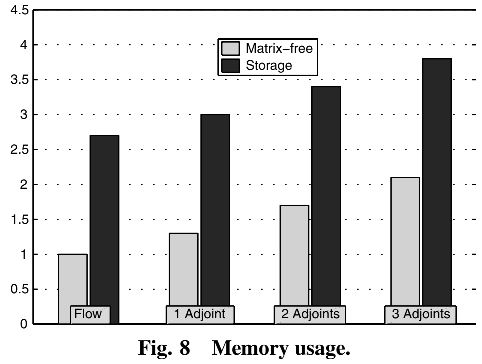


**Fig. 8 Memory usage.**

The adjoint solver was verified using the linearized solver, which was implemented according to Eq. (16). As already mentioned, Fig. 5a shows the residual history of the three solvers in terms of nonlinear iterations. As can be seen, the solvers converge at the same rate. When the solution stabilizes itself, the average number of linear iterations (see Fig. 5b) appears to be the same for the three solvers. Compared with the flow solver, the adjoint solver requires more iterations in the startup phase, whereas the linearized solver requires less. The convergence history for the lift, drag, and pitching-moment sensitivities are shown in Figs. 5d–5f, respectively. Only the first 100 nonlinear iterations are shown. As can be seen, the sensitivities computed with the linearized and the adjoint solvers converge exactly to the same value. These sensitivity values have also been compared with values obtained by finite differences, with a suitable increment of $10^{-6}$. It turned out that the lift sensitivity differs 0.87%, the drag sensitivity differs 0.27%, and the pitching-moment sensitivity differs 2%.

## VI. Conclusions

The discrete adjoint of an unstructured finite volume solver for the three-dimensional Euler equations was developed and implemented. An efficient two-pass assembly for a MUSCL-like reconstruction scheme was presented. The assembly uses the same data structure as the flow solver and is completely matrix-free.

An implicit solution scheme for the flow equations was tailored to the adjoint equations. The scheme is a defect-correction iteration that uses symmetric Gauss–Seidel preconditioning and features the possibility to store the matrices or to run completely matrix-free. The adjoint solver, in case of both storage and matrix-free solutions, was shown to require almost the same CPU time and memory as the flow solver.

To compute the sensitivity of more functionals simultaneously, the solution scheme was modified to solve the equations for multiple right-hand sides. Solving up to three adjoint equations simultaneously was shown to give appreciable time savings compared with sequential solutions.

## Acknowledgments

This research was supported by the Dutch Technology Foundation STW, Applied Science Division of the Netherlands Organisation for Scientific Research (NWO), and the technology program of the Dutch Ministry of Economic Affairs. The authors would like to thank FOI for providing their Edge software. In particular, the first author wants to thank Olivier Amoignon for his support in using this software when he was a visiting researcher at FOI in Stockholm.

## References

[1] Newman III, J. C., Taylor III, A. C., Barnwell, R., Newman, P., and Hou, G.-W., "Overview of Sensitivity Analysis and Shape Optimization for Complex Aerodynamic Configurations," *Journal of Aircraft*, Vol. 36, No. 2, 1999, pp. 87–96.

[2] Mohammadi, B., "A New Optimal Shape Design Procedure for Inviscid and Viscous Turbulent Flows," *International Journal for Numerical Methods in Fluids*, Vol. 25, No. 2, 1997, pp. 183–203. doi:10.1002/(SICI)1097-0363(19970730)25:2<183::AID-FLD545>3.0.CO;2-U

[3] Nemec, N., and Zingg, D., "Newton-Krylov Algorithm for Aerodynamic Design Using the Navier Stokes Equations," *AIAA Journal*, Vol. 40, No. 6, 2002, pp. 1146–1154.

[4] 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, 2003, pp. 198–205.

[5] Nielsen, E. J., Lu, J., Park, M. A., and Darmofal, D. L., "An Implicit, Exact Dual Adjoint Solution Method for Turbulent Flows on Unstructured Grids," *Computers and Fluids*, Vol. 33, No. 9, 2004, pp. 1131–1155. doi:10.1016/j.compfluid.2003.09.005

[6] Müller, J.-D., and Cusdin, P., "On the Performance of Discrete Adjoint CFD Codes Using Automatic Differentiation," *International Journal for Numerical Methods in Fluids*, Vol. 47, Nos. 8–9, 2005, pp. 939–945. doi:10.1002/fld.885

[7] Amoignon, O., and Berggren, M., "Adjoint of a Median-Dual Finite-Volume Scheme: Application to Transonic Aerodynamic Shape Optimization," Uppsala Univ., Rept. 2006-013, Uppsala, Sweden, 2006.

[8] Mavriplis, D. J., "Discrete Adjoint-Based Approach for Optimization Problems on Three-Dimensional Unstructured Meshes," *AIAA Journal*, Vol. 45, No. 4, 2007, pp. 740–750. doi:10.2514/1.22743

[9] Carpentieri, G., Koren, B., and van Tooren, M. J. L., "Adjoint-Based Aerodynamic Shape Optimization on Unstructured Meshes," *Journal of Computational Physics*, Vol. 224, No. 1, 2007, pp. 267–287. doi:10.1016/j.jcp.2007.02.011

[10] Barth, T. J., "Aspects of Unstructured Grids and Finite-Volume Solvers for the Euler and Navier-Stokes Equations," *Unstructured Grid Methods for Advection Dominated Flows*, AGARD Rept. AGARD-R-787, Neuilly-sur-Seine, France, May 1992.

[11] Selmin, V., and Formaggia, L., "Unified Construction of Finite Element and Finite Volume Discretizations for Compressible Flows," *International Journal for Numerical Methods in Engineering*, Vol. 39, No. 1, 1996, pp. 1–32. doi:10.1002/(SICI)1097-0207(19960115)39:1<1::AID-NME837>3.0.CO;2-G

[12] Venkatakrishnan, V., "Convergence to Steady State Solutions of the Euler Equations on Unstructured Grids with Limiters," *Journal of Computational Physics*, Vol. 118, No. 1, 1995, pp. 120–130. doi:10.1006/jcph.1995.1084

[13] Roe, P., "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," *Journal of Computational Physics*, Vol. 43, No. 2, 1981, pp. 357–372. doi:10.1016/0021-9991(81)90128-5

[14] Steger, J., and Warming, R., "Flux Vector Splitting of the Inviscid Gasdynamic Equations with Application to Finite-Difference Methods," *Journal of Computational Physics*, Vol. 40, No. 2, 1981, pp. 263–293. doi:10.1016/0021-9991(81)90210-2

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

[16] Dwight, R. P., and Brezillon, 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

[17] Koren, B., "Defect Correction and Multigrid for an Efficient and Accurate Computation of Airfoil Flows," *Journal of Computational Physics*, Vol. 77, No. 1, 1988, pp. 183–206. doi:10.1016/0021-9991(88)90162-3

[18] Mavriplis, D. J., "On Convergence Acceleration Techniques for Unstructured Meshes," AIAA Paper 98-2966, June 1998.

[19] Schmitt, V., and Charpin, F., "Pressure Distributions on the ONERA-M6-Wing at Transonic Mach Numbers," *Experimental Database for Computer Program Assessment*, AGARD Rept. AGARD-AR-138, Neuilly-sur-Seine, France, May 1979.

[20] Brodersen, O., and Stürmer, A., "Drag Prediction of Engine-Airframe Interference Effects Using Unstructured Navier-Stokes Calculations," AIAA Paper 2001–2414, June 2001.

[21] Eliasson, P., "Edge, a Navier–Stokes Solver, for Unstructured Grids," Swedish Defence Research Agency, Rept. FOI-R–0298–SE, Stockholm, Sweden, 2001.

[22] Carpentieri, G., "A Finite-Volume Solver for Unstructured Meshes," Delft Univ. of Technology, Rept. AE-TR-02-001-07, Delft, The Netherlands, 2007.