Towards Aerodynamic Shape Optimization of an Oblique Wing using the ADjoint Approach

Charles A. Mader[1], Joaquim R. R. A. Martins[2]
University of Toronto
Toronto, Ontario, Canada, M3H 5T6

Andre C. Marta[3] *Stanford University*
Stanford, CA 94305, U.S.A.

June 8, 2007

[1]M.A.Sc. Candidate,(416)667-7747,mader@utias.utoronto.ca
[2]Assistant Professor, CASI Member
[3]Doctoral Candidate

Abstract

The ADjoint method is applied to a three-dimensional Computational Fluid Dynamics (CFD) solver to generate the sensitivities required for aerodynamic shape optimization. The ADjoint approach selectively uses Automatic Differentiation (AD) to generate the partial derivative terms in the discrete adjoint equations. By selectively applying AD techniques, the computational cost and memory overhead incurred by using AD are significantly reduced, while still allowing for the the accurate treatment of arbitrarily complex governing equations and boundary conditions. Once formulated, the discrete adjoint equations are solved using the Portable, Extensible Toolkit for Scientific computation (PETSc). With this approach, the computed adjoint vector can be used to calculate the total sensitivities required for aerodynamic shape optimization of a complete aircraft configuration. The resulting sensitivities are compared with finite-difference derivatives to verify accuracy. Once formulated, these derivatives can then be used to perform gradient based aerodynamic optimization.

1 Introduction

1.1 The Oblique Wing

The oblique wing concept has been around since the late 1950's and was primarily developed by R. T. Jones during the 1960's and 1970's. R. T. Jones was able to show that an elliptic oblique wing was an optimum wing shape for supersonic flight [14]. However, despite this fact, the oblique wing configuration has not been developed beyond a test aircraft - the most extensive test of this design was the NASA built AD-1 test aircraft of the 1980's [1]. One of the major reasons for this is that, because of the asymmetry of the design, the flight dynamics of this configuration are problematic. In particular, some very prominent pitch-roll coupling developed in the AD-1 test aircraft [1]. In addition, there are minimum size limitations inherent to the oblique flying wing configuration. This limitation is caused by the combination of passenger cabin size requirements and minimum thickness to chord ratio (t/c) requirements for supersonic flight [26]. In recent years, there has been a resurgence in interest in the oblique wing configuration. This is partly because of the increased interest in more efficient, higher speed aircraft, but is also due in large part to the improvement of computer controls and the improvement in, and increased use of, UAVs. Because UAVs operate autonomously, two of the major issues with oblique wing configurations are removed. Because they are computer controlled, active stability control is more easily integrated, thus reducing some of the dynamic issues. Also, because the carrying capacity of a UAV is much smaller than a typical aircraft, an oblique wing UAV can be much smaller than a passenger aircraft would have to be to maintain an acceptable t/c ratio for supersonic flight. In this research we use the ADjoint method to develop the shape sensitivity information required to perform gradient based aerodynamic shape optimization of a full aircraft configuration. We intend to use aerodynamic shape optimization to find an improved oblique wing shape that takes advantage of the aerodynamic benefits of the asymmetric design while minimizing the adverse flight dynamic characteristics.

1.2 Adjoint methods

Adjoint methods for sensitivity analysis involving partial differential equations (PDE's) have been known and used for over three decades. They were first applied to solve optimal control problems and thereafter were used to perform sensitivity analysis of linear structural finite-element models. The first application to fluid dynamics is due to Pironneau [23]. The method was then extended by Jameson to perform airfoil shape optimization [13]. This method was also used, in conjunction with a Newton-Krylov solver, for airfoil shape optimization by Nemec and Zingg [21]. Since then, the adjoint method has been developed for more complex problems, leading to its application to the design optimization of complete aircraft configurations considering aerodynamics alone [24, 25], as well as aerodynamic and structural interactions [15]. The adjoint method has also been generalized for multidisciplinary systems [16]. However, because of the complexity inherent in developing a manually differentiated adjoint, these methods have not made their way into widespread general use.

2 Methodology

To remove this impediment, the ADjoint (Automatic Differentiation Adjoint) method has been developed. As described by Martins et al. [17, 18], the ADjoint method uses reverse mode automatic differentiation to accurately and efficiently compute the partial derivative terms required by the discrete adjoint equations. Once formulated from these partial derivatives, the discrete adjoint equations are solved using the Portable Extensible Toolkit for Scientific Computation (PETSc), which has several built-in capabilities including a sparse matrix solver.

In this work, the ADjoint method is applied to NSSUS, a flow solver under development at Stanford University through the ASC turbomachinery program [2]. NSSUS is a vertex-centered flow solver being developed to solve the Reynolds-averaged Navier-Stokes equations with a variety of boundary conditions and turbulence models. However, in this work only the Euler equations have been considered. Total sensitivities have been developed for the force coefficients C_L and C_D and the moment coefficients C_{M_x}, C_{M_y} and C_{M_z} with respect to the grid coordinates of the CFD Mesh. The sensitivities can be combined with the sensitivity of the grid coordinates to a set of design variables such as sweep, span, twist and wing surface shape to provide the total sensitivities required for aerodynamic shape optimization.With these sensitivities implemented, it is a fairly simple step to proceed onto aerodynamic shape optimization and from there onto aerostructural multidisciplinary design optimization.

3 Background

3.1 Semi-Analytic Sensitivity Analysis

For aerodynamic shape optimization, one must 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 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

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \frac{\partial I}{\partial w}\frac{\mathrm{d}w}{\mathrm{d}x}. \tag{3}$$

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,

$$\frac{\mathrm{d}\mathcal{R}}{\mathrm{d}x} = \frac{\partial \mathcal{R}}{\partial x} + \frac{\partial \mathcal{R}}{\partial w}\frac{\mathrm{d}w}{\mathrm{d}x} = 0. \tag{4}$$

This expression provides the means of eliminating the total derivative $\mathrm{d}w/\,\mathrm{d}x$ from the total sensitivity computation for I. By rewriting equation (4) as

$$\frac{\partial \mathcal{R}}{\partial w}\frac{\mathrm{d}w}{\mathrm{d}x} = -\frac{\partial \mathcal{R}}{\partial x} \tag{5}$$

and substituting the solution into equation 3 we get equation 6

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} - \frac{\partial I}{\partial w}\left[\frac{\partial \mathcal{R}}{\partial w}\right]^{-1}\frac{\partial \mathcal{R}}{\partial x}. \tag{6}$$

Note that all of the terms in equation 6 are partial derivatives. However, there is now a series of linear system solutions that need to be solved. One can either solve the system created by the last two terms, which corresponds to the direct sensitivity method, or one can solve the system generated by the second and third terms corresponding to the adjoint sensitivity method.

The most computationally intensive step for both of these problems is the solution of the respective linear systems. In the case of the first problem — the *direct method* — we have to solve a linear system of N_w equations 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. We are interested in the adjoint method because we have several design variables, x, and few output variables, I. The adjoint equations are written as shown in equation 7 and 8

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial I}{\partial w}, \tag{7}$$

and

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \psi^T \frac{\partial \mathcal{R}}{\partial x}, \tag{8}$$

where ψ is the *adjoint vector*.

When it comes to implementation, there are two main ways of obtaining the discrete adjoint equations (7) for a given system of PDEs. The *continuous adjoint approach* forms a continuous adjoint problem from the governing PDEs and then discretizes this problem to solve it numerically. The *discrete adjoint approach* forms an adjoint from the discretized PDEs. Each of these approaches results in a different system of linear equations, but they will both, in theory, converge to the same result as the mesh is refined. However, the discrete approach has certain advantages in that the sensitivities are consistent with those produced by the discretized solver and that 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, in our opinion, all of the advantages mentioned earlier outweigh the potential increase in memory requirements.

3.2 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,

$$\frac{\partial w}{\partial t} + \frac{\partial f_i}{\partial x_i} = 0, \tag{9}$$

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

$$w = \begin{bmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho u_3 \\ \rho E \end{bmatrix}, \qquad f_i = \begin{bmatrix} \rho u_i \\ \rho u_i u_1 + p\delta_{i1} \\ \rho u_i u_2 + p\delta_{i2} \\ \rho u_i u_3 + p\delta_{i3} \\ \rho u_i H \end{bmatrix}. \tag{10}$$

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 that 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 application of the required boundary conditions (however complex they may be) and any artificial dissipation terms that may need to be added for numerical stability.

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

$$K_{ij} = \left[\frac{\partial X_i}{\partial \xi_j}\right], \qquad J = \det(K), \tag{11}$$

$$K^{-1}_{ij} = \left[\frac{\partial \xi_i}{\partial X_j}\right], \qquad S = JK^{-1}, \tag{12}$$

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

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

$$\frac{\partial Jw}{\partial t} + \frac{\partial F_i}{\partial \xi_i} = 0, \tag{13}$$

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

In semi-discrete form the Euler equations are,

$$\frac{\mathrm{d}w_{ijk}}{\mathrm{d}t} + \mathcal{R}_{ijk}(w) = 0, \tag{14}$$

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

The adjoint equations (7) can be re-written for this flow solver as,

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial I}{\partial w}. \tag{15}$$

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

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \psi^T \frac{\partial \mathcal{R}}{\partial x}. \tag{16}$$

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 using manual differentiation or using finite differences. Where appropriate we will use the reverse mode of automatic differentiation.

3.3 Automatic Differentiation

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

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

$$t_i = f_i\left(t_1, t_2, \ldots t_{i-1}\right), \quad i = n+1, n+2, \ldots, m. \tag{17}$$

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

$$\frac{\partial t_i}{\partial t_j} = \sum_{k=1}^{i-1} \frac{\partial f_i}{\partial t_k}\frac{\partial t_k}{\partial t_j}, \quad j = 1, 2, \ldots, n. \tag{18}$$

Using the forward mode, we choose one j and keep it fixed. We then work our way forward in the index i until we get the desired derivative.

The reverse mode, on the other hand, works by fixing i, the desired quantity we want to differentiate, and working our way backwards in the index j all the way down to the independent variables. The two modes are directly related to the direct and adjoint methods.

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.

There are automatic differentiation tools available for a variety of programming languages including fortran, c/c++ and Matlab. ADIFOR [6], TAF [10], TAMC [11] and Tapenade [12, 22] 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.

We chose to use Tapenade as it is the only non-commercial tool with support for Fortran 90. Tapenade is the successor of Odyssée [9] and was developed at the INRIA. It uses source transformation and can perform differentiation in either forward or reverse mode.

4 Implementation

To compute the desired sensitivities, we need to form the discrete adjoint equations (15), solve them and then use the total sensitivity equation (16). We will use automatic differentiation to generate code that computes each of the partial sensitivity matrices present in these equations. For the purpose of demonstration in this paper, we compute the sensitivity of the five force and moment coefficients, C_L ,C_D, C_{M_x},C_{M_y} and C_{M_z} with respect to the mesh coordinates, i.e., $I = C_i$ and $x = X(i, j, k)$.

The NSSUS solver is a new finite-difference, higher-order solver that has been developed at Stanford University under the Advanced Simulation and Computing (ASC) program sponsored by the Department of Energy [2]. It is a generic node-centred, multi-block, multi-processor solver, tested for the Euler equations, and currently being extended to the Reynolds-averaged Navier–Stokes. The finite-difference operators and artificial dissipation terms follow the work by Mattsson [19, 20] and the boundary conditions are implemented by means of penalty terms, according to the work by Carpenter [7, 8]. Despite being capable of performing computations up to eight-order accuracy, the implementation of the adjoint solver was restricted to second-order for simplicity. The extension to higher order should be straightforward to accomplish.

4.1 Brief summary of the discretization

This section summarizes the spatial discretization. Further details can be found in Mattsson [19, 20] and Carpenter [7, 8].The internal discretization is straightforward and only requires the first neighbours in each coordinate direction for the inviscid fluxes and the first and second neighbours for the artificial dissipation fluxes, see Figure 1. However the boundary treatment needs to be explained in more detail. As the finite-difference scheme only operates on the nodes of a block, one-sided difference formulae are used near block boundaries (be it a physical or an internal boundary). Consequently, the nodes on the interface of internal boundaries are multiply defined, see Figure 2.

These multiple instances of the same physical node are driven to the same value by means of a penalty term, i.e. an additional term is added to the residual $\mathcal{R}$ which is proportional to the difference between the instances. This reads

$$\mathcal{R}^i_{\text{blockA}} = \mathcal{R}^i_{\text{blockA}} + \tau(w^i_{\text{blockB}} - w^i_{\text{blockA}}) \tag{19}$$

and a similar expression for $\mathcal{R}^i_{\text{blockB}}$. In equation (19), τ controls the strength of the penalty and is a combination of a user defined parameter and the local flow conditions, see Mattsson [19] for more details. Hence, the residual of nodes on a internal block boundary is a function of its local neighbours in the block and the corresponding instance in the neighbouring block.

The boundary condition (BC) treatment is very similar to the approach described above except that the penalty state used in equation (19) is now determined by the boundary conditions.

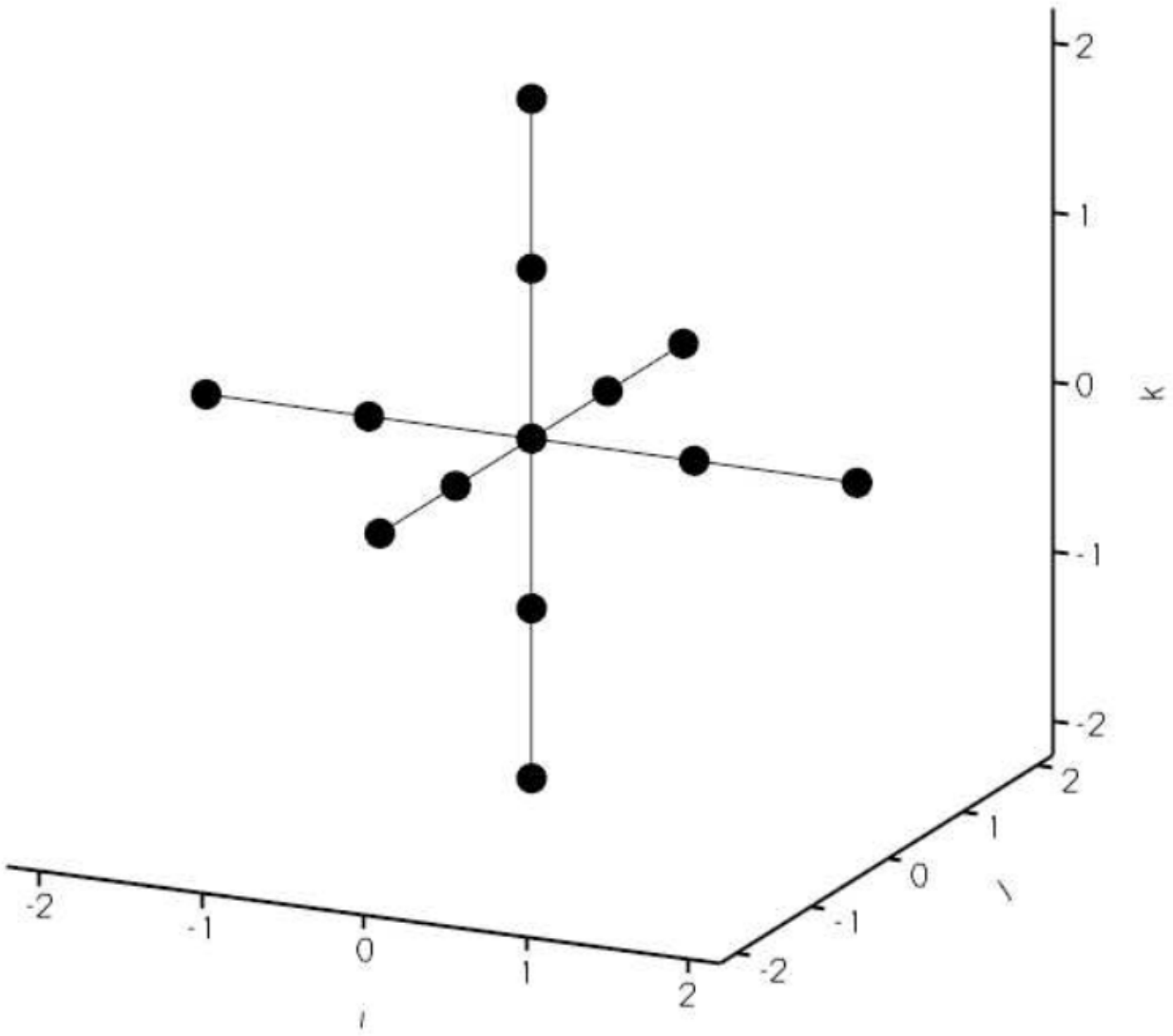

Figure 1: Stencil for the vertex-centred residual computation.

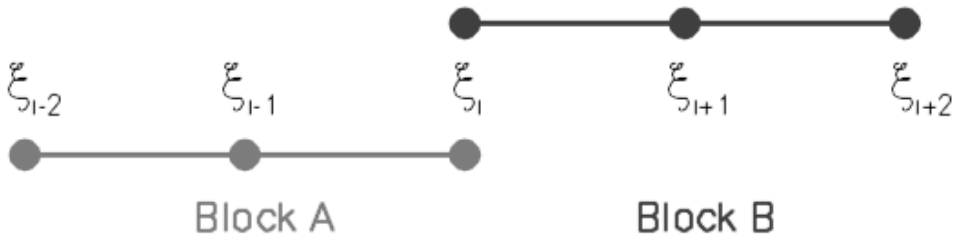

Figure 2: Block-to-block boundary stencil.

4.2 Computation of $\partial R/\partial w$

To simplify the following discussion, we define the following numbers,

N_n**:** The number of nodes 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 nodes in the stencil whose variables affect the residual of a given node. In our case, when considering inviscid and dissipation fluxes, the stencil is as shown in Figure 1 and $N_s = 13$.

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

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, their discretization and the problem boundary conditions. To compute it we need to consider the routines in the flow solver that, for each iteration, compute the residuals based on the flow variables, w. In the following discussion we note that the residual computations are carried out within the context of the NSSUS flow solver. The computation of the residual in NSSUS can be summarized as follows,

1. Compute inviscid fluxes: For our inviscid flux discretization the only flow variables, w, that influence the residual at a node are the flow variables at that node and at the six nodes directly adjacent to the node.

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

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

4. Apply boundary conditions: Additional penalty terms terms are added to enforce the BCs, see section 4.1. Note that internal block boundaries are also considered as boundaries.

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 node is only obtained at the end, when all contributions have been accounted for. Using Automatic Differentiation (AD) on this original routine containing several loops would entail several unnecessary calculations. Thus, to make the implementation of the discrete adjoint solver more efficient, it was necessary to re-write the flow residual routine such that it computed the residual for a single specified node. The re-engineered residual routine is a function with the residuals at a given node, `rAdj`, returned as an output argument and the stencil of flow variables, `wAdj`, that affect the residuals at that node provided as an input argument.

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

4.3 Computation of $\partial C_i/\partial w$

The right-hand side (RHS) of the adjoint equations (15) for the functions of interest C_D,C_L, C_{M_x},C_{M_y} and C_{M_z} are easily computed for this flow solver. Because this specific flow solver works with primitive variables $w = (\rho, u, v, w, p)$ and since, for inviscid flow, C_D, C_L, C_{M_x}, C_{M_y} and C_{M_z} are simple surface integrations of the pressure, the derivatives $\partial C_D/\partial w$ and $\partial C_L/\partial w$ are always zero except for $w_5(= p)$. Therefore, it became trivial to derive analytically the expression for these partial derivatives from the flow solver routine that calculated the functions of interest.

4.4 Adjoint Solver

The adjoint equations (15) can be re-written for the example case as follows,

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

With the two partial derivatives in this equation computed, the adjoint vector, ψ can now be computed. 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 [4, 3, 5]. PETSc is a suite of data structures and routines for the scalable, parallel solution of scientific applications modeled by PDE's. It employs the message passing interface (MPI) standard for all interprocessor communication. Using PETSc's data structures, $\partial R/\partial w$ and $-\partial C_D/\partial w$ are stored as sparse entities, whose rows are split among the processors following the same distribution as the flow data for optimal performance. Once the sparse matrices were setup, PETSc's built in, parallel, Generalized Minimum Residual (GMRES) Krylov solver was used to compute the adjoint solution.

4.5 Computation of $\partial \mathcal{R}/\partial X(i, j, k)$

The computation of the partial sensitivity of the residuals with respect to the mesh coordinates, $\partial \mathcal{R}/\partial X(i, j, k)$, was accomplished using an extension of the method used to compute the flux jacobian, $\partial \mathcal{R}/\partial w$. The stencil based approach still applies, however in this situation, the metric transformations need to be taken into account in the residual computation. Once again the first two layers of adjacent nodes in each of the three coordinate directions are required for each nodal residual. This leads to the same size and shape stencil that was present in the flux jacobian computation. Therefore, to streamline the code, the stencil based residual routine discussed above was modified to include the metric transformations, making it a function of both the states w and the grid coordinates $X(i, j, k)$. The modified routine was then re-differentiated, simultaneously, with respect to both w and $X(i, j, k)$ allowing for the computation of both sets of derivatives. Once again, the matrix is very sparse, so the PETSc data structures are used to store the matrix.

4.6 Computation of $\partial C_i/\partial X(i, j, k)$

The final partial derivative term required for the total sensitivity equation is the explicit effect of the mesh coordinates on the force and moment coefficients. This effect shows up as a change in direction of the normal vectors and associated areas for the surface nodes in the mesh. However, because the force and moment coefficients are a sum over the entire grid, the small stencil used to compute the flux jacobian is no longer valid. In this case, the stencil must become the entire mesh. On the other hand, this does provide some other benefits. For example, now we have a situation where a single reverse mode differentiation will return all of the sensitivities required for one force or moment coefficient. While this does lead to slightly higher memory costs for this computation, it does lead to very fast derivatives. Once again, due to the sparsity of the matrix, the derivative values are stored in PETSc.

4.7 Total Sensitivity Equation

The total sensitivity (16) in this case can be written as,

$$\frac{\mathrm{d}C_i}{\mathrm{d}X(i, j, k)} = \frac{\partial C_i}{\partial X(i, j, k)} + \psi^T \frac{\partial \mathcal{R}}{\partial X(i, j, k)}, \tag{21}$$

Table 1: Mesh coordinate sensitivity verification: $\frac{dI}{dX(i,j,k)}$ for hypersonic test case, block 3, index 2,7,13.

Control Node	Cost Fun. J	Adjoint	Finite-Diff. (1×10^{-4})	Δ
1	C_L	-0.001015	-0.000910	*13.7%*
	C_D	-0.000218	-0.000230	*5.02%*
Control Node	Cost Fun. J	Adjoint	Finite-Diff. (1×10^{-3})	Δ
1	C_L	-0.001015	-0.001051	*0.39%*
	C_D	-0.000218	-0.000222	*1.37%*

where the objective function C_i represents the i^{th} coefficient of interest and the independent variable $X(i,j,k)$ represents the mesh coordinates at location i,j,k. With all four of the partial derivatives matrices computed, and the adjoint equation solved, all that remains is to multiply the terms together as shown in equation 21. This will leave us with a single vector of length $3 \times N_n$ for each force or moment coefficient of interest.

5 Results

The following results are based on three distinct test cases. The first test case, used to verify the sensitivity results, is a half body model of simple hypersonic vehicle (figure 3). This is a six block test case and has been run at Mach=3.0. The remaining two test cases, shown in figures 4 and 5 were used mostly for timing purposes. The first is a more complex hypersonic vehicle, intended as an analog to the NASA X-43 test plane, simulated at Mach = 5. While the last test case is an oblique flying wing modeled at Mach = 1.5. All cases use Euler wall boundary conditions for the wing surface. Figures 6 through 11, show the meshes and flow solutions for the three test cases.

Accuracy results for the hypersonic test case are shown in Table 1. This comparison to a forward finite difference shows acceptable accuracy. The relative error is less than 1.5% for the larger step size. However, because of the sensitivity of the finite difference result to step size, it is uncertain which of the two results is more correct. We plan to compare these results against complex step results in the near future and expect this comparision to show an accuracy of 7 or more digits. This expectation is based on the results of a previous comparison done with a single block version of the ADjoint implemented on the SUmb flow solver [18].

The timing results for the oblique wing and X-43 test cases are shown in Table 2. As can be seen for both cases, the ADjoint solution is less expensive than the flow solution, varying from 2/3 to 1/50 of the flow solution time, depending on the test case. The large variation in this ratio come largely from the fact that the flow solver is still under development and as such is not fully optimized. Also, the timing results are affected by the fact that the code uses and explicit time integration scheme with the corresponding limitations on CFL number.

When it comes to comparing the performance of the various components in the adjoint solver, 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 are the computation of the flux Jacobian and the computation of the mesh partial, $\partial\mathcal{R}/\partial X(i,j,k)$. When one takes into consideration the number of terms in these matrices, spending less than 25% of the flow solution time in this computation is quite impressive.

Also, while we have not done rigorous testing on the memory requirements of this code, preliminary observations indicate that the memory required for the ADjoint code is approximatly ten times that required for the original flow solver. Given the pattern of use on most parallel computers, this is well within the acceptable limits for an adjoint code.

The last set of results shown here are the lift and drag sensitivities of the oblique wing case, shown in figures 12 and 13.

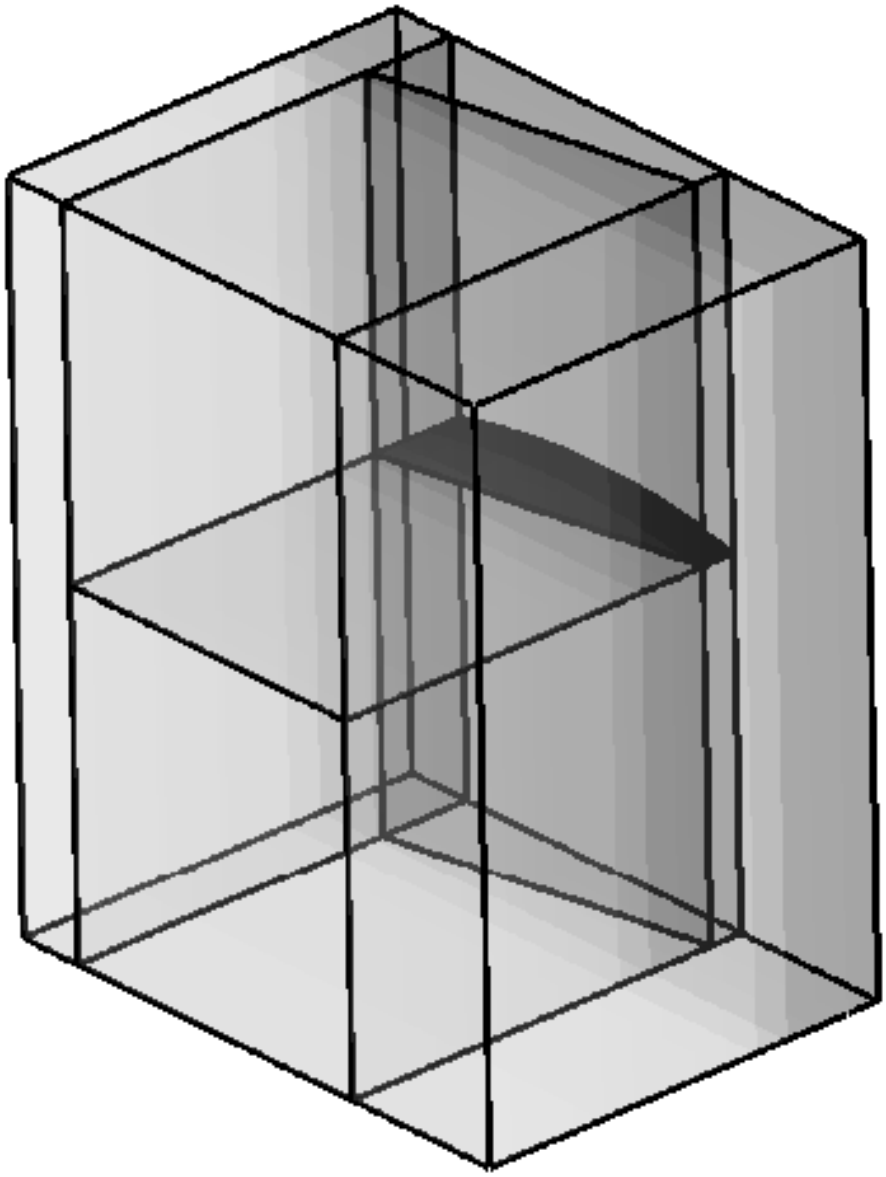

Figure 3: Hypersonic half-body

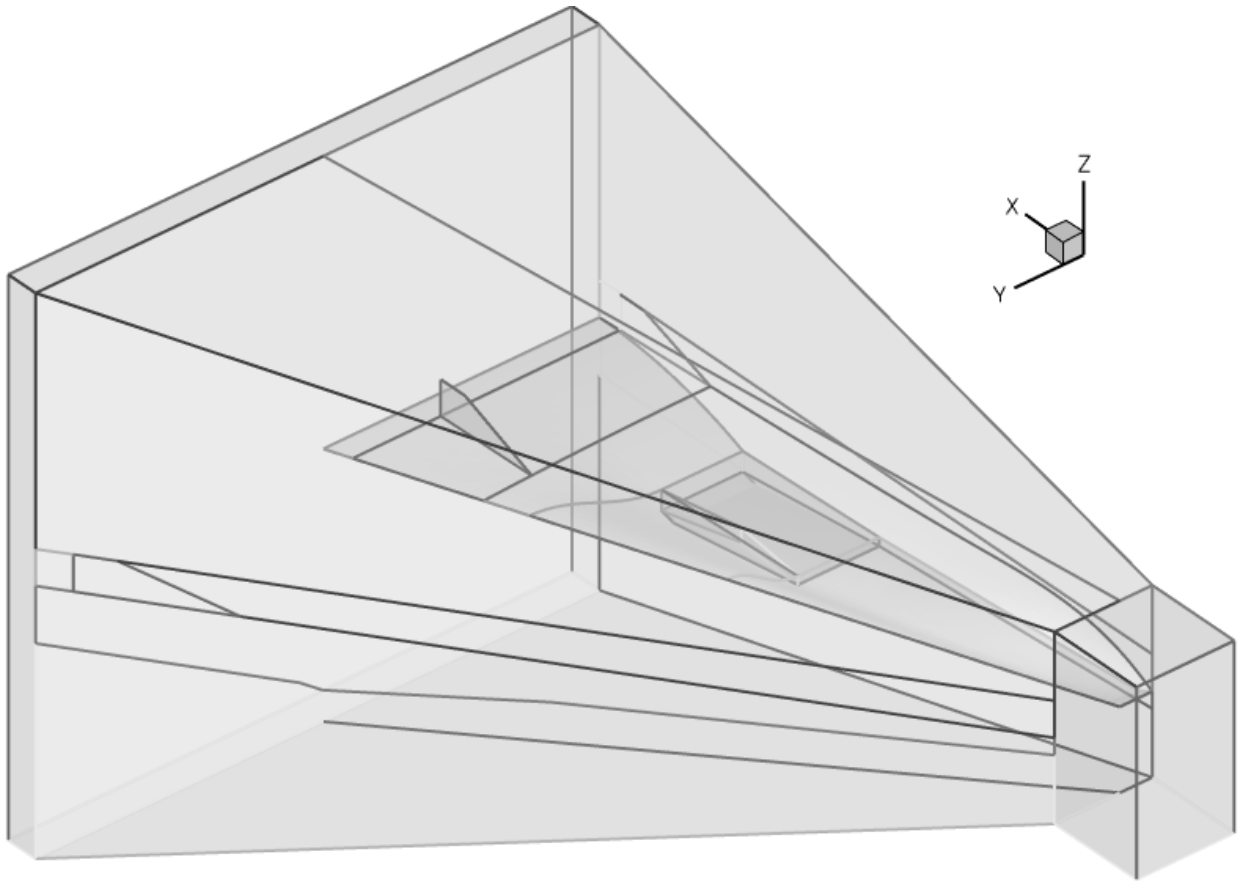

Figure 4: X-43 Analog

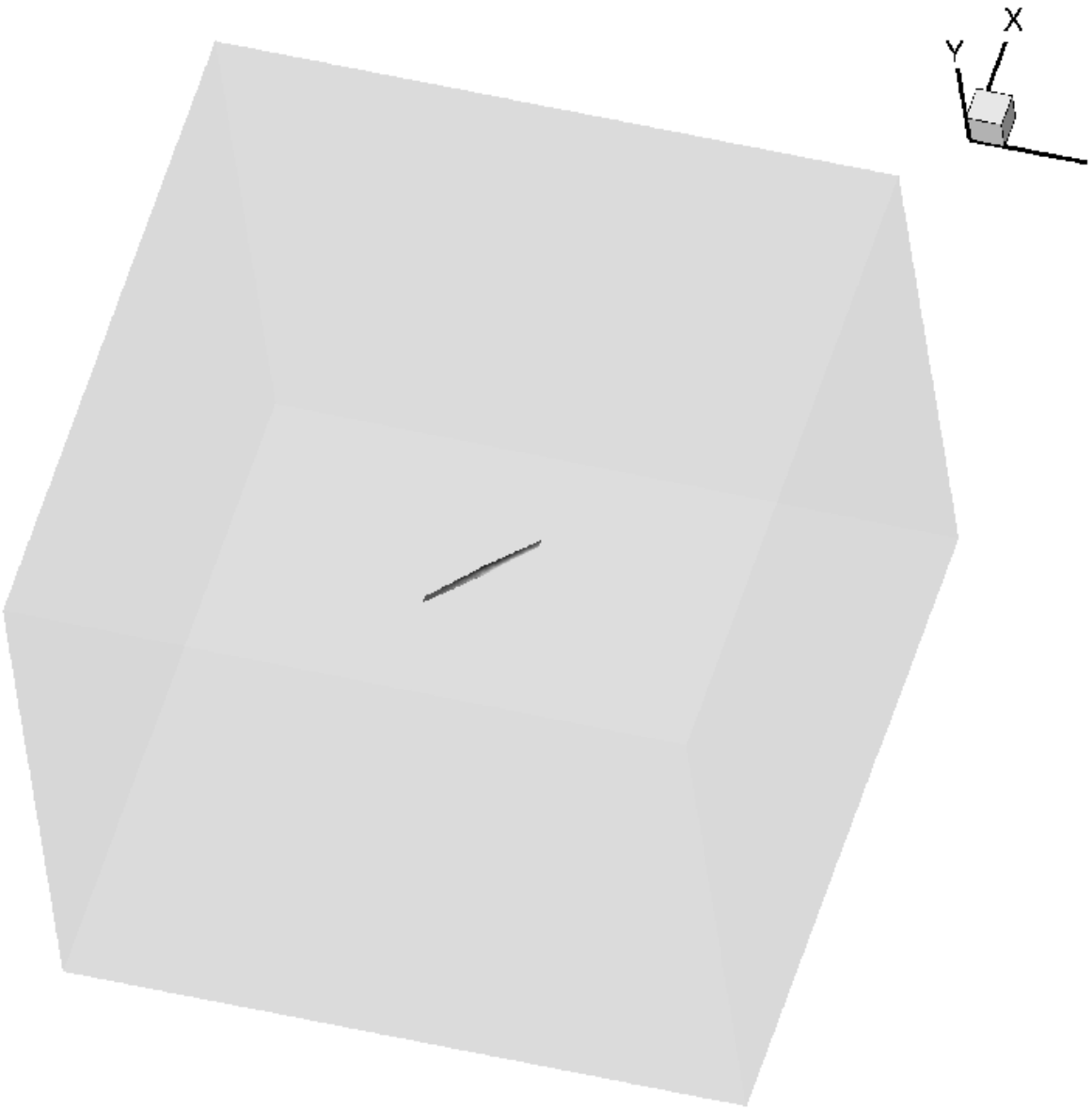

Figure 5: Oblique flying wing

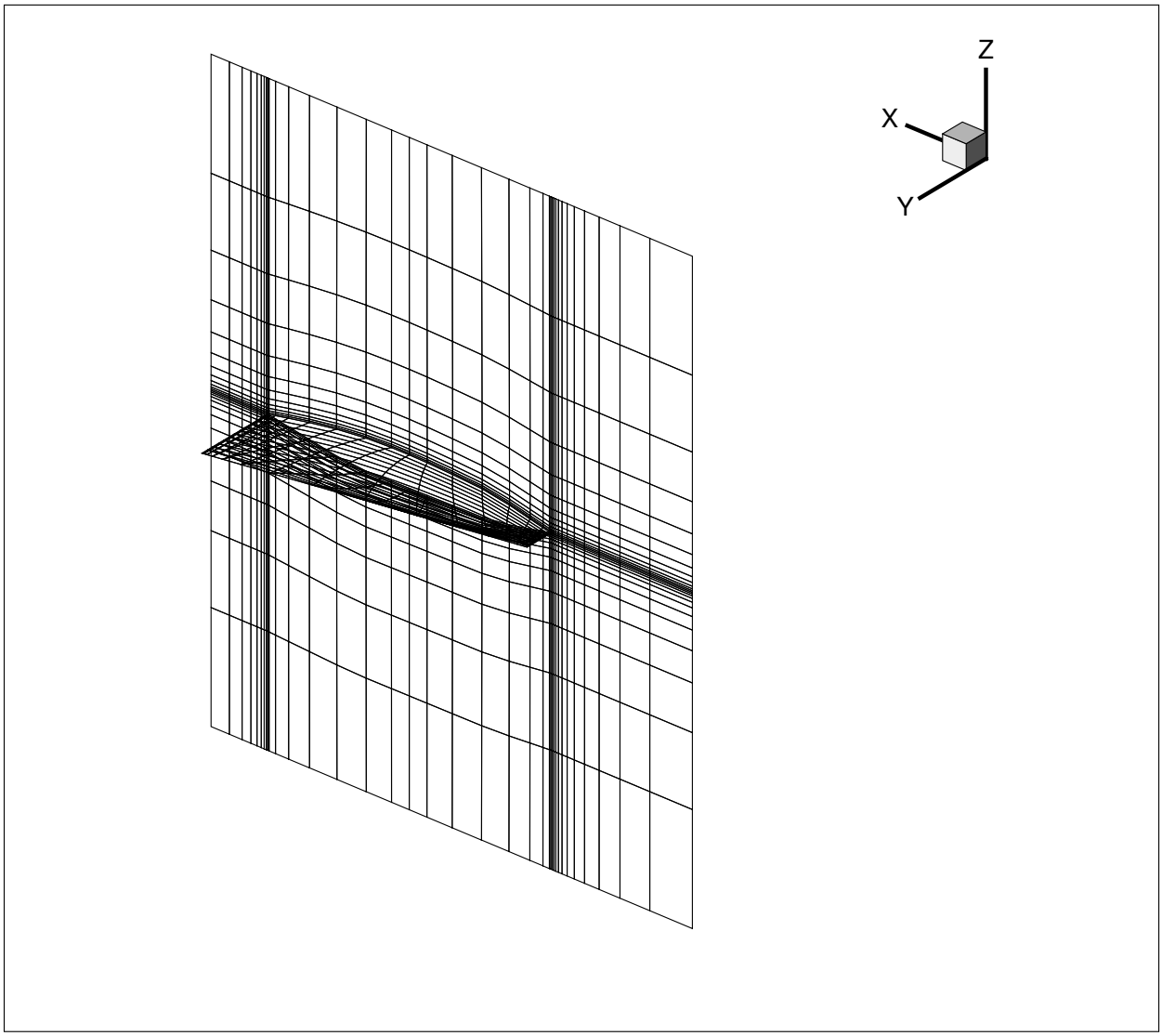

Figure 6: Mesh for hypersonic test case

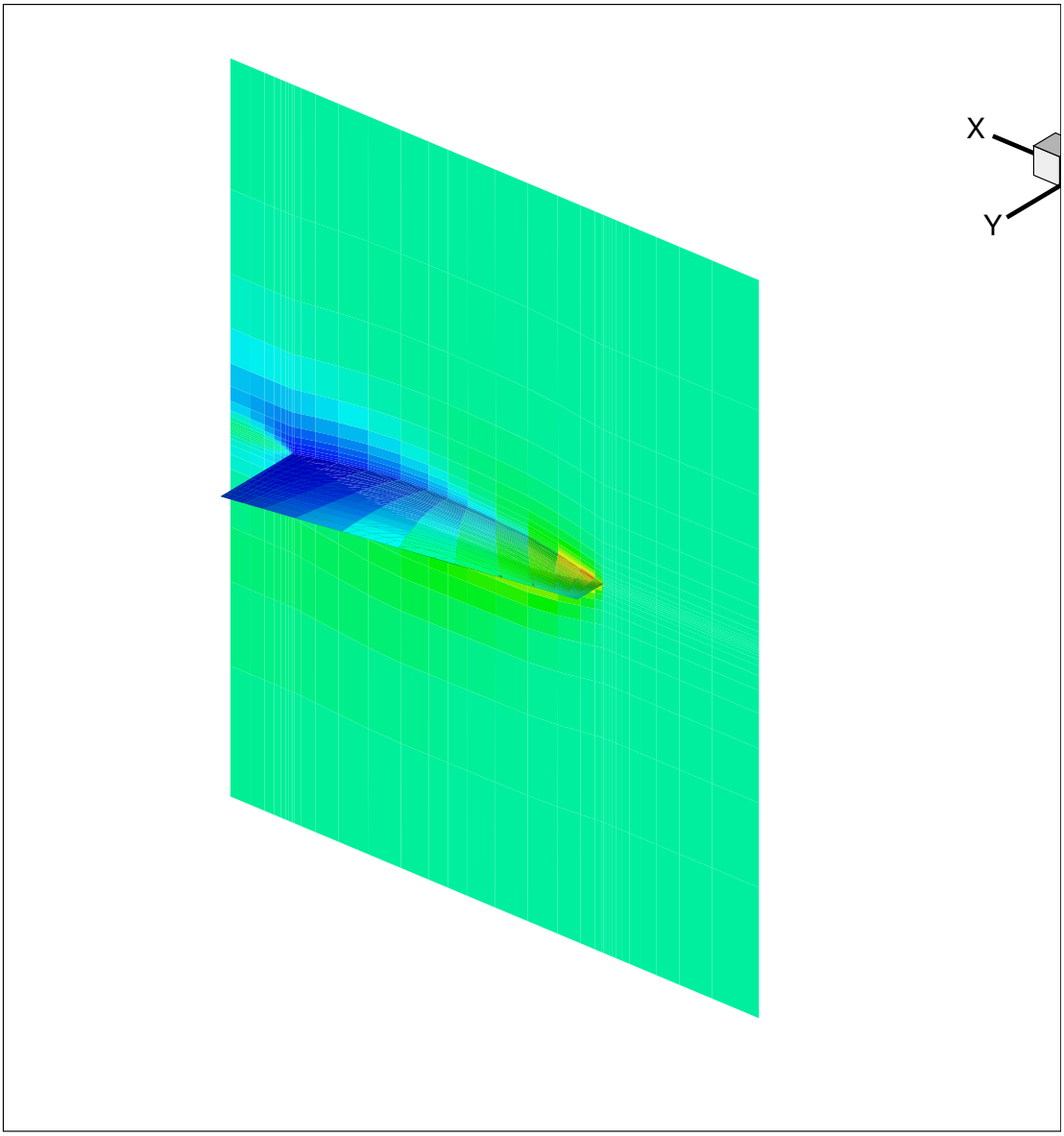

Figure 7: Contour plot of pressure

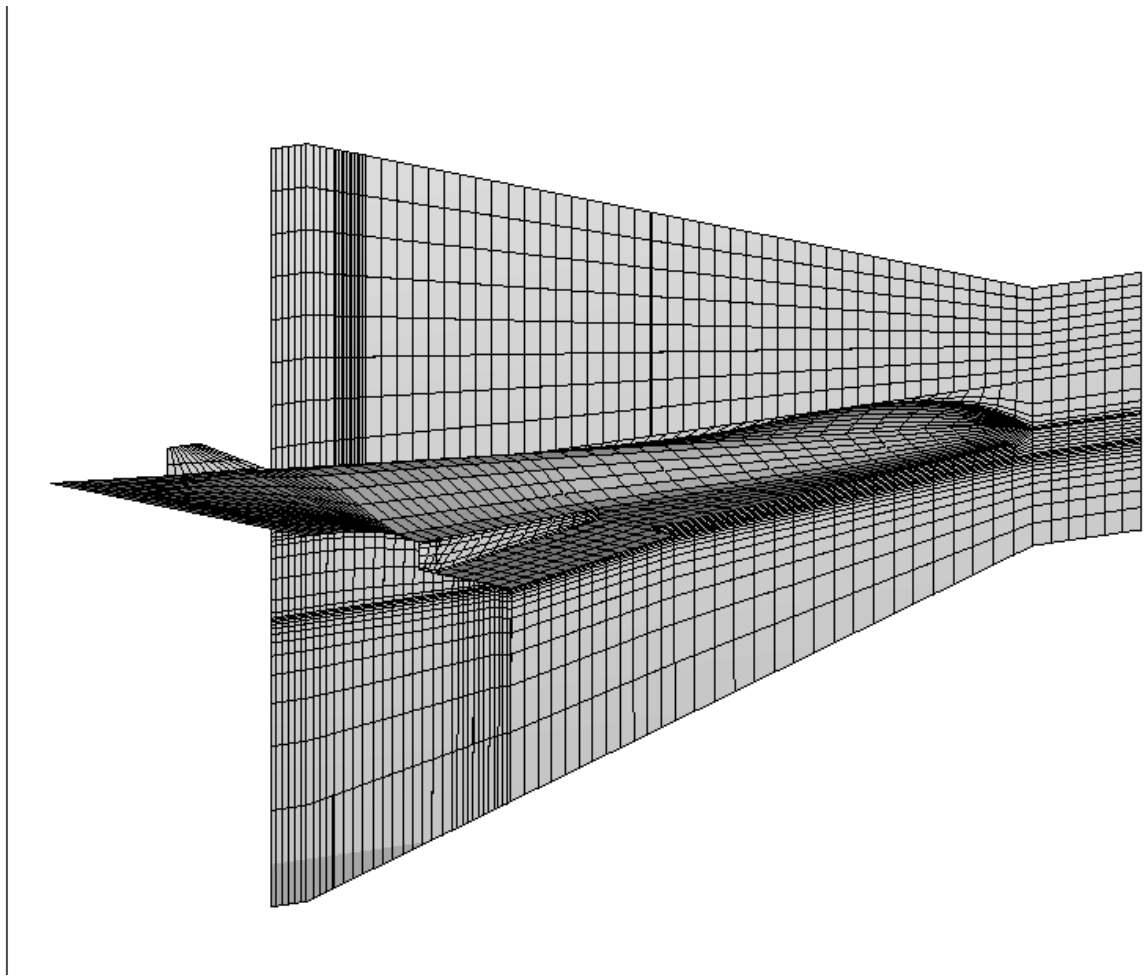

Figure 8: Mesh for hyperplane test case

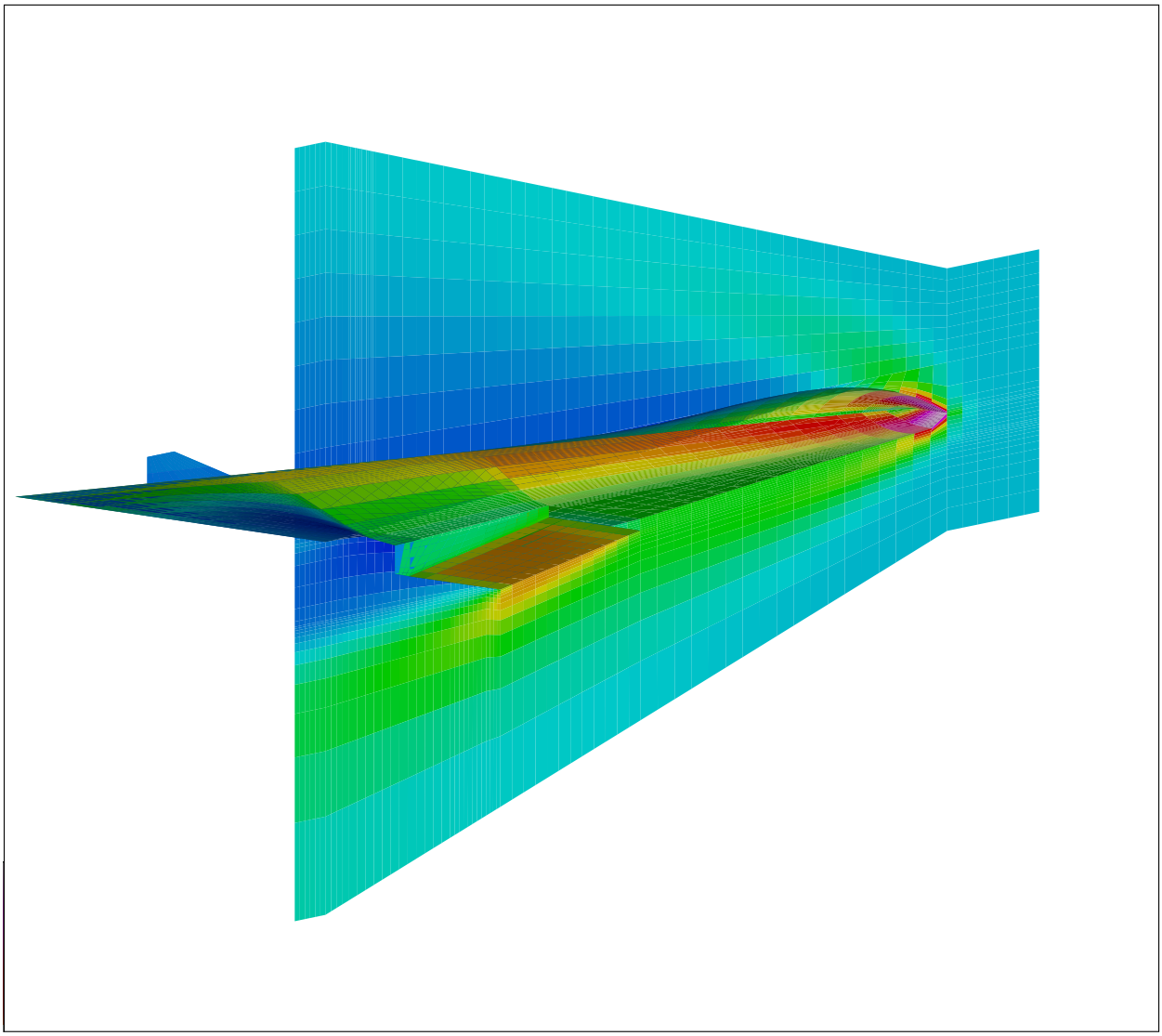

Figure 9: Contour plot of pressure

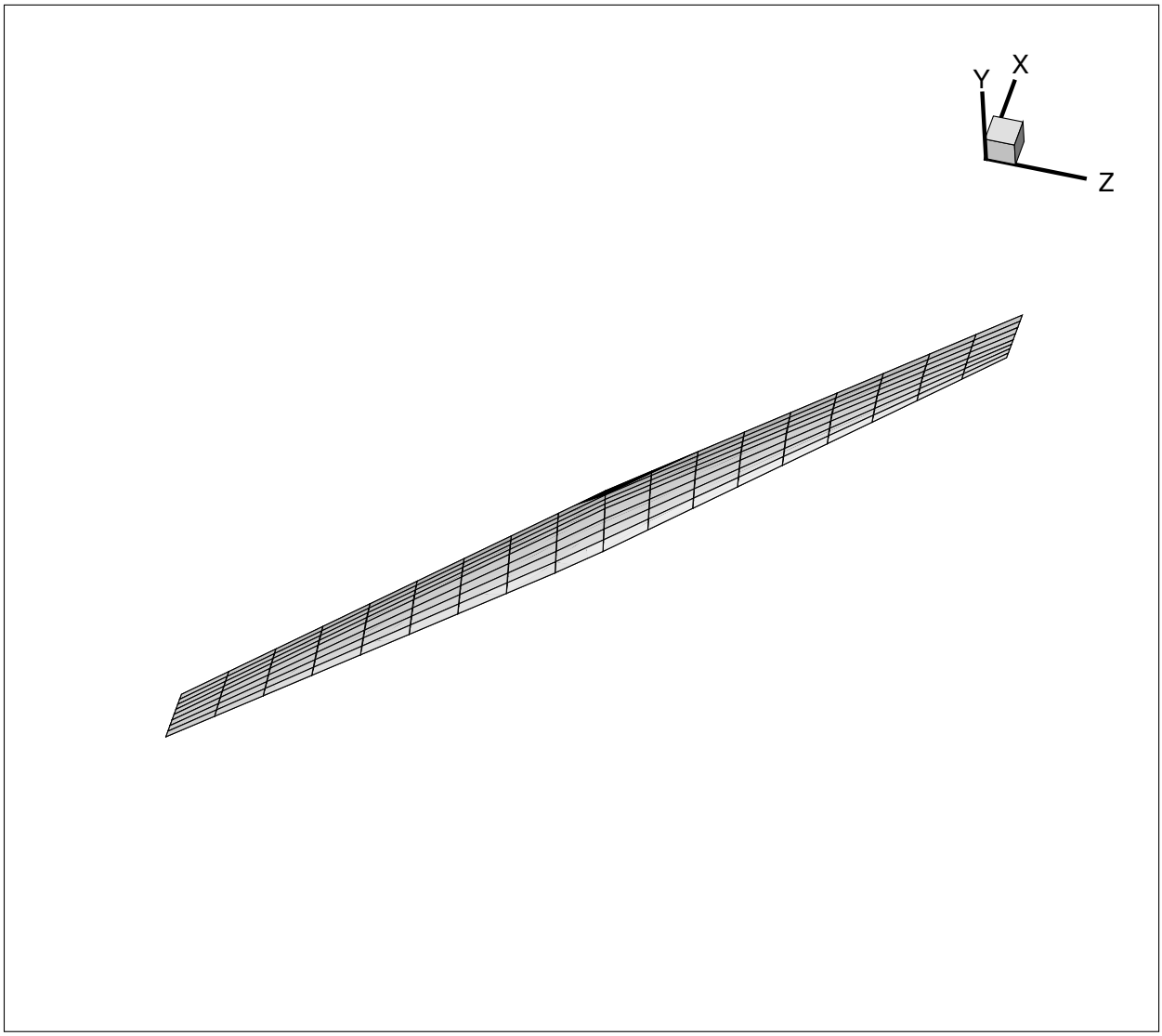

Figure 10: Mesh for oblique wing test case

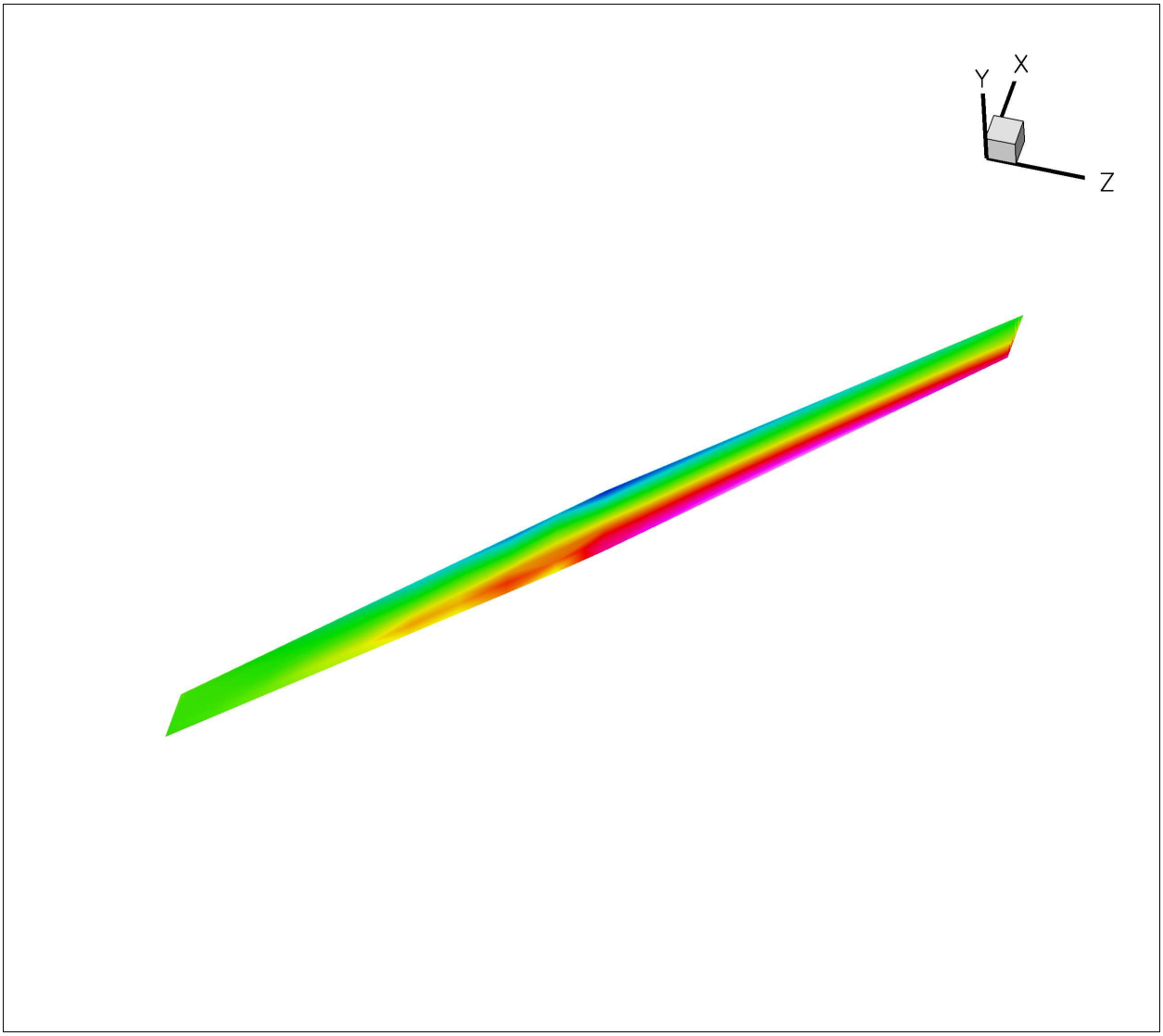

Figure 11: Contour plot of pressure

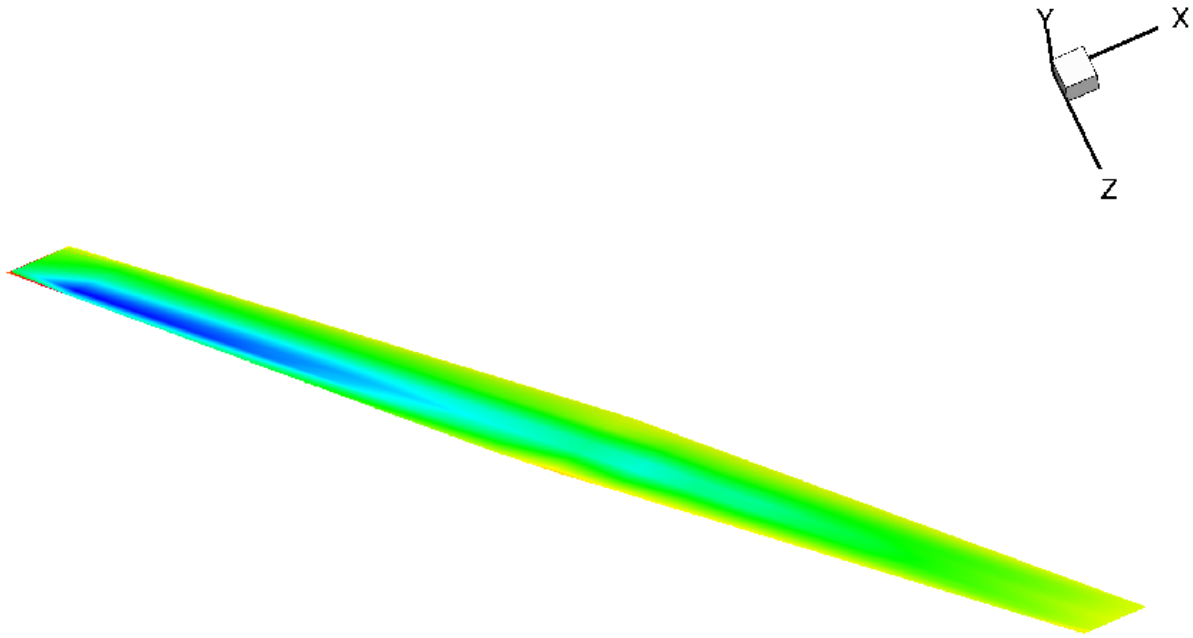

Figure 12: Lift sensitivities

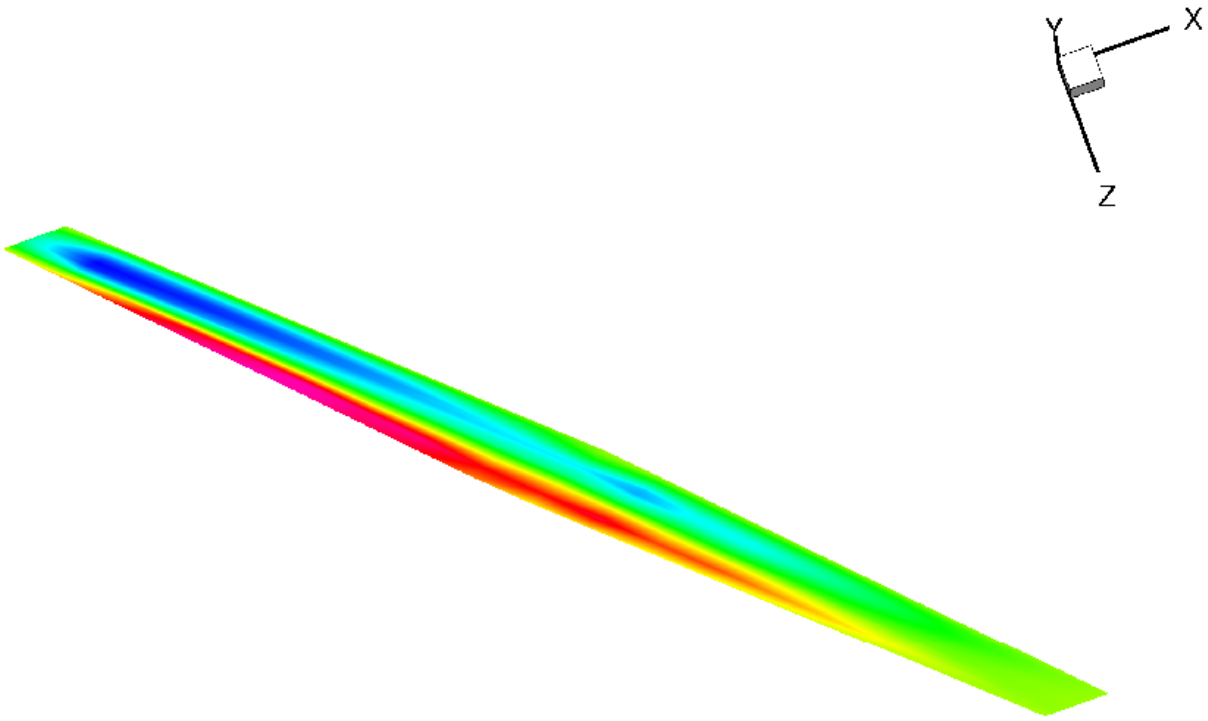

Figure 13: Drag Sensitivities

	Oblique Wing	Hyperplane
No.Processors	1	2
No. Nodes	21820	50776
Flow solution	63.98	1771.29
ADjoint	40.17	46.58
Breakdown:		
Setup PETSc Variables	0.08	0.07
Compute flux Jacobian	7.80	15.85
Compute grid partial	15.76	20.76
Compute RHS	0.00	0.00
Solve the adjoint equations	16.33	9.62
Compute the total sensitivity	0.20	0.28

Table 2: ADjoint computational cost breakdown (times in seconds)

As can be seen in the figures, there is a higher sensitivity to both lift and drag on the forward swept portion of the wing. Also, as we would expect, there is a high sensitivity to drag along certain portions of the wing leading edge. These are all characteristics that should lead to interesting optimization results.

6 Conclusions

In this work we have applied the ADjoint method to the NSSUS flow solver to generate the mesh coordinate sensitivities required to perform aerodynamic shape optimization. We have validated the resulting sensitivities through comparison to finite difference results, though we expect to do a more thorough comparison of the results against the complex step method in the near future. The implementation has also been shown to be very efficient, with the total ADjoint solution taking less time than the flow solver. Finally, we have shown some overall sensitivity distributions for the oblique wing case, as an example of the results that the ADjoint implementation on NSSUS can generate. These results also form the basis of the sensitivities required for aerodynamic shape optimization of an oblique wing.

Acknowledgments

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] W. Andrews, A. Sim, R. Monaghan, L. Felt, T. McMurtry, and R. Smith. Ad-1 oblique wing aircraft program. SAE Technical Paper Series 801180, 1980.

[2] ASC. Advanced simulation and computing. http://www.llnl.gov/asc, 2007.

[3] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

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

[5] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[6] A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.

[7] M. H. Carpenter, D. Gottlieb, and S. Abarbanel. Time-stable boundary conditions for finite-difference schemes solving hyperbolic systems: Methodology and application to high-order compact schemes. *Journal of Computational Physics*, 111(2):220–236, Apr. 1994.

[8] M. H. Carpenter, J. Nordström, and D. Gottlieb. A stable and conservative interface treatment of arbitrary spatial accuracy. *Journal of Computational Physics*, 148(2):341–365, Jan. 1999.

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

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

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

[12] L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004.

[13] A. Jameson. Aerodynamic design via control theory. *Journal of Scientific Computing*, 3(3):233–260, sep 1988.

[14] R. T. Jones. The oblique wing - aircraft design for transonic and low supersonic speeds. *Acta Astronautica*, 4:99 – 109, 1977.

[15] J. R. R. A. Martins, J. J. Alonso, and J. J. Reuther. High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530, 2004.

[16] J. R. R. A. Martins, J. J. Alonso, and J. J. Reuther. A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design. *Optimization and Engineering*, 6(1):33–62, March 2005.

[17] J. R. R. A. Martins, J. J. Alonso, and E. van der Weide. An automated approach for developing discrete adjoint solvers. In *Proceedings of the 2nd AIAA Multidisciplinary Design Optimization Specialist Conference*, Newport, RI, 2006. AIAA 2006-1608.

[18] J. R. R. A. Martins, C. A. Mader, and J. J. Alonso. Adjoint: An approach for rapid development of discrete adjoint solvers. In *Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Portsmouth, VA, 2006. AIAA 2006-7121.

[19] K. Mattsson and J. Nordström. Summation by parts operators for finite difference approximations of second derivatives. *Journal of Computational Physics*, 199(2):503–540, Sept. 2004.

[20] K. Mattsson, M. Svärd, and J. Nordström. Stable and accurate artificial dissipation. *Journal of Scientific Computing*, 21(1):57–79, Aug. 2004.

[21] M. Nemec and D. W. Zingg. Newton-krylov algorithm for aerodynamic design using the navier-stokes equations. *AIAA Journal*, 40(6):1146–1154, 2002.

[22] V. Pascual and L. Hascoët. Extension of TAPENADE towards Fortran 95. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, 2005.

[23] O. Pironneau. On optimum design in fluid mechanics. *Journal of Fluid Mechanics*, 64:97–110, 1974.

[24] J. Reuther, J. J. Alonso, A. Jameson, M. Rimlinger, and D. Saunders. Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers: Part I. *Journal of Aircraft*, 36(1):51–60, 1999.

[25] J. Reuther, J. J. Alonso, A. Jameson, M. Rimlinger, and D. Saunders. Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers: Part II. *Journal of Aircraft*, 36(1):61–74, 1999.

[26] A. J. D. Velden and I. Kroo. The aerodynamic design of the oblique flying wing supersonic transport. NASA Contractor Report 177552, 1990.