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

by

Charles Alexander Mader

A thesis submitted in conformity with the requirements
for the degree of Masters of Applied Science
Institute for Aerospace Studies
University of Toronto

Abstract

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

Charles Alexander Mader
Masters of Applied Science
Institute for Aerospace Studies
University of Toronto
2007

The ADjoint is a hybrid sensitivity analysis method that takes advantage of the best aspects of both Automatic Differentiation (AD) and the semi-analytic adjoint method. The key feature of the ADjoint method is that it selectively uses AD to calculate the required partial derivative terms in the discrete adjoint sensitivity equations. This selective use of AD significantly reduces the computational cost and memory overhead of using AD. Further, because of the use of AD, the method can provide exactly consistent derivatives for arbitrarily complex governing equations and boundary conditions. In the following work, the ADjoint method is applied to two three-dimensional Computational Fluid Dynamics (CFD) solvers. The implementation in the first solver is simply a proof of concept, while the implementation for the second solver is complete and provides all the derivatives required for aerodynamic shape optimization. The resulting ADjoint sensitivities are compared with complex-step derivatives to establish their accuracy.

Acknowledgements

First and foremost, I would like to thank my adviser, Professor Joaquim Martins, for all his support and encouragement throughout this project. He was always willing to make time to discuss the project and provide insights, particularly when major road blocks presented themselves. Without his support, the project would not have been nearly the success that it has turned out to be.

Secondly, I would like to thank Dr. Andre C. Marta for his contributions to the project. Without the groundwork he provided, the development shown in this thesis would not have been possible.

Finally, I would like to thank my colleagues, particularly Ian Chittick and Graeme Kennedy, who provided valuable advice and who's constant intervention helped me keep my sanity during this work.

Financial support for this work was provided by the Natural Sciences and Engineering Research Council of Canada and the University of Toronto.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As we push forward into the future of aircraft design, the requirements on aircraft designers are becoming increasingly stringent. In addition to the traditional requirements of designing a safe aircraft with good handling characteristics, the designer is also faced with the constant demand of passengers who wish to fly farther, faster, for less money. Further, the areas of environmental impact and noise pollution are also starting to have a significant impact in aircraft design. These more restrictive sets of competing design goals are causing the aircraft designer to explore new design methods in order to provide the best overall aircraft performance for a given situation. One of these new tools is design optimization. More specifically, in the particular case of aircraft design, Aerodynamic Shape Optimization. Aerodynamic shape optimization is a process whereby numerical analysis tools such as Computational Fluid Dynamics (CFD) are used in conjunction with numerical search tools – also known as optimizers – to search a given design space for the best solution for a given set of requirements. One class of tools that can be used for this type of optimization is gradient-based search algorithms. However, in order for these methods to work well, an efficient method for calculating the gradients for the numerical analysis is required. Finding a method to efficiently compute the gradients of large, complex numerical codes is the major focus of this work.

1.1 Motivation

This work discusses the ADjoint, an approach for the rapid development of discrete adjoint solvers. Adjoint solvers are a key component in the development of high-fidelity, gradient based optimization algorithms. These are, in turn, a key tool in the implementation of aerodynamic shape optimization methods. In order to take advantage of the information provided by a high-fidelity analysis, it is generally desirable to include a large amount of design flexibility when conducting a high-fidelity shape optimization. This typically requires a large number of design variables to allow sufficient variability in all components of the shape of the aircraft being optimized. However, because of this large number of design variables, the optimization algorithm used needs to be able to handle large numbers of design variables efficiently. While numerous examples exist of gradient-free algorithms being used for shape optimization, it is generally accepted that gradient-free algorithms become infeasible for more than 20 or 30 design variables. This leaves gradient based algorithms as the only option for large scale design optimization. In order to conduct large-scale gradient based optimization, an efficient sensitivity analysis method is required. Otherwise, the computational time spent evaluating derivatives will far outweigh all the other computational costs in the optimization.

This is where adjoint methods become useful. While the computational cost of methods like finite differences and the complex-step scale with the number of design variables [20, 30], the cost of the adjoint method has been shown to be essentially independent of the number of design variables, thus allowing for a large number of shape derivatives to be calculated efficiently [16, 20, 30]. However, as will be discussed in section 2.3, the implementation of adjoint methods in three-dimensional, high-fidelity, Computational Fluid Dynamics (CFD) codes has proved to be quite difficult. As a result, the use of adjoint methods in this field is still not particularly widespread. This is the motivation behind the ADjoint approach. The ADjoint approach takes advantage of the best features of both Automatic Differentiation(AD) and adjoint methods to provide a straightforward, yet accurate and efficient, method for implementing adjoint methods.

1.2 Research Overview

In the following sections, the ADjoint approach will be introduced and discussed. The early sections introduce the ADjoint as well as the basic concepts of automatic differentiation and adjoint methods upon which the ADjoint is based. Section 2.4 discusses a preliminary implementation of the ADjoint conducted on the first flow solver and shows conclusively the advantageous properties of the ADjoint, namely the accuracy and computational efficiency of the method. The later sections discuss the full implementation of ADjoint sensitivities in the aerodynamic framework being constructed to conduct aerodynamic shape optimization. Section 4.2 discusses the full implementation of the ADjoint method on the second code, including the parallelization and multiblock aspects of the implementation. Finally, section 4.3 shows how the flow solver ADjoint sensitivities can be combined with geometry and grid manipulation tools to compute the total aerodynamic shape sensitivities required for shape optimization.

1.3 Contributions

The main contribution of this work is the idea of a hybrid approach between automatic differentiation and adjoint methods. By combining the two together, the benefits of both techniques are realized. The independence of computational cost from the number of design variables typically associated with adjoint methods is attained, while the accuracy and extensibility associated with AD are also achieved. Further, each technique tends to eliminate the deficiencies of the other. In one case, the use of AD eliminates the tedious process involved in generating the partial derivatives required for the adjoint equations. In the other, using the adjoint method to reduce the scope of the AD significantly reduces the implementation difficulties and memory costs frequently associated with AD. The following research outlines the steps required to implement sensitivities using the ADjoint approach and shows that it is both accurate and efficient.

Chapter 2

The ADjoint Approach

For gradient based optimization of complicated functions, determining the derivatives of the objective function can be a significant bottleneck in the optimization process. Traditional methods such as finite differencing can be relatively simple to implement, but tend to sacrifice speed and accuracy as a result [22, 30]. Consider the general forward finite difference formula,

$$\frac{\mathrm{d}I(x)}{\mathrm{d}x} = \frac{I(x+h) - I(x)}{h}. \tag{2.1}$$

In this case, it is quite clear that the derivative with respect to each successive x requires a further evaluation of the function $f(x+h)$. Thus, for large numbers of design variables, the cost of evaluating the derivatives is extremely high. Further, the accuracy of the method is limited by opposite trends in truncation and round off error [22]. While the truncation error decreases with step size, round off error, caused by subtractive cancellation, increases as the step size decreases. Thus, there is a limit, greater than machine zero, below which the error cannot be reduced [22]. Further, the minimum error does not occur at a fixed step size, thus it is very difficult to achieve the minimum theoretical error with this method.

Both of these issues are addressed by adjoint methods. Adjoint methods provide high accuracy and have a computational cost that is essentially independent of the number of design variables present [16, 30]. While this sounds like the ultimate solution to the sensitivity problem, in reality, things are not so simple. In many cases, for example the turbulence models in modern CFD codes, the partial derivatives required to compute an adjoint solution are prohibitively complex to derive. As a result, assumptions are often made to simplify the derivatives, which results in a reduction in the accuracy of the derivatives provided by the method. An excellent discussion on this topic is presented by Dwight and Brezillion [9]. The key then, is finding a method with which one can compute the partial sensitivities required by the adjoint method. The ADjoint approach provides a robust method for dealing with this issue. By using automatic differentiation to calculate the partial sensitivities in the adjoint equations, the efficiency of an adjoint method is maintained, while allowing the method to be extended to codes of arbitrary complexity without difficulty. The use of automatic differentiation also provides exactly consistent partial derivatives, which leads to numerically exact total sensitivities.

In the following section, the ADjoint method is developed. The adjoint equations are introduced, followed by a brief explanation of automatic differentiation. Then, the concept of the ADjoint, which combines the two, is explained. To conclude the development, exactly consistent derivatives, developed using the ADjoint on the first flow solver, are shown.

2.1 Adjoint Equations

To derive the adjoint system of equations, consider a generic, single discipline case. Further, consider a single objective function, that is a function of both the design variables x as well as the system state variables w. In the generic case, let this be

$$I = I(x, w). \tag{2.2}$$

Also, define a set of governing equations,

$$\mathcal{R}\left(x, w\left(x\right)\right) = 0, \tag{2.3}$$

which represent the physics of the system and define the states of the system, w, when they are satisfied. Next, consider the total derivatives of both the objective function I and the governing equations $\mathcal{R}$. These are written as follows:

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \frac{\partial I}{\partial w}\frac{\mathrm{d}w}{\mathrm{d}x} \tag{2.4}$$

$$\frac{\mathrm{d}\mathcal{R}}{\mathrm{d}x} = \frac{\partial \mathcal{R}}{\partial x} + \frac{\partial \mathcal{R}}{\partial w}\frac{\mathrm{d}w}{\mathrm{d}x} = 0. \tag{2.5}$$

Unfortunately, both of these equations contain the total derivative $\mathrm{d}w/\mathrm{d}x$, which requires the governing equations to be satisfied for every x. However, as is expressed in equation (2.5), the total derivative of the residuals with respect to the design variables, i.e. the derivative including a solution of the system for each new design variable, will be zero. The states will be modified as the system is solved to ensure that this is the case. Because of this fact, the system in equation (2.5) can be rearranged to isolate $\mathrm{d}w/\mathrm{d}x$ as follows,

$$\frac{\mathrm{d}w}{\mathrm{d}x} = -\left[\frac{\partial \mathcal{R}}{\partial w}\right]^{-1} \frac{\partial \mathcal{R}}{\partial x}. \tag{2.6}$$

Having done this, the result from equation (2.6) can be substituted back into equation (2.4) to create an alternate form of the total sensitivity equation as shown in equation (2.7)

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} - \frac{\partial I}{\partial w}\left[\frac{\partial \mathcal{R}}{\partial w}\right]^{-1} \frac{\partial \mathcal{R}}{\partial x}. \tag{2.7}$$

From this equation, one can generate either of the two typical semi-analytic methods used. To get the direct method, one solves the system generated by the last two terms, which is analogous to solving the system of equations represented in equation (2.6). This method requires solving one linear system for each x of interest, which causes the cost of the method to scale with the number of design variables, N_x. The alternative is to solve the system generated by the second and third terms in equation (2.7), which has the form,

$$\psi = -\frac{\partial I}{\partial w}\left[\frac{\partial \mathcal{R}}{\partial w}\right]^{-1}. \tag{2.8}$$

This approach is commonly referred to as the adjoint method, which is more traditionally written as,

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^{T} \psi = -\frac{\partial I}{\partial w}. \tag{2.9}$$

In this case, one has to solve a linear system for each I, so the cost of computing derivatives scales with the number of objective functions, N_I, rather than the number of design variables N_x. The selection of which semi-analytic method to use, adjoint or direct, comes down to the relative number of design variables, N_x, and objective functions, N_I. For cases with large numbers of objective functions, the direct method is obviously more efficient, while for cases with many design variables, the adjoint method is better. In the case of aerodynamic shape optimization there are typically relatively few objective values, usually C_D and C_L. However there can be on the order of hundreds or thousands of design variables describing the shape of the design. Thus, for aerodynamic shape optimization, the adjoint method will generally provide significant computational advantages. Thus, for the following work, equation (2.7) will be solved using the adjoint method with the equations taking the following form:

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial I}{\partial w}, \tag{2.10}$$

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \psi^T \frac{\partial \mathcal{R}}{\partial x}, \tag{2.11}$$

where ψ is the *adjoint vector* and $\mathrm{d}I/\,\mathrm{d}x$ is the vector of total sensitivities required for optimization.

2.2 Automatic Differentiation

The process of automatic differentiation, which is also known as computational differentiation or algorithmic differentiation, has been around for many years. Its earliest implementation dates back to the early 1960's, for example the work published by Wengert in 1964 [35]. Essentially, it is the process of systematically applying the chain rule of differentiation to a computer code to accumulate the value of the derivative as the value of the function is being computed or, as Rall and Corliss put it,

> " AD is the systematic application of the familiar rules of calculus to computer programs, yielding programs for the propagation of numerical values of first, second, or higher derivatives." [29]

By treating the code in this manner, even very complex derivatives can be broken down into components that are simple enough to treat with very basic derivative methods, the total effect of which can be determined by the combination of the simple derivatives. A more detailed description of the method is given below.

Consider a general problem with N_x independent variables, x, and N_y dependent variables, y. Let the independent variables take the values $t_1, t_2, \dots t_{N_x}$. In the case of aerodynamic shape optimization these would be the design variables describing the shape of the aerodynamic body. Then, let the dependent variables take the values $t_{N_x+1}, t_{N_x+2}, \dots, t_m, t_{m+1}, \dots, t_{m+N_y}$. Where t_{N_x+1} to t_m are the intermediate variables of the algorithm and t_{m+1} to t_{m+N_y} are the output variables of the algorithm. Given these definitions, one can write write the value of the calculation at any stage of the algorithm as follows,

$$t_i = f_i\left(t_1, t_2, \dots t_{i-1}\right), \quad i = N_x+1, N_x+2, \dots, m+N_y. \tag{2.12}$$

Given this definition of the function at any stage in the algorithm, its derivative can be formulated using the multivariable form of the chain rule as expressed in equation (2.13).

$$\frac{\partial t_i}{\partial t_j} = \sum_{k=j}^{i-1} \frac{\partial f_i}{\partial t_k}\frac{\partial t_k}{\partial t_j}, \quad j = 1, 2, \ldots, n \tag{2.13}$$

To demonstrate this, consider the simple problem defined by equations (2.14) and (2.15).

$$y_1 = x_1^2 x_2 + \sin x_3 \tag{2.14}$$

$$y_2 = x_1 x_2 x_3 + e^{x_2} \tag{2.15}$$

This example has two dependent output variables, y_j, and three independent variables, x_i. Next, using the notation of Rall and Corliss [29], one can write the evaluation of y_1 and y_2 in algorithmic form as a sequence of simple operations.

$$\begin{aligned} t_1 &= x_1 \\ t_2 &= x_2 \\ t_3 &= x_3 \\ t_4 &= t_1 \times t_1 \\ t_5 &= t_4 \times t_2 \\ t_6 &= \sin t_3 \end{aligned} \qquad \begin{aligned} t_7 &= t_5 + t_6 \\ y_1 &= t_7 \\ t_8 &= t_1 \times t_2 \\ t_9 &= t_8 \times t_3 \\ t_{10} &= e^{t_2} \\ t_{11} &= t_9 + t_{10} \\ y_2 &= t_{11} \end{aligned} \tag{2.16}$$

Now, knowing that the intermediate variable t_7 represents the output y_1 and that the intermediate variable t_1 represents the input x_1, one can compute the derivative $\partial t_7/\partial t_1$, which represents the derivative of y_1 with respect to x_1. The analytic derivative for this case is simply,

$$\frac{\partial y_1}{\partial x_1} = 2x_1x_2 \tag{2.17}$$

Now, one can write the same derivative in terms of equation (2.13). In this case, i takes on a value of 7 and j takes on a value of 1. Given these values, equation (2.13) can be expanded as follows:

$$\frac{\partial t_7}{\partial t_1} = \frac{\partial t_7}{\partial t_1}\frac{\partial t_1}{\partial t_1} + \frac{\partial t_7}{\partial t_2}\frac{\partial t_2}{\partial t_1} + \frac{\partial t_7}{\partial t_3}\frac{\partial t_3}{\partial t_1} + \frac{\partial t_7}{\partial t_4}\frac{\partial t_4}{\partial t_1} + \frac{\partial t_7}{\partial t_5}\frac{\partial t_5}{\partial t_1} + \frac{\partial t_7}{\partial t_6}\frac{\partial t_6}{\partial t_1}. \tag{2.18}$$

Now, consider each of the derivatives in this section. First of all, consider the first component in each term of the summation. These terms represent the change in the output variable, t_7, caused by a change in each one of the intermediate variables. Based on the expressions in equation set (2.16),one can write the following,

$$\begin{aligned} \frac{\partial t_7}{\partial t_1} &= 0 & \frac{\partial t_7}{\partial t_4} &= 0 \\ \frac{\partial t_7}{\partial t_2} &= 0 & \frac{\partial t_7}{\partial t_5} &= 1 \\ \frac{\partial t_7}{\partial t_3} &= 0 & \frac{\partial t_7}{\partial t_6} &= 1 \end{aligned} \tag{2.19}$$

Thus, it becomes obvious that t_7 depends explicitly only on t_5 and t_6.

Next, consider the second component in each term of the summation. These terms represent the total change in the intermediate variables t_i for a change in the input variable of interest, in this case t_1. Again, based on the expressions in equation set (2.16), the following expressions can be written as,

$$\begin{aligned} \frac{\partial t_1}{\partial t_1} &= 1 \\ \frac{\partial t_2}{\partial t_1} &= 0 \\ \frac{\partial t_3}{\partial t_1} &= 0 \\ \frac{\partial t_4}{\partial t_1} &= 2t_1 \\ \frac{\partial t_5}{\partial t_1} &= \frac{\partial t_5}{\partial t_4}\frac{\partial t_4}{\partial t_1} + \frac{\partial t_5}{\partial t_2}\frac{\partial t_2}{\partial t_1} = t_2 \times 2t_1 + t_4 \times 0 = 2t_1t_2 \\ \frac{\partial t_6}{\partial t_1} &= 0 \end{aligned} \tag{2.20}$$

Note that the chain rule from equation (2.13) is again used to represent the derivative $\partial t_5/\partial t_1$ because of its implicit dependence on t_1, through t_4. In this case, however, several zero derivatives have been neglected to save space. Now, one can combine the expressions from equation sets (2.19) and (2.20) to evaluate equation (2.18) as follows,

$$\frac{\partial t_7}{\partial t_1} = 0 \times 1 + 0 \times 0 + 0 \times 0 + 0 \times 2t_1 + 1 \times 2t_1t_2 + 1 \times 0 = 2t_1t_2. \tag{2.21}$$

Substituting in the original variables, one gets the result,

$$\frac{\partial y_1}{\partial x_1} = 2x_1x_2, \tag{2.22}$$

which is the same as the analytic result.

However, in practice, automatic differentiation tools do not apply the chain rule in this brute force way. The application is done step by step in the algorithm, which leads to two distinct approaches: the forward mode and the reverse mode. The forward mode approach, which is the easier of the two to grasp, starts with one input variable, t_j, and steps through the algorithm in the normal (forward) direction, building up derivative information as it progresses. At the end of the algorithm, this approach produces derivative information for all of the algorithm output variables with respect to that single input variable.

The reverse mode is somewhat less intuitive. With this approach, the algorithm performs a single forward pass to calculate the values of all of the intermediate variables, t. Having done this, the algorithm then performs a reverse sweep, accumulating the derivative influence from output to input. The advantage of this approach is that because it starts with a single output variable and steps backward through the algorithm, it can compute the influence of all the input variables on that output in a single pass. In cases where there are more input variables than output variables, this can lead to significant computational savings.

To help illustrate this point, each approach will be examined in more detail using the previous example. To demonstrate the forward mode, consider the input variable x_1, which corresponds to t_1. In the forward mode approach, the algorithm computes both the values and the derivative of the code at each step in the evaluation. The values are evaluated as in equation set (2.16), while the derivative are evaluated as shown in equation sets (2.23) and (2.24).

$$\begin{aligned}
\frac{\partial t_1}{\partial t_1} &= 1 \\
\frac{\partial t_2}{\partial t_1} &= 0 \\
\frac{\partial t_3}{\partial t_1} &= 0 \\
\frac{\partial t_4}{\partial t_1} &= 2t_1 \\
\frac{\partial t_5}{\partial t_1} &= t_2\frac{\partial t_4}{\partial t_1} + t_4\frac{\partial t_2}{\partial t_1} = 2t_1t_2 \\
\frac{\partial t_6}{\partial t_1} &= 0 \\
\frac{\partial t_7}{\partial t_1} &= 1 \times \frac{\partial t_5}{\partial t_1} + 1 \times \frac{\partial t_6}{\partial t_1} = 2t_1t_2 \\
\frac{\partial y_1}{\partial t_1} &= 2t_1t_2
\end{aligned} \tag{2.23}$$

$$\begin{aligned}
\frac{\partial t_8}{\partial t_1} &= t_2 \times 1 + t_1 \times \frac{\partial t_2}{\partial t_1} = t_2 \\
\frac{\partial t_9}{\partial t_1} &= t_3 \times \frac{\partial t_8}{\partial t_1} + t_8 \times \frac{\partial t_3}{\partial t_1} = t_2 t_3 \\
\frac{\partial t_{10}}{\partial t_1} &= 0 \\
\frac{\partial t_{11}}{\partial t_1} &= 1 \times \frac{\partial t_9}{\partial t_1} + 1 \times \frac{\partial t_{10}}{\partial t_1} = t_2 t_3 \\
\frac{\partial y_2}{\partial t_1} &= t_2 t_3
\end{aligned} \tag{2.24}$$

Note that at each step, the derivative is only a function of the values and derivatives computed previously. Thus, all of the derivatives with respect to t_1 can be computed in a single forward pass. However, though the derivatives for all of the output variables are generated in one pass, a separate pass has to be conducted for each input variable. In the case of design or shape optimization, where there are typically many design variables, this can incur a significant cost.

Now consider the reverse mode. In this case, one starts with a single output variable and steps backward through the algorithm, developing the derivatives in reverse order. For this process to work, the values of the intermediate variables must all be known before hand, thus a single forward pass through the algorithm is a prerequisite to this process. Consider the output value y_2, which is related to t_{11}. For the sample problem above, the analytic results for the derivative of y_2 with respect to the three input variables are,

$$\begin{aligned}
\frac{\partial y_2}{\partial x_1} &= x_2 x_3 \\
\frac{\partial y_2}{\partial x_2} &= x_2 e^{x_2} + x_1 x_3 \\
\frac{\partial y_2}{\partial t_3} &= x_1 x_2.
\end{aligned} \tag{2.25}$$

Now using the reverse mode method, the differentiation is constructed as follows,

$$\begin{aligned}
\frac{\partial t_{11}}{\partial t_{11}} &= 1 \\
\frac{\partial t_{11}}{\partial t_{10}} &= 1 \\
\frac{\partial t_{11}}{\partial t_9} &= 1 \\
\frac{\partial t_{11}}{\partial t_8} &= \frac{\partial t_{11}}{\partial t_9}\frac{\partial t_9}{\partial t_8} = 1 \times t_3
\end{aligned} \tag{2.26}$$

$$
\begin{aligned}
\frac{\partial t_{11}}{\partial t_7} &= 0\\
\frac{\partial t_{11}}{\partial t_6} &= 0\\
\frac{\partial t_{11}}{\partial t_5} &= 0\\
\frac{\partial t_{11}}{\partial t_4} &= 0\\
\frac{\partial t_{11}}{\partial t_3} &= \frac{\partial t_{11}}{\partial t_9}\frac{\partial t_9}{\partial t_3} = 1 \times t_8 = t_1 t_2\\
\frac{\partial t_{11}}{\partial t_2} &= \frac{\partial t_{11}}{\partial t_{10}}\frac{\partial t_{10}}{\partial t_2} + \frac{\partial t_{11}}{\partial t_8}\frac{\partial t_8}{\partial t_2} = 1 \times t_2 e^{t_2} + t_3 \times t_1 = t_2 e^{t_2} + t_1 t_3\\
\frac{\partial t_{11}}{\partial t_1} &= \frac{\partial t_{11}}{\partial t_8}\frac{\partial t_8}{\partial t_1} = t_3 \times t_2 = t_2 t_3,
\end{aligned}
\tag{2.27}
$$

giving the following final results,

$$
\begin{aligned}
\frac{\partial y_2}{\partial x_1} &= x_2 x_3\\
\frac{\partial y_2}{\partial x_2} &= x_2 e^{x_2} + x_1 x_3\\
\frac{\partial y_2}{\partial t_3} &= x_1 x_2.
\end{aligned}
\tag{2.28}
$$

Again, the derivatives are the same as the analytic results. However, this time, the derivatives of a single output variable with respect to all of the input variables are computed with only one forward pass and one backward pass through the algorithm. It is apparent then, that if there are more input variables than output variables, the reverse mode can allow significant computational savings.

However, while for this simple example it is fairly straight forward to break down the problem and apply simple calculus rules, in practice the application for which AD is intended are large and potentially very complex computer codes with thousands or even millions of variables. In these cases, it is nearly impossible and most certainly impractical to attempt a similar "manual" approach. As a result, computer tools have been developed to aid in the implementation of AD.

There are two main approaches used by the tools developed to implement 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. These statements would be analogous to the derivative formulae developed in equation sets (2.23) and (2.24) for forward mode and equation sets (2.26) and (2.27) for reverse mode when considering the simple example described previously. The operator overloading approach is implemented by defining a new, user defined, data type that is used instead of real numbers. This new data type includes not only the value of the original variable, but the derivative as well. In addition, all of the intrinsic operations and functions have to be redefined (overloaded) so that the derivative component is computed along with the desired function value.

There are automatic differentiation tools available for a variety of programming languages including Fortran, C/C++ and Matlab. A list of some of the available tools for Fortran has been included in Appendix D for reference, however for this work the tool Tapenade [14, 27, 13] was used. Tapenade is a non-commercial, source transformation tool that includes support for Fortran 90, a requirement for this work. The tool, which was developed at INRIA, is the successor to Odyssée [10] and is capable of performing differentiation in either forward or reverse mode.

Having outlined the basic concepts of AD and discussed the tools used to implement it, it is now possible to discuss the ADjoint in more detail.

2.3 The ADjoint

While adjoint methods have been know for quite some time, for example one of the earliest application to a fluids application is due to Pironeau [28] in 1974, the use of adjoint methods in fluid dynamics is still not commonplace. Research applications of the adjoint have progressed steadily. For example, the method was extended by Jameson to perform airfoil shape optimization [16] in the 1980's and was then used, in conjunction with a Newton-Krylov solver, for airfoil shape optimization by Nemec and Zingg [25]. Since then, the adjoint method has also been developed for more complex problems, leading to its application to the design optimization of complete aircraft configurations considering aerodynamics alone [30, 31], as well as aerodynamic and structural interactions [19]. The adjoint method has also been generalized for multidisciplinary systems [20].

Thus, given all of this activity in the research field, why has the adjoint method not gained more commonplace use in everyday applications? One of the main reasons for this is the complexity and effort involved in generating adjoint code in complex three–dimensional flow solvers. In many cases, for example in the turbulence models used to close the Reynolds-averaged Navier-Stokes (RANS) equations, the expressions used in the flow solver are prohibitively difficult to differentiate by hand. Thus, in many cases, implementing an adjoint computation is dismissed as too onerous. Often, to eliminate this problem, numerical differentiation techniques are used to alleviate the difficulty. In many cases, finite–difference methods are used to generate the partial derivatives outlined in section 2.1. However, this again leads to accuracy issues in the derivatives. More recently, Nielsen and Kleb demonstrated a method to generate the required derivatives using complex variables [26]. This approach has many similarities to the ADjoint approach and was also able to provide accurate, efficient sensitivities. Thus, given all of the previously discussed difficulties encountered when trying to compute accurate and efficient sensitivities, the goal of the ADjoint approach is to reduce the complexity of implementing an adjoint to the point where it is considered a competitive alternative for sensitivity analysis in three-dimensional CFD solvers.

The idea behind the ADjoint approach is quite simple. It is a hybrid approach between pure automatic differentiation and the adjoint method. The central idea of the ADjoint method is simply to compute the four partial derivative terms in the expanded form of the total sensitivity equation (2.7), $\partial\mathcal{R}/\partial w$, $\partial I/\partial w$, $\partial I/\partial x$ and $\partial\mathcal{R}/\partial x$, using automatic differentiation. Once these terms are computed, the adjoint method, as described by equations (2.35) and (2.36), is used to compute the desired vector of total sensitivities. By approaching the problem in this manner, the advantages of both techniques are exploited. The use of the adjoint method allows for efficient computation of the derivatives of few outputs with respect to many input values, while the use of automatic differentiation to compute the partial derivatives allows for very high accuracy and a relatively fast and simple implementation. To demonstrate these points, the ADjoint approach was used to implement an adjoint code on the three–dimensional CFD solver SUmb.

2.4 ADjoint Implementation

To demonstrate the feasibility of the ADjoint concept, the ADjoint approach was used to develop sensitivities for the SUmb flow solver [34]. SUmb, which has been developed at Stanford University under the sponsorship of the Department of Energy, is a finite-volume, cell-centered multi-block solver for the Reynolds-averaged Navier–Stokes equations (steady, unsteady, and time-spectral) and it provides options for a variety of turbulence models with one, two and four equations. For the purposes of demonstration, the sensitivities of both lift and drag coefficient, C_L and C_D, with respect to the free stream Mach number, M_∞ were developed. In the following discussion the explanations are limited to the derivatives of C_D, however the development of the sensitivities for C_L

parallel this development exactly.

2.4.1 CFD Adjoint Equations

The first step in the ADjoint approach is to develop the discrete total sensitivity equation (2.7) for the specific case of this flow solver. For the purposes of this discussion, the sensitivities are developed in terms of the three–dimensional Euler equations. The governing equations for the three–dimensional Euler flows are,

$$\frac{\partial w}{\partial t} + \frac{\partial f_i}{\partial x_i} = 0, \tag{2.29}$$

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

$$w = \begin{bmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho u_3 \\ \rho E \end{bmatrix}, \qquad f_i = \begin{bmatrix} \rho u_i \\ \rho u_i u_1 + p\delta_{i1} \\ \rho u_i u_2 + p\delta_{i2} \\ \rho u_i u_3 + p\delta_{i3} \\ \rho u_i H \end{bmatrix} \tag{2.30}$$

.

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

$$K_{ij} = \left[\frac{\partial X_i}{\partial \xi_j}\right], \qquad J = \det(K), \tag{2.31}$$

$$K_{ij}^{-1} = \left[\frac{\partial \xi_i}{\partial X_j}\right], \qquad S = JK^{-1}, \tag{2.32}$$

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,

$$\frac{\partial Jw}{\partial t} + \frac{\partial F_i}{\partial \xi_i} = 0, \tag{2.33}$$

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,

$$\frac{\mathrm{d}w_{ijk}}{\mathrm{d}t} + \mathcal{R}_{ijk}(w) = 0, \tag{2.34}$$

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

Thus, for this flow solver, the adjoint equation can be written as,

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial I}{\partial w}, \tag{2.35}$$

and the total sensitivity equation can be written as,

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \psi^T \frac{\partial \mathcal{R}}{\partial x}. \tag{2.36}$$

where ψ is the adjoint vector, $I = C_D$, and $x = M_\infty$ for the specific case show below.

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

The computation of $\partial R/\partial w$ is one of the most expensive portions of an adjoint computation. As a result, an efficient method of computing this term is needed. The original residual computation in SUmb contains a set of nested loops, whereby the fluxes for the entire grid are computed for each direction in turn. The final value of the residual in each cell being computed only at the end of all of these computations. Attempting to generate the require derivatives by performing automatic differentiation over the entire nested residual routine in SUmb would have been prohibitively expensive, both computationally and in terms of memory cost. However, by nature, the $\partial R/\partial w$ matrix is very sparse. The residual in each cell only depends on a relatively small number of nearest-neigbour and next nearest-neighbour cells. To take advantage of this fact, a set of routines that calculates the residual at a given a cell location was created. This set of routines mimics the original computation of the residuals exactly and includes all of the dissipation terms and boundary conditions, but does not loop over all of the cells in the domain. The code to compute this single cell residual was constructed from the original residual evaluation routines in the flow solver by removing the loops over all the cells in the domain and making necessary adjustments so that the appropriate boundary conditions were called for every cell in the stencil. Based on this formulation, the stencil that affects a given cell when considering the Euler and dissipation fluxes is shown in Figure 2.1. Once formulated, the single cell residual routine was then differentiated using Tapenade. in this case, because there are as many 13 input cells for every output residual, the reverse mode of AD was used to generate the code required to calculate $\partial R/\partial w$. This code now computes all of the required derivatives for a single cell in the mesh and a set of loops over the domain can be used to fully populate the $\partial R/\partial w$ matrix. Once computed, the elements of the $\partial R/\partial w$ matrix are stored in PETSc's sparse data structures for later use in the computation of the ADjoint.

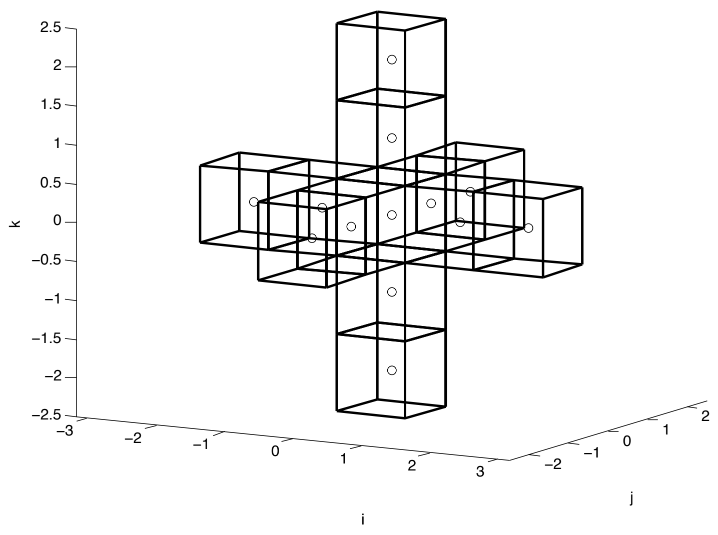

Figure 2.1: Stencil for $\partial R/\partial w$ in SUmb

Note that in this work, because the intention was only to prove the concept of the ADjoint, only the Euler and dissipation fluxes are considered. The viscous fluxes and turbulence models have been neglected. However, because the ADjoint method uses automatic differentiation, its feasibility depends only on the existence of residual code, not on the particular computations in the residual code. To extend the ADjoint to include viscous effects would simply involve including the viscous residual terms in the single cell residual routines.

2.4.3 Computation of $\partial C_D/\partial w$

The right-hand side (RHS) vector of the adjoint equations (2.35) — or matrix, in the case of multiple functions of interest — represents the direct effect of the flow variables on the function of interest. In the cases shown here, the functions of interest are C_D and C_L, so the derivatives needed are $\partial C_D/\partial w$ and $\partial C_L/\partial w$, respectively. As with the residual equations, modified versions of the original functions were used to compute the derivatives. However, in this case, because C_D and C_L are integrated quantities, the stencil for the computation must include the entire surface of the model. As a result, in general, it would be most efficient to compute these derivatives with reverse mode AD. However, in this case, given the relatively low cost of this computation, forward mode differentiation was used for simplicity.

2.4.4 Adjoint Solver

The adjoint equations (2.35) can be re-written for this specific case as,

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial C_D}{\partial w}. \tag{2.37}$$

As was mentioned previously, both the $\frac{\partial \mathcal{R}}{\partial w}$ and the right hand side in this system of equations are very sparse. To solve this system efficiently, and having in mind that a parallel adjoint solver was desired, it was decided to use PETSc (portable, extensible toolkit for scientific computation) [4, 3, 5] as the solver for this work. PETSc is a suite of data structures and routines for the scalable, parallel solution of scientific applications modeled by PDEs. Using PETSc's data structures, $\partial R/\partial w$ and $-\partial C_D/\partial w$ were stored as sparse entities. Once the sparse matrices were filled, one of PETSc's built in solvers was used to compute the adjoint solution. In this case, the PETSc implementation of GMRES was used.

2.4.5 Total Sensitivity Equation

Having computed the adjoint vector, ψ, the total sensitivity equation (2.36) can be written for this case as,

$$\frac{\mathrm{d}C_D}{\mathrm{d}M_\infty} = \frac{\partial C_D}{\partial M_\infty} + \psi^T \frac{\partial \mathcal{R}}{\partial M_\infty}, \tag{2.38}$$

where we have chosen the free stream Mach number, M_∞, to be the independent variable. As can be seen from equation (2.38), there are only two remaining terms required to form the total sensitivity equation: $\partial C_D/\partial M_\infty$ and $\partial \mathcal{R}/\partial M_\infty$.

The final term, $\partial \mathcal{R}/\partial M_\infty$, shares many components with the flux Jacobian, $\partial R/\partial w$, which has already been discussed. Thus, much of the same logic regarding the use of a single cell residual calculation still applies. However, in this case, the result is a vector, not a matrix. Thus, for this computation, the residual routines were differentiated in forward mode. The same argument applies for the first term in equation (2.38). This term has a single input, so a forward mode differentiation was used.

2.5 ADjoint Validation Results

In this section the validity of the ADjoint approach is demonstrated for a pair of single block test cases run on the SUmb flow solver. Results for both the accuracy and efficiency of the approach are shown.

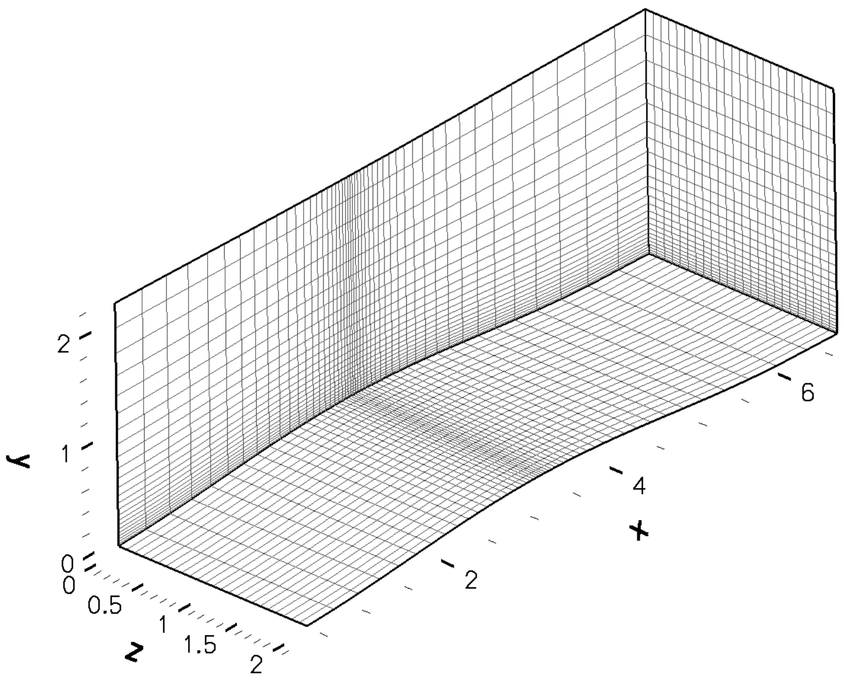

Figure 2.2: Fine bump computational domain

2.5.1 Test Cases

Two test cases have been used to demonstrate the ADjoint method: channel flow over a bump and a subsonic wing. For the bump case, the front and back walls of the channel have symmetry boundary conditions imposed on them, while the top wall is flat and the bottom wall has been deformed with a sinusoidal bump to create a reasonable variation in the flow. The inflow and outflow faces of the domain have non-reflecting boundary conditions imposed on them and the upper and lower walls use a linear pressure extrapolation boundary condition. The free stream Mach number is 2.

The wing geometry used for the second test case is the Lockheed-Air Force-NASA-NLR (LANN) wing [32], which is a supercritical transonic transport wing. A symmetry boundary condition is used at the root and a linear pressure extrapolation boundary condition is used on the wing surface. The freestream Mach number is 0.621.

The meshes for these test cases are shown in Figures 2.2 and 2.4. Both cases have relatively small meshes ($48 \times 24 \times 24$) and ($64 \times 16 \times 12$), respectively, and are being used as a proof of concept. Figures 2.3 and 2.5 show the density contours for the flow solution of each case.

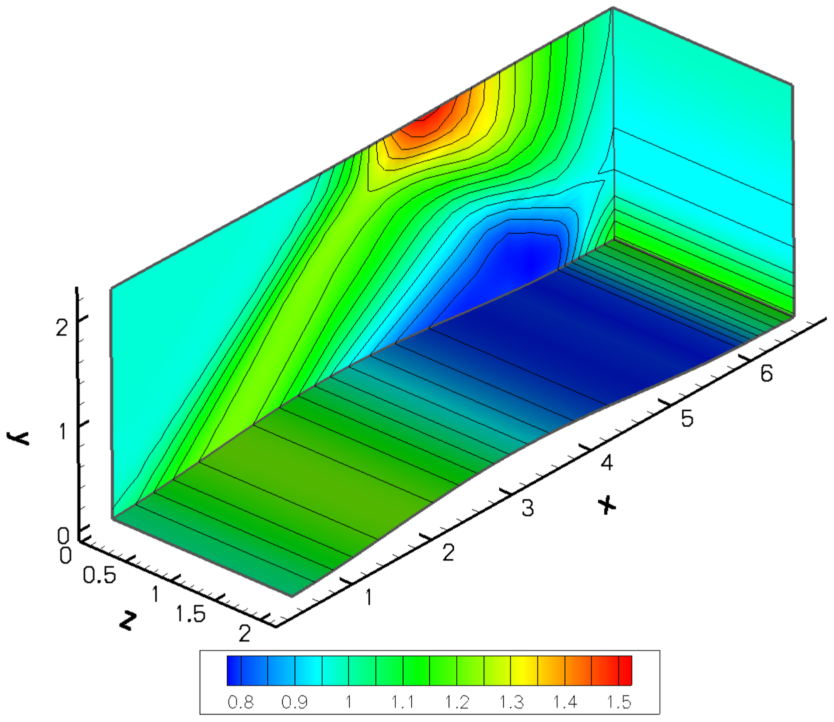

Figure 2.3: Contour plot of density

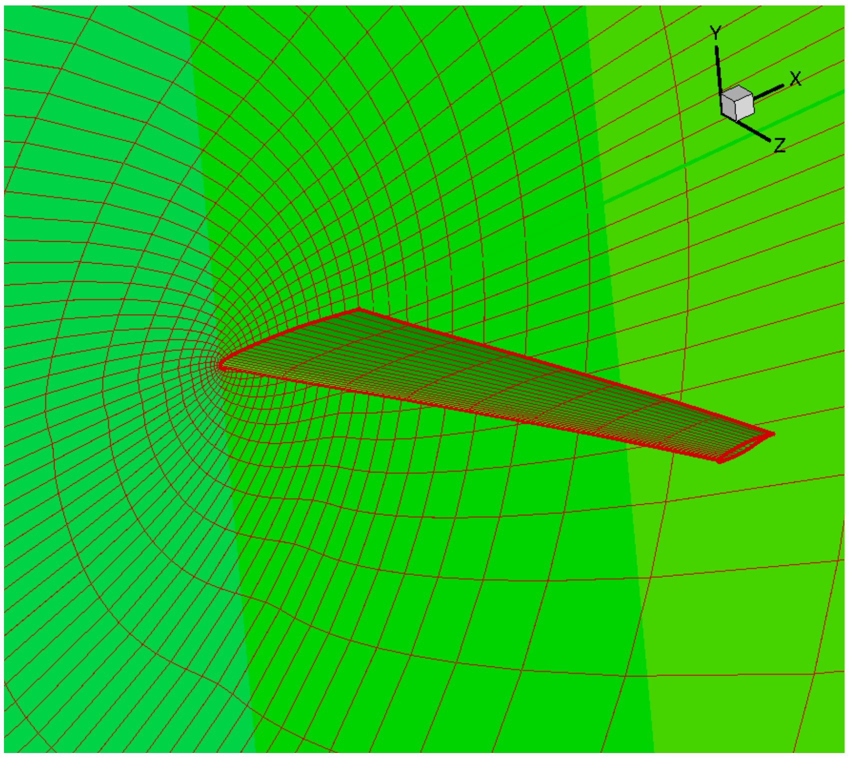

Figure 2.4: LANN wing computational domain

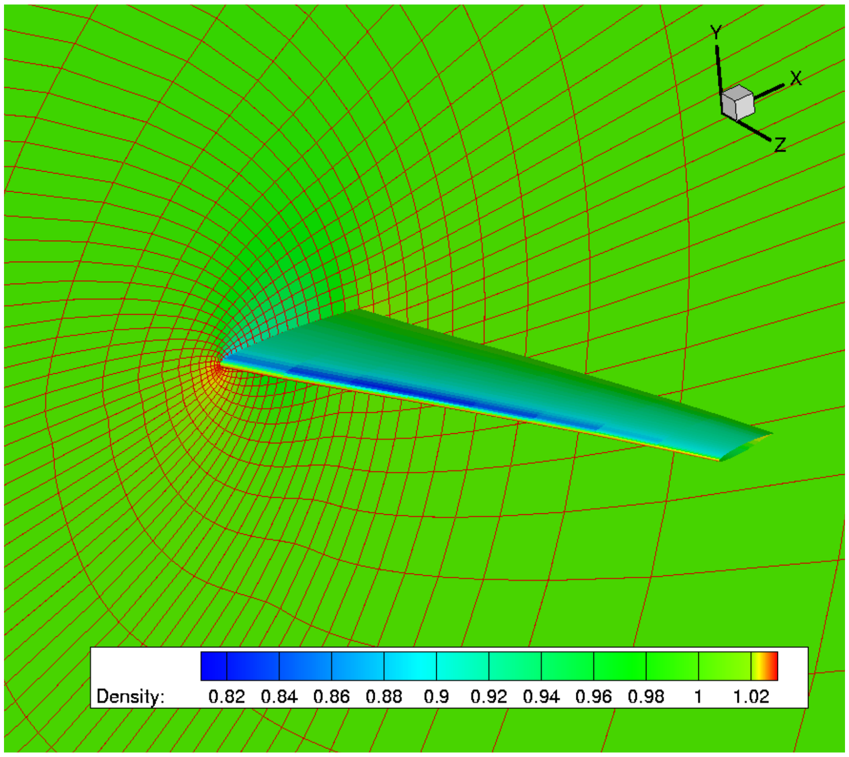

Figure 2.5: Contour plot of density

2.5.2 Lift and Drag Coefficient Sensitivity Results

The benchmark sensitivity results were obtained using the complex-step derivative approximation, which is numerically exact [22]. That is to say that the precision of the sensitivity is of the same order as the precision of the solution. The derivative in this case is given by,

$$\frac{\mathrm{d}C_D}{\mathrm{d}M_\infty} = \frac{\mathrm{Im}\left[C_D(M_\infty + ih)\right]}{h}, \tag{2.39}$$

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

In Table 2.1 results are shown for the sensitivities of both drag and lift coefficients with respect to freestream Mach number for each of the two different test cases. As the table shows, the adjoint sensitivities for these cases are extremely accurate, yielding between 11 and 13 digits agreement when compared to the complex-step results. This is consistent with the convergence tolerance that was specified both in PETSc for the adjoint solution and in SUmb for the flow solution.

To analyze the performance of the ADjoint solver, several timings were performed. They are shown in Table 5.3 for the two cases mentioned above. The fine grid has 203, 840

Table 2.1: Sensitivities of drag and lift coefficients with respect to M_∞

Mesh	Coefficient	Inflow direction	ADjoint	Complex step
Fine	C_D	(1,0,0)	-0.027950118302470*5*	-0.027950118302470*9*
	C_L		0.58128604734707 *7*	0.58128604734708*8*
Fine	C_D	(1,0.05,0)	-0.0615598631060 4*38*	-0.0615598631060 4*44*
	C_L		-0.36479675465278*87*	-0.36479675465279*97*
Wing	C_D	(1, 0.0102,0)	0.00942875710535*217*	0.00942875710535*312*
	C_L		0.267882125954*74*	0.267882125954*68*

flow variables and the wing grid has $108,800$ flow variables.

Table 2.2: ADjoint computational cost breakdown (times in seconds)

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

The total cost of the ADjoint solver, including the computation of all the partial derivatives and the solution of the adjoint system, is less than one fourth the cost of the flow solution for the fine bump case and less that one eighth the cost of the flow solution for the wing case. This is even better than what is usually observed in the case of adjoint solvers developed by conventional means, showing that the ADjoint approach is indeed very efficient. Comparing the performance of the various components in the adjoint solver indicates that the largest amount of time was spent in the solution of the adjoint equations. This is another indication that the single cell residual approach, combined with reverse mode automatic differentiation is, in fact, an efficient approach. The costliest of the automatic differentiation routines was the computation of $\partial R/\partial w$. When one takes into consideration the number of terms in this matrix, spending only 5% of the flow solution time in this computation is very impressive.

Also, while no rigorous measurement of the memory requirements of this code was performed, preliminary observations indicate that the memory required for the ADjoint code is approximately ten times that required for the original flow solver. Most of this memory increase is due to the storage of $\partial R/\partial w$ for the matrix based linear adjoint solution. Given the pattern of use on most parallel computers, this is considered well within the acceptable limits for an adjoint code.

2.6 Conclusion

In this section, the ADjoint and it underlying concepts, notably automatic differentiation and adjoint methods, have been introduced and discussed. The results from the application of the ADjoint method to the SUmb flow solver are shown, and clearly demonstrate the validity of the ADjoint approach. The comparison against the complex-step method shows exact consistency to the convergence level of the flow solver, while the timing results show excellent efficiency relative to the flow solver. The price that is paid in memory as a result of storing the entire $\partial R/\partial w$ matrix, while not insignificant, is worthwhile considering the accuracy and computational efficiency that are achieved.

Chapter 3

Aerodynamic Shape Optimization Framework

To enable aerodynamic shape optimization, several tools besides the flow solver are required to handle geometry definition, design variable definition and mesh warping. These tools – which are a subset of the tools contained in the πADO multidisciplinary design optimization framework – are coded in Fortran and, as such, need an external wrapping to interface with each other. In the case of the πADO framework, the Python programming language is used to create this wrapping. In each case, the Fortran tool is wrapped with a Python interface which allows the Fortran code to be used in Python scripts through the use of language independent shared object (.so) libraries. This approach allows direct variable passing between the different tools, which increases the speed and versatility of the framework immensely. A detailed discussion of this methodology can be found in the work by Alonso et al. [1]. In the following sections, descriptions are provided for each of the various tools in the framework. This provides a basis on which to proceed with the discussion on sensitivity analysis in Chapter 4.

3.1 pyAerosurf

pyAerosurf is the geometry engine in the πADO framework. Its role is to generate a parametric surface mesh of the aircraft from a series of text based input files (.geo, .inp) which describe the geometry of the aircraft. This description is provided in terms of number, type, shape and location of each component. The tool has the capability to model fuselages, wings, tails – both vertical and horizontal –, nacelles and pylons.

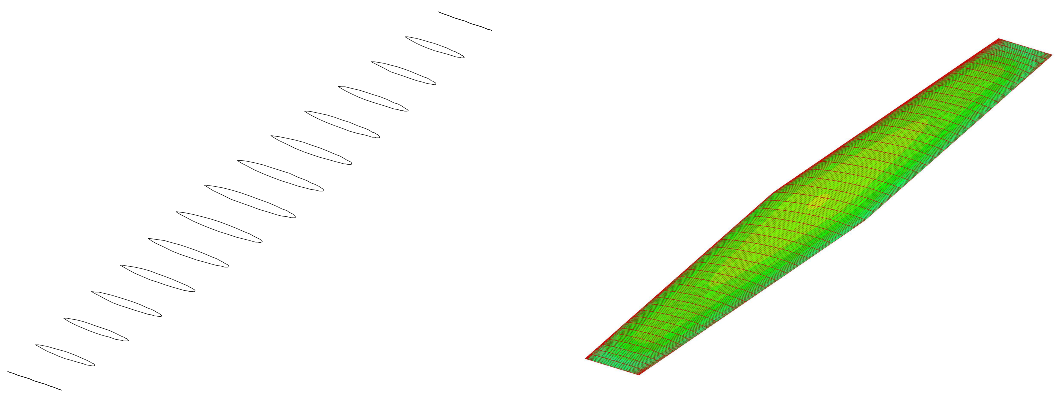

Figure 3.1: Parametric surface generation: The left hand figure shows the defining two-dimensional sections, while the right hand figure shows the lofted parametric surface.

Each of the main components of the aircraft is described using a set of cross sectional profiles, which specify the shape of the component at various stations along its length. Based on these section profiles, pyAerosurf generates a parametric description of each component of the aircraft using bi-cubic splines. These surfaces are then intersected to generate an overall parametric description of the aircraft. A sample set of sections and the corresponding lofted surface are shown in Figure 3.1. Note that because this is a single component, no intersection operations were required. Once the parametric description of the aircraft is generated, pyAerosurf can apply a variety of design variables to the aircraft. These variables include parameters such as: the leading edge coordinate of the wing sections (which will control sweep, span and dihedral), section twist angle, chord length, thickness, as well as the surface shape of the wing, which is modified using a distribution of Hicks-Henne bumps [30, 15] on the surface of the aircraft. Hicks-Henne bumps are simply a set of shape functions which can be used to specify local perturbations at specific location on the surface of the wing. Each perturbation is also given an area of influence over which the shape function is reduced to zero for integration back into the original wing surface. In this work, a sine bump shape function is used.

Once the design variables are specified, pyAerosurf generates a new surface mesh which can be used to warp the CFD volume mesh. An example of this design variable modification is shown in Figure 3.2. It is important that the dimensions of the intersected pyAerosurf patches do not change between design iterations. Otherwise, the parameterization generated to interface the geometry mesh with the CFD volume mesh by pyCFD-CSM – discussed in section 3.2 – will become invalid and the volume mesh warping procedure will not function correctly. To ensure that this is the case, pyAerosurf

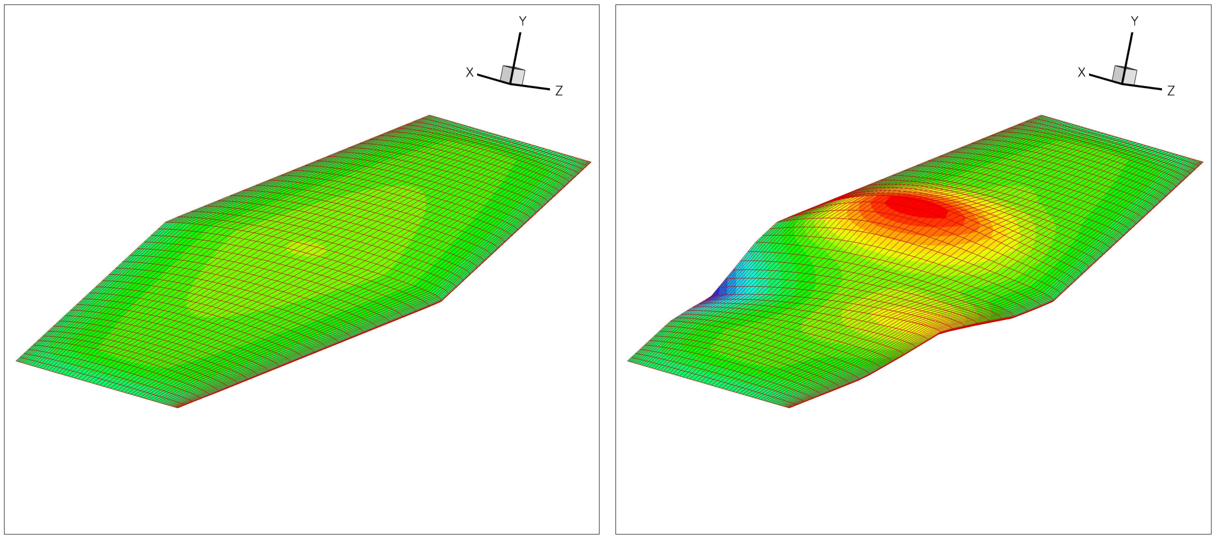

Figure 3.2: Design variable application: The left hand figure shows the unperturbed surface geometry, while the right hand figure shows the geometry perturbed by a twist variable and a surface bump. Both perturbations are exaggerated to improve the visualization.

allows the user to provide a baseline model – also generated by pyAerosurf – which can be used as a template for the patch configuration of all subsequent intersection operations.

Though the primary purpose of the pyAersurf module is to manage the paramaterized geometry of the aircraft, as discussed above, the Fortran code also has the capability of handling the geometry based design variables of the aircraft. However, the standard method for adjusting design variables in Aerosurf is to pass in a text file with all of the desired design changes specified. While it would have been possible to use this approach, it is much more efficient to pass the design variable information directly from the optimizer to the geometry tool. Thus, in order to allow modification of the design variables by the optimizer at run-time, a Python interface was generated for this functionality as well. This interface generates a text string that emulates the text file that would normally be passed into the program. By doing this, the optimizer can specify the design variables for each design iteration and regenerate the design variable input during run-time. To aid in this process, the Python “designVariable” class shown in Figure 3.3 was created to store all of the data required for each design variable. At the beginning of each optimization run, the user can specify what types of design variables they wish to use, e.g., surface shape, twist, leading edge location, and a dictionary containing all the required instances of the design variable class is generated and populated. Based on the number of sections

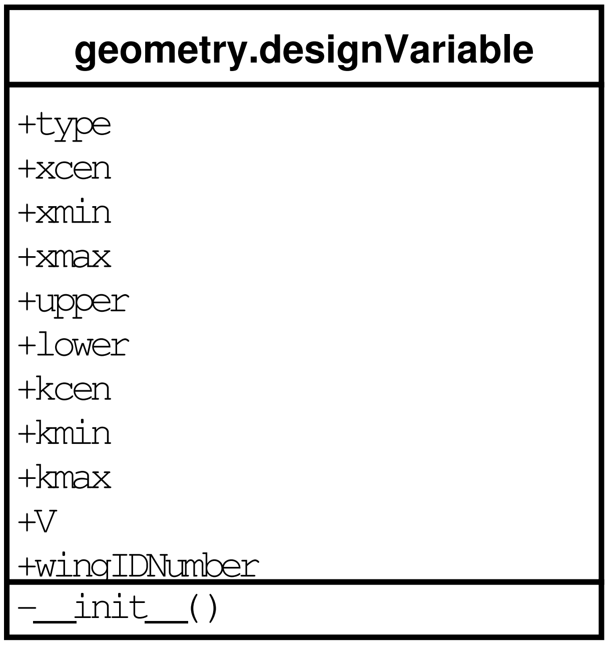

Figure 3.3: UML diagram of the `designVariable` class in pyAerosurf

defining the shape and the number of design variables requested, the values in the class are automatically populated to distribute the design variables over the surface of the model. Using this approach, the only variable the optimizer has to modify during the run is the value or “V” attribute of each instance of the design variable class, where there is one instance of the class for each design variable.

3.2 pyCFD-CSM

pyCFD-CSM is the module that generates the relationship between the surface mesh generated by pyAerosurf and the CFD surface mesh present in pyWarp and pyNSSUS. This association is developed using an alternating digital tree (ADT) search, which determines the associativity of the points on the two surface meshes. From this associativity, a set of weighting factors is generated relating the two sets of point. Using these weighting factors, any shape changes in the geometry model, caused by a change in design variable, can be interpolated to the CFD Surface. More details on this methodology can be found in Martins [18] and Alonso et al. [1].

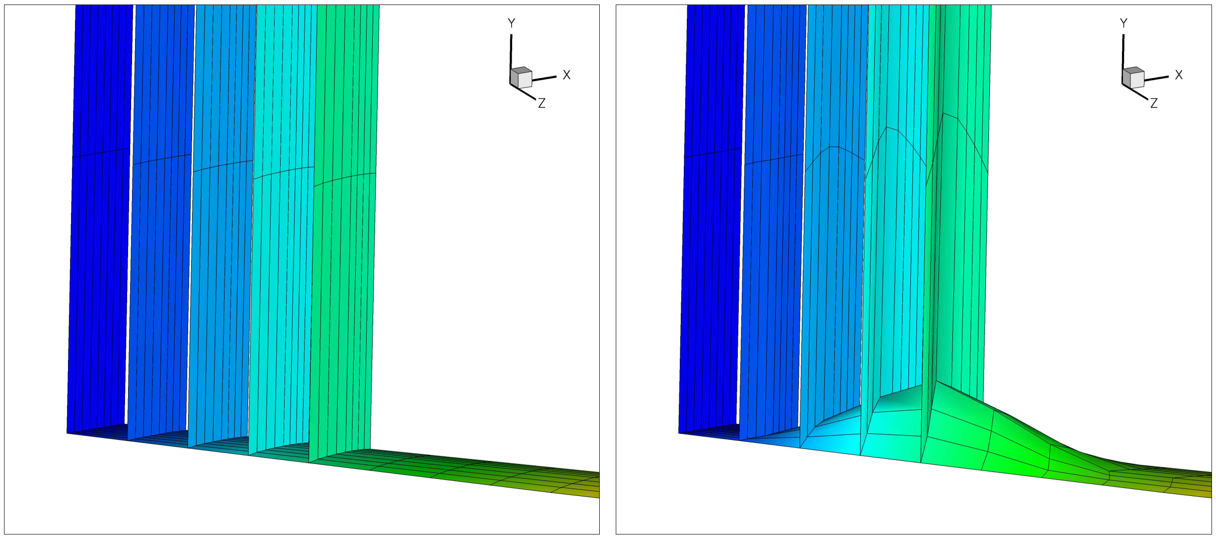

Figure 3.4: Warped mesh visualization: The left hand figure shows the unwarped mesh, while the right hand figure shows the effect that a large surface bump has on the nodes of the volume mesh. The vertical strips are artificial surfaces showing the first five layers of nodes in the volume mesh to allow for the visualization of the perturbations in the volume mesh.

3.3 pyWarp

pyWarp is the mesh warping algorithm used in πADO. It is an algebraic, multiblock, mesh warping algorithm for structured meshes. The algorithm moves, in order, the explicitly perturbed corners, edges and faces, the implicitly perturbed corners and edges, the implicitly perturbed faces and then, from that information, perturbs all of the interior points. Each of the grid lines of the mesh is scaled linearly, based on a set of weights generated from the unperturbed mesh. The multiblock capability is enabled through a set of master corners and edges. An example of that mesh warping is depicted in Figure 3.4. More details on this method can be found in Martins [18] and Alonso et al. [1].

3.4 pyNSSUS

The NSSUS solver is a three–dimensional, higher–order, finite–difference solver that has been developed at Stanford University under the Advanced Simulation and Computing (ASC) program sponsored by the United States Department of Energy [2]. It is a node-based, multi-block, multi-processor solver, tested for the Euler equations and ideal magnetohydrodynamic equations [17], and currently being extended to the Reynolds–Averaged

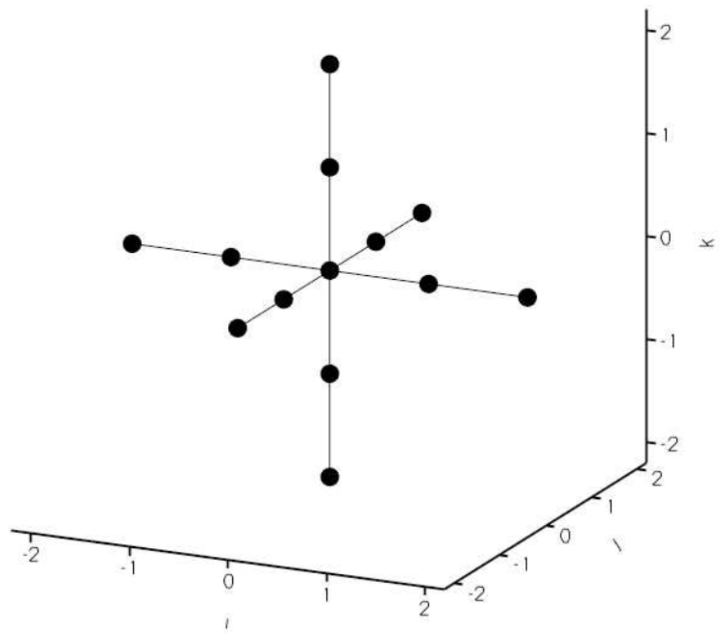

Figure 3.5: Stencil for the vertex-centred residual computation.

Navier–Stokes equations for a variety of turbulence models. The finite–difference operators and artificial dissipation terms follow the work by Mattsson [23, 24] and the boundary conditions are implemented by means of penalty terms, according to the work by Carpenter [7, 8]. The solver is capable of solving the flow using equations up to eighth order accuracy, however the work shown here only deals with portions of the code used to solve equations using first and second order accurate discretizations.

3.4.1 Discretization Overview

The flow equations are discretized over the CFD mesh primarily on an internal, block-by-block basis. The internal discretization for each block is carried out using central finite differences. This means that, for second order accuracy, each node only requires the first level of next nearest neigbour nodes for the inviscid fluxes. The dissipation fluxes require first and second nearest neighbours. Thus, each node is affected by the stencil of nodes shown in Figure 3.5. An exception to this occurs at the boundaries of each block. In order to keep the discretization internal to the block, one sided difference formulae are used at all block boundaries, both internal and external. However, because of this one sided treatment at the boundaries, the boundary node information is required for

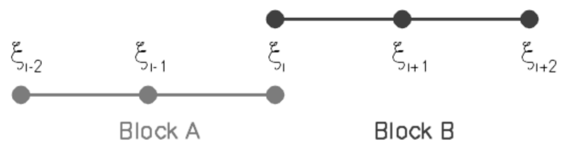

Figure 3.6: Block-to-block boundary stencil.

every block. As a result, the internal boundary nodes occur multiple times, once in each block that touches a given internal boundary. An example of the node locations for a two-block, face to face boundary is shown in Figure 3.6 . However, left unaltered, this scheme would not provide any connectivity between the blocks in the mesh. In order to enable the solution of the system on the whole mesh, the multiple instances of the internal boundary nodes need to be driven towards the same value. This is accomplished through the use of penalty terms. More specifically, an additional penalizing term is added to the residual $\mathcal{R}$ of each boundary node. This penalty term is proportional to the difference between the respective values of the boundary node instances and take the following form:

$$\mathcal{R}^i_{\text{blockA}} = \mathcal{R}^i_{\text{blockA}} + \tau(w^i_{\text{blockB}} - w^i_{\text{blockA}}), \tag{3.1}$$

where the value τ controls the strength of the penalty and is a combination of a user defined parameter and the local flow conditions. A similar expression can be written for $\mathcal{R}^i_{\text{blockB}}$. More details on this parameter can be found in Mattsson [23]. Based on this discretizaton, the residual values of the internal boundary nodes are a function of the neighbouring nodes in their local block as well as the corresponding instance of the node in any neighbouring blocks. The physical boundary condition (BC) treatment is very similar to the approach described above. The difference is that the penalty state used in equation (3.1) is determined by the boundary condition as opposed to another instance of the node in a neighbouring block. More details on this discretization and the associated penalty parameter can be found in Mattsson [23, 24] and Carpenter [7, 8].

3.4.2 Implementation Overview

As with all of the other components of the framework, a Python wrapper was created for NSSUS. The original NSSUS code is written in Fortran 90 and is capable of running in parallel, using the MPI protocol. I/O functionality is handled with files. The mesh and boundary conditions are specified with either PLOT3D or CGNS formatted files.

The flow conditions, boundary condition options, iterations parameters and discretization options are all set through a text based input file. The same holds true for the Python implementation. The name of the input file is simply provided as an input to the initialization routine in the NSSUS Python class, and then the Python script calls the same Fortran based I/O routines that the original code would have called. The flow solution is handled in a similar manner. A Python level driving script calls the original Fortran routines in the desired order. This has the added benefit that it allows the user to reconfigure the code from NSSUS somewhat in order to better accommodate the optimization process. The output of the final solution is also handled by Python functions that call the original Fortran routines.

3.5 pySNOPT

pySNOPT is a numerical optimization package for the solution of large scale non-linear optimization problems. The underlying Fortran code, SNOPT, is a quasi-Newton, SQP optimizer. The optimizer can handle large numbers of design variables and is capable of handling fixed design variable bounds as well as linear and non-linear constraints. Once again the Fortran code is wrapped in Python, with the Python wrapper allowing for direct interaction between the optimizer and the other portions of the code.

3.6 Framework Summary

Thus, having wrapped each of the individual components of the framework in Python, it is now possible to combine them together into a single functioning aerodynamic analysis and optimization package. Diagrams of the information flow between the modules are shown for both the aerodynamic analysis and the aerodynamic optimization in figures 3.7 and 3.8 respectively. As shown in Figure 3.7, all of the information feeds forward in the aerodynamic analysis. pyAerosurf generates the geometry and passes it to pyCFD-CSM. pyCFD-CSM creates the association with the CFD surface mesh and passes a new surface to pyWarp. pyWarp generates a new volume mesh and passes it to pyNSSUS, which in turn calculates a flow solution. However, as shown in Figure 3.8 there is both feed forward and feed back of information in the optimization. The same forward flow of information that occurs in the analysis occurs again, but at the end of each analysis pass, flow solution values and derivative information are passed back to the optimizer so

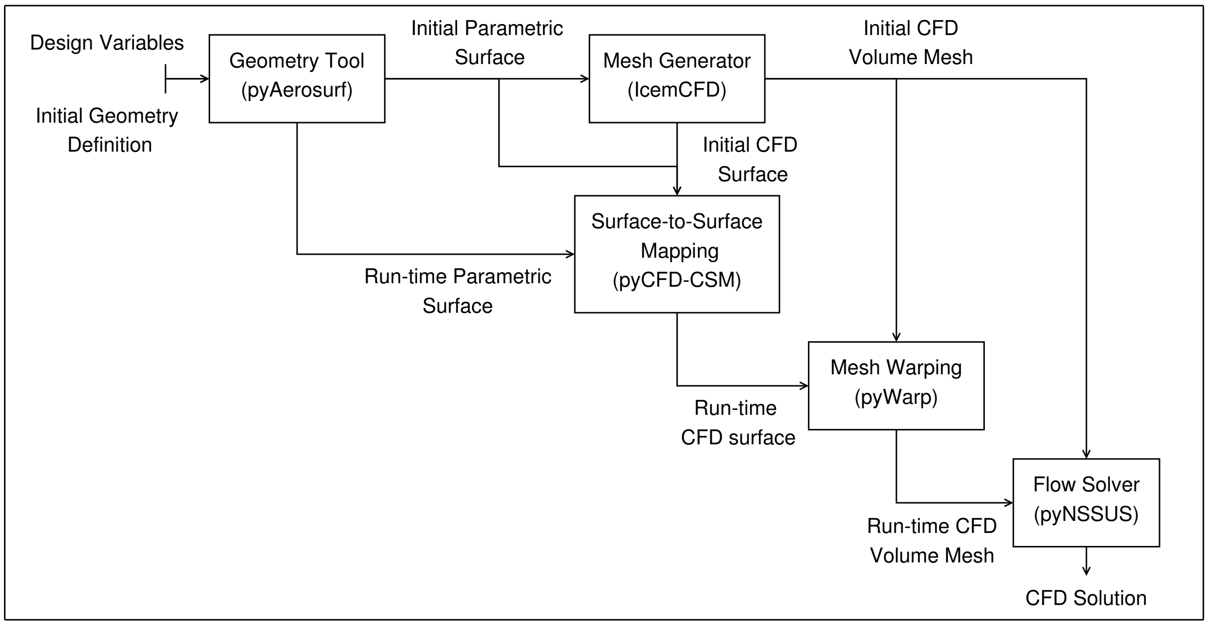

Figure 3.7: πADO analysis information flow

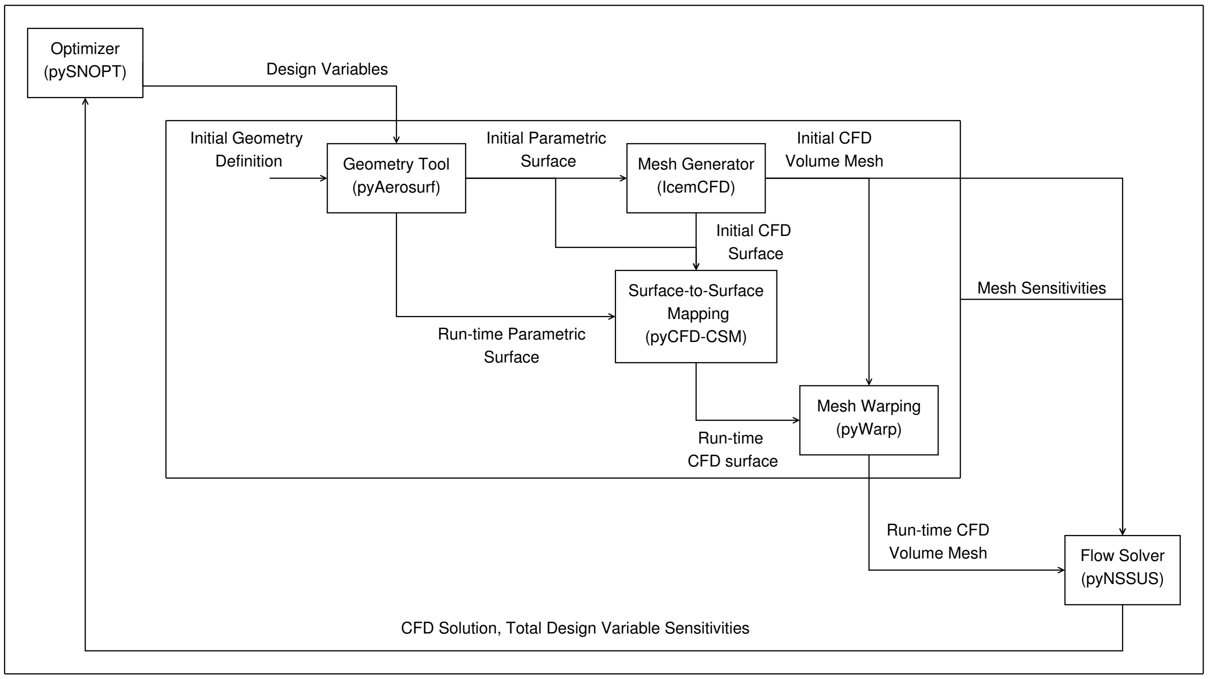

Figure 3.8: πADO aerodynamic optimization information flow

that it can select a new design point. This new design point is returned to the analysis and the process is repeated. In order for this optimization process to be efficient, the computation of the derivatives must be efficient. Otherwise, the optimization process will bog down and be very slow in producing a result. The details of this computation – in this case, the aerodynamic derivative implementation for πADO – are discussed in Chapter 4.

Chapter 4

Sensitivity Analysis

The previous two chapters have laid the groundwork for a discussion of the computation of the design variable sensitivities. In Chapter 2, the concept of the ADjoint was introduced as an efficient way of combining adjoint methods and automatic differentiation to get accurate sensitivities. In Chapter 3, the layout of the aerodynamic portion of the πADO framework was described. In this chapter, the last major component of the optimization framework, the computation of the design variable sensitivities, will be discussed. As was discussed in Chapter 3, the aerodynamic framework is made up of a number of different components. Unfortunately, there is no "one size fits all" technique that works well for all of the components. As will be discussed in this chapter, a variety of techniques have been used to take advantage of the structure of each of the different components of the framework.

From the discussion in Chapter 3 it is apparent that there are two different types of code in the framework. The flow solver is an iterative code based on residual equations, while the remainder of the components in the framework are not. This difference in the codes has implications in terms of the derivatives. First of all, the flow solver is far more expensive computationally than all of the other components combined. Secondly, the semi-analytic methods used to generate the ADjoint are applicable to the flow solver, but not to the other components of the framework. As a result, the design variable sensitivities for this work were computed in two parts using two different methods. The derivatives of the geometry manipulation and mesh warping tools – i.e. pyAerosurf, pyCFD-CSM and pyWarp – were computed using the complex-step method, while the derivatives inside pyNSSUS were computed using the ADjoint method outlined in Chapter 2. The complex-step method used for the geometry and mesh components generates the derivatives of the volume mesh coordinates with respect to the design variables, while the ADjoint method used in pyNSSUS generates the derivatives of the force coefficients $(C_L, C_D, C_{Mx}, C_{My}, C_{Mz})$ with respect to those same volume mesh coordinates. Once each of these two terms is computed, they are multiplied together to form the total sensitivities required for the optimization. More details on each of theses methods is provided in the following sections.

4.1 Geometry and Mesh Component Sensitivities

The sensitivities of the grid coordinates with respect to the geometry design variables were calculated using the complex-step method [22, 33]. Several methods were considered for computing these derivatives, including finite–difference methods, the complex-step method, automatic differentiation and semi-analytic methods. Some initial work was conducted using finite–differencing, but as will become clear in the results section, finite–difference methods simply could not achieve the accuracy desired. This left a choice between complex-step methods, automatic differentiation (AD), and semi-analytic methods. In this particular part of the problem, semi-analytic methods – including adjoint methods – are not applicable, because these segments of code do not involve calculations based on residual equations. Without residual equations, the formulation of the partial derivative in the adjoint equations is not possible. This leaves the choice between the complex-step method and automatic differentiation, where we know from Martins et al. [21] that the complex-step is essentially equivalent to forward mode AD using operator overloading.

4.1.1 Overview

With the preceding discussion in mind, consider the problem at hand. For the geometry and mesh definition portion of the framework, the sensitivities of interest are the sensitivities of the CFD volume mesh coordinates with respect to a set of geometric design variables describing the planform and cross–sectional shape of the wing. In the context of a typical problem, there will be far more mesh coordinates than design variables. To describe a wing, for example, one may have 15 or 20 cross sections each with 20 to 25 associate design variable for location, shape, twist, etc. This means that the number of design variables is on the order of hundreds to thousands $(\mathcal{O}(10^2) - \mathcal{O}(10^3))$ of variables.

However, in a typical high-fidelity CFD mesh, one may have hundreds of thousands or even millions of nodes($\mathcal{O}(10^5) - \mathcal{O}(10^6)$), each with three degrees of freedom. From this simple comparison, it is quite clear that it is far more beneficial to use a forward sensitivity method, like the complex-step method or forward mode AD, than it is to use a reverse sensitivity method such as reverse mode AD. The decision to use complex-step as opposed to forward mode AD was made based on the relative ease of implementation for these particular code segments.

4.1.2 Implementation

Having decided to use the complex-step method to implement the derivatives in the geometry and mesh handling components of the framework, the implementation of the derivatives was relatively straightforward. The complexification of the fortran code was carried out using the methods and tools proposed by Martins et al. [22]. The application of this methodology went smoothly with two exceptions. The original methodology proposed by Martins does not consider the extension of the method to codes wrapped with python. However, this problem was overcome with two modifications to the complexification procedure. First, one needs to complexify the signature file (*.pyf) associated with the wrapping, this includes the complex variable declarations in the wrapping, allowing it to match up with the modified source code. Then, one needs to set all of the python variables – particularly any declared arrays – to be of complex data types. With these modifications made to the complexification procedure, each of the codes was able to receive complex number perturbations from its python level driver code.

The second major issue encountered while complexifying these codes is one that is a more fundamental issue with the complex-step method. At the conceptual level, both the complex-step method and automatic differentiation rely on the augmented code following the same logical path through the code as the original version for a given operation. In two sections of the geometry code, there are branching statements that check for real valued perturbations from the initial state. These commands are setup to bypass large sections of code if no change in the mesh or design variables are present and to prevent division by zero. While these checks work well in the original code, they causes the complex-step method to fail. In a case where a complex perturbation exists, the perturbation code needs to be run as if there was a real perturbation so that the complex component is modified correctly. However, the base branching statements for the complex-step method

```
def getCoordinateSensitivity(self,x,xindex,xyzref):
    #initialize derivative array for a single design variable
    dx = zeros([len(xyzref)*3],'d')
    #set stepsize
    deltax = 1e-20j
    #store reference design variables
    xref = x[xindex]# x.copy()
    #perturb design variables
    x[xindex] = x[xindex]+deltax
    #calculate new volume mesh coordinates
    xyznew = self.recalculateMesh(x)
    #restore original design variables
    x[xindex] = xref
    #compute derivative
    tmp = (xyznew.imag)/deltax.imag #complex step
    dx = tmp
    return dx
```

Figure 4.1: Geometry and mesh component complex-step function

check only the real part of the solution. Thus, in two sections of the code, branching statements had to be modified to identify the complex perturbations. The exact changes required have been included in Appendix E.

With the exception of these minor modifications, the complexification of the geometry and mesh manipulation code went smoothly. With the complexified versions of the code in place, a simple python level driver script is all that is needed to generate the required derivatives. The script, shown in Figure 4.1, loops over the design variables and adds a complex perturbation to each in turn. The resulting complex values computed in the node coordinates produced from pyWarp represent the sensitivities of each of the node coordinates of the CFD volume mesh with respect to the design variables. This completes the first half of the required derivatives. The computation of the second half of these derivatives, which are computed in pyNSSUS, is discussed below.

4.2 pyNSSUS Sensitivities

The second half of the sensitivity computation required for aerodynamic shape optimization is the computation of the sensitivities of the aerodynamic force coefficients with respect to the CFD volume mesh coordinates. Because the flow solver is based on a set of residual equations, semi-analytic methods are applicable. Further, there are relatively few force coefficient output values, in this case only C_L ,C_D, C_{M_x},C_{M_y} and C_{M_z} while there are many input values, $\mathcal{O}(10^5)-\mathcal{O}(10^6)$ CFD volume mesh nodes. Thus, an adjoint type method is best suited to this application. As a result, for the reasons discussed in Chapter 2, the ADjoint approach, also introduced in Chapter 2, will be used to compute the sensitivities in pyNSSUS. The basic structure of the ADjoint formulation is the same for pyNSSUS as it was for SUmb, as described in section 2.4. The following sections describe the specific implementation details of the sensitivities in pyNSSUS.

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

To frame this discussion, consider the following size definitions:

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

N_s: The number of nodes in the stencil whose state variables affect the residual of a given node. When considering inviscid and dissipation fluxes, the stencil is as shown in Figure 3.5 and $N_s = 13$.

N_w: The number of flow variables for each node. For the Euler equations $N_w = 5$.

The values in the Jacobian matrix, $\partial R/\partial w$, are independent of both the choice of objective function and the selected design variable. The values of the matrix depend only on the governing equations, their discretization and boundary conditions, and the states of the flow solver at the converged solution. As a result, this matrix can be formed once and reused for all of the necessary adjoint solutions. However, the Jacobian matrix is both very large and very sparse, $N_nN_w \times N_nN_w$ with no more than $N_w(N_s+1)$ entries in any row or column. Thus, to store it efficiently, a sparse matrix data structure is necessary. In this work, the Portable Extensible Toolkit for Scientific computation (PETSc) [4, 3, 5] has been used for both the storage of the partial derivatives and the computation of the adjoint solution. PETSc is a suite of data structures and routines for

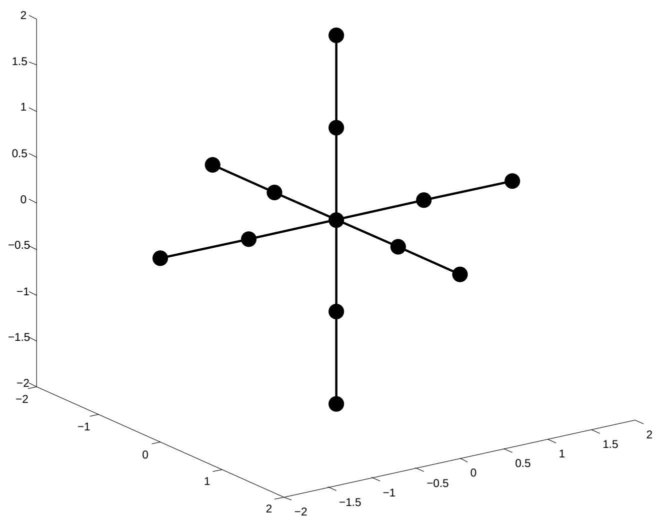

Figure 4.2: Stencil for the $\partial R/\partial w$ computation, based on Euler equations with 2^{nd} order finite difference discretization

the scalable, parallel solution of scientific applications modeled by PDEs. It employs the message passing interface (MPI) standard for all inter-processor communication. This broad, general applicability makes it ideally suited for use in conjunction with the ADjoint approach.

Consider the residual computation in pyNSSUS. For this case, the Euler equations are being used with a second order, finite difference discretization. Thus, the procedure for computing the residual in pyNSSUS can be described as follows:

1. Compute inviscid fluxes: For the 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 this portion of the residual, the states,w, at the current node and the 12 adjacent nodes, ± 2 in each of the three directions need to be considered, as shown in Figure 4.2.

3. Apply boundary conditions: Additional penalty terms are added to enforce the boundary conditions (see section 3.4.1 for details). In this context, the internal block boundaries are also considered as boundary conditions and are enforced using penalty terms.

In the actual flow solver routines, the computation of the residuals is performed using three nested loops – one for each of the three coordinate directions. The loops are setup such that the flux is calculated in each direction and then summed to get the residual. While this method is efficient, it also means that the correct value of the residual for any given node is only obtained at the end of all three loops, when all contributions have been accounted for. Using automatic differentiation (AD) on this original routine would require many unnecessary calculations, since the residual at each node is only affected by a limited number of nodes immediately surrounding the target node. Thus, to make the implementation of the discrete adjoint solver more efficient, it was necessary to re-write the flow residual routine in such a way that it was capable of computing the residual for a single specified node. This new set of routines takes in the states of the local stencil as an input and returns the residual values at the node specified. In order to get the information for the whole mesh, a series of loops is needed to specify each node of the mesh, one at a time, as an input to the routine.

Given this new residual routine, one can now determine what mode of automatic differentiation will provide the best performance for computing the terms in $\partial R/\partial w$. The new residual routine 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. Given that there are far more input variables than output variables, reverse mode AD is going to be far more efficient. In this case, the reverse mode would require N_w calls to the differentiated routine, while the forward mode would require $(N_w \times N_s)$ calls. There is a computational cost penalty associated with each call to a reverse mode differentiated routine, due to its inherently more complicated implementation. However, for sufficiently large stencils and sufficiently costly computations, this penalty is more than offset by the reduced number of calls required.

Thus, for the $\partial R/\partial w$ derivatives in pyNSSUS, the reverse mode of AD is used to differentiate the single cell residual calculation. Once differentiated, the code provides the $N_w \times (N_w \times N_s)$ sensitivities for each node with N_w calls to the differentiated routine. A set of three nested loops is then used to loop over all of the nodes in the mesh, thus generating the values for all of the rows in the $\partial R/\partial w$ matrix. As they are generated, these values are stored in PETSc sparse matrix format, where they can be used later in the solution of the adjoint equation (4.1).

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

The right-hand side (RHS) of the adjoint equations (2.35) 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 the expression for these partial derivatives analytically from the flow solver routine that calculated the functions of interest. However, in the general case of a flow solver where this approach is not applicable, one would use an approach similar to the one described in section 4.2.5 in which a routine is created that takes the whole grid of state variables as inputs values and computes the force coefficients. This routine can then be differentiated, most likely in reverse mode, to compute the desired derivatives.

4.2.3 Adjoint Solver

Consider a single example output value, for example C_D. The adjoint equations (2.35) can then be re-written as,

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial C_D}{\partial w}. \tag{4.1}$$

With the two partial derivatives in this equation computed as outlined above, the adjoint vector, ψ can be computed. As discussed previously, both the Jacobian and the right hand side in this system of equations are very sparse, so PETSc has been used to store and compute the adjoint solution. Once the sparse matrices were setup, PETSc's parallel Krylov solver was used to compute the adjoint solution. While the PETSc library has a number of different solvers, in this work, only the Generalized Minimum Residual (GMRES) solver was tested.

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

While the computation of the adjoint solution is independent of the design variables, eventually their impact needs to be taken into account. In the total sensitivity equation, this is done by the first and last terms, $\partial \mathcal{R}/\partial X(i, j, k)$ and $\partial C_i/\partial X(i, j, k)$. 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,

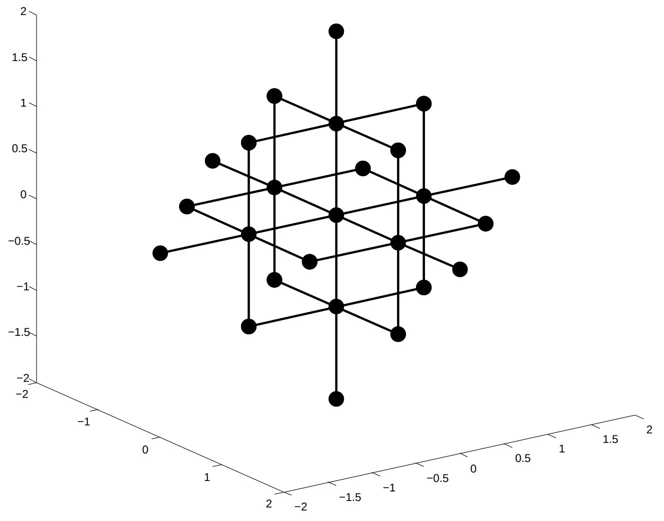

Figure 4.3: Stencil for the $\partial R/\partial X$ computation

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. However, because of the addition of the metric terms, an additional set of first neighbour nodes is required. This extended stencil is shown in Figure 4.3. However, because this stencil still fits inside the same $N_w \times N_w$ node box as the $\partial\mathcal{R}/\partial w$ stencil, the same set of single cell residual routines can be used. 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.

4.2.5 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, $\partial C_i/\partial X(i,j,k)$. 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 surface of the model, the small stencil used to compute the Jacobian is no longer valid. In this case, the stencil must be extended to include the entire surface of the mesh. While this does raise some issues, it also leads to some significant advantages. For example, a single call to a reverse mode differentiated routine 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 also leads to very fast derivatives. Note that for simplicity in this case, the stencil was actually extended to include the entire volume mesh. While this will again increase memory usage and reduce the efficiency of the derivative computation, at this stage, the impact is not considered to be significant. Once again, due to the sparsity of the matrix, the derivative values are stored in PETSc.

4.2.6 Total Sensitivity Equation

For the specific case of the mesh coordinate derivatives, the total sensitivity equation (2.36) can be written as,

$$\frac{\mathrm{d}C_i}{\mathrm{d}X(i,j,k)} = \frac{\partial C_i}{\partial X(i,j,k)} + \psi^T \frac{\partial \mathcal{R}}{\partial X(i,j,k)}, \tag{4.2}$$

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 4.2. This will provide a single vector of total sensitivities with length $3N_n$ for each force or moment coefficient of interest. This vector is passed back to the python driver routine where it can be combined with the geometry sensitivities to compute the total design variable sensitivities required for optimization.

4.3 Total Design Variable Sensitivities

The previous two sections discuss the implementation of the two halves of the design variable sensitivities. Once these two halves are computed, they are brought back to the python driver program as matrices and multiplied together to compute the final design variable sensitivities. The main reason for computing the derivatives in this fashion is the one laid out at the beginning of this chapter: the fact that one portion of the code was amenable to semi-analytic methods, while the other portion was not. However, even given this distiction, there are many ways in which the various components could have been grouped for computation.

Consider equation (4.3), the total sensitivity equation for the entire aerodynamic framework:

$$\frac{\mathrm{d}C_i}{\mathrm{d}X_{DV}} = \frac{\mathrm{d}C_i}{\mathrm{d}X_{CFD_{vol}}} \frac{\mathrm{d}X_{CFD_{vol}}}{\mathrm{d}X_{CFD_{surf}}} \frac{\mathrm{d}X_{CFD_{surf}}}{\mathrm{d}X_{GEO_{surf}}} \frac{\mathrm{d}X_{GEO_{surf}}}{\mathrm{d}X_{DV}}. \tag{4.3}$$

Each one of the derivatives can be directly related to a specific portion of the framework: $\mathrm{d}C_i/\,\mathrm{d}X_{CFD_{vol}}$ is related to the flow solver (pyNSSUS), $\mathrm{d}X_{CFD_{vol}}/\,\mathrm{d}X_{CFD_{surf}}$ comes from the warping algorithm (pyWarp), $\mathrm{d}X_{CFD_{surf}}/\,\mathrm{d}X_{GEO_{surf}}$ comes from the surface to surface relations of pyCFD-CSM and $\mathrm{d}X_{GEO_{surf}}/\,\mathrm{d}X_{DV}$ comes from the geometry code (pyAerosurf).

The most straight froward way to handle these derivatives would be to compute each of the terms in equation (4.3) separately, each in their own separate codes, and then combine them to get the total derivative. This would allow the ADjoint method to be used in the flow solver, while allowing other, better suited methods to be used for the remaining components of the framework. However, this would result in a number of inefficiencies. Each of the components listed in equation (4.3) is, in and of itself, a large matrix. In most of the cases, these intermediate matrices are larger than the final matrix of derivatives of interest. Thus, a large amount of memory would be wasted in storing each of these individually. Further, using a method such as finite-differences or complex-step over each one of these routines would require numerous calls to each of the code segments, increasing the computational time significantly. Thus, a method is needed to combine the computation of some of these terms to reduce the amount of memory and computational time required to compute the desired derivatives.

Leaving aside the flow solver for the time being, there are three code segments to consider: pyAerosurf, pyCFD-CSM, and pyWarp. Each of these codes has been complexified to enable the use of the complex-step method. Further, with the modifications described in section 4.1.2 this complexification has been extended to the python level wrapper of each of these codes. Thus, a single complex-step derivative computation can be conducted to compute the derivatives of all three codes at the same time. By adding a complex perturbation to a design variable and carrying the resulting complex perturbations through the python interface to each successive fortran module, one can compute the derivatives for all three code segments in a single pass.

This approach is very effective when applied to the first three modules, but breaks down when the flow solver is considered. Because of its iterative nature, the complex-step method is far to expensive to use over the entire flow solver. Using it for the partial derivatives in an adjoint method would be possible, but as discussed in section 4.2, reverse mode AD is more efficient for the computation of the partial derivatives in the flow solver. Further, making the flow solver complex affects the speed of the flow solver itself, which is not desirable. This creates a logical split in code segments between the flow solver and the rest of the geometry and mesh codes. The derivatives of the geometry and mesh components of the framework, those related to the last three terms in equation (4.3), are computed using a complex-step method over the entire calculation, while the derivatives of the flow solver are computed using the ADjoint method as discussed in section (4.2). When the ADjoint method is applied to the flow solver sensitivities, equation (4.3) takes the following form,

$$\frac{\mathrm{d}C_i}{\mathrm{d}X_{DV}} = \left[\frac{\partial C_i}{\partial X_{CFD_{vol}}} + \psi^T \frac{\partial \mathcal{R}}{\partial X_{CFD_{vol}}}\right] \frac{\mathrm{d}X_{CFD_{vol}}}{\mathrm{d}X_{CFD_{surf}}} \frac{\mathrm{d}X_{CFD_{surf}}}{\mathrm{d}X_{GEO_{surf}}} \frac{\mathrm{d}X_{GEO_{surf}}}{\mathrm{d}X_{DV}}. \tag{4.4}$$

On the flow solver side, the last major issue is how best to deal with the remaining partial derivatives in the flow solver portion of equation (4.4). While it would be possible to combine these derivatives with the final result from the geometry sections using forward mode AD, this would result in a significant amount of additional code in the flow solver sensitivity routines and would also require a significant amount of extra data transfer across the python-fortran interface. As a result, both of these values are calculated in the flow solver using reverse mode AD, which allows most of the routines from the actual adjoint computation to be reused. These partial derivatives are then combined with the adjoint vector ψ such that a single total sensitivity vector $\mathrm{d}C_i/\,\mathrm{d}X_{CFD_{vol}}$ can be passed through the python-fortran interface for use in the final sensitivity calculation.

The last remaining component of the total sensitivity calculation is the relative indexing when multiplying the two halves of the total sensitivities together. This issue originates from the fact that while the flow solver is a parallel code, the geometry and mesh components are not, thus there is not necessarily one-to-one correspondence between the two sets of derivatives. However, because of its parallel nature, there is a global node numbering scheme built into pyNSSUS. This numbering system assigns a global index to each of the nodes in the mesh based on its block and row number in the overall structured multiblock mesh. However, these number are not, by default, available to the geometry portion of the code. To allow the geometry derivatives to access these indices, a function was added to the python wrapper which can extract the global indexing system from the flow solver. This indexing is then available to the geometry portion of the sensitivities, allowing for accurate interfacing of the two sets of derivative components.

Chapter 5

Results

To demonstrate the implementation of the framework and the resulting design variable sensitivities, a series of test cases is evaluated. In the following sections, the test cases are introduced and a series of results are presented, with each section representing a particular class of result pertaining to the ADjoint. A summary of the results and conclusions is presented at the end of the chapter.

5.1 Test Cases

In order to verify the accuracy of the sensitivity implementation and characterize the performance of the aerodynamic portion of the πADO framework, a series of test cases is generated and run. To help characterize the framework's performance, each test case is generated in order to test a specific portion of the framework's capability. In this section, each test case is introduced, including some of their key characteristics. A brief mention is also made regarding the purpose of each test case.

5.1.1 Infinite Wing

The infinite wing test case, shown in Figure 5.1, is a simple, pseudo three dimensional test case. The airfoil is a straight, untapered wing, modeled with inviscid wall boundary conditions on the surface of the wing and symmetry boundary conditions at either end. The mesh used is a single block C-mesh with 9,471 nodes, where the wake cut is located at the trailing edge of the airfoil. The test case was modeled at a Mach number of 0.9 and an angle of attack of 5 degrees. To lend context to the solution, a sample density solution for the mesh is shown in Figure 5.2.

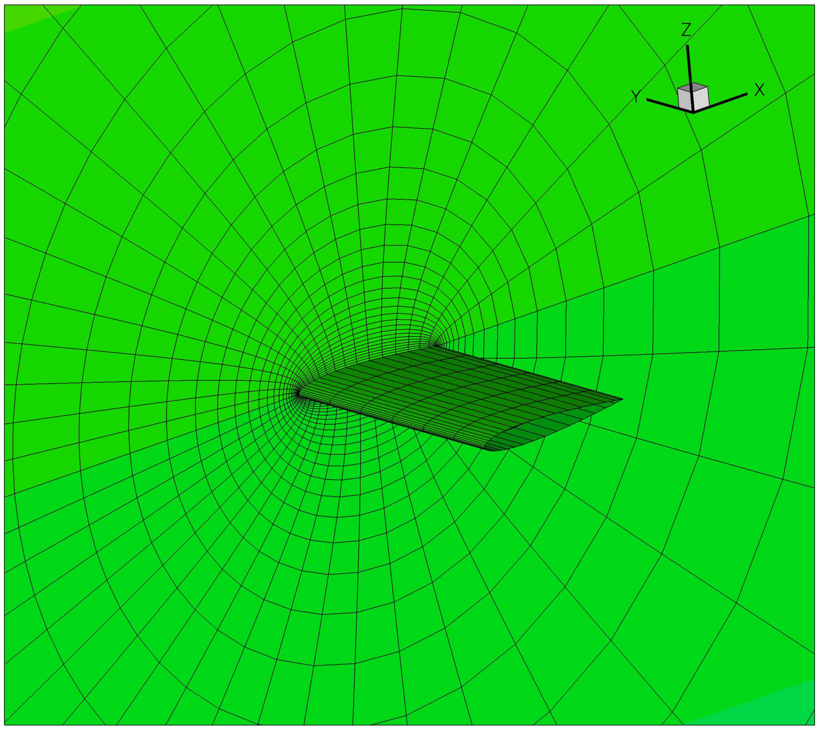

Figure 5.1: Infinite wing computational domain

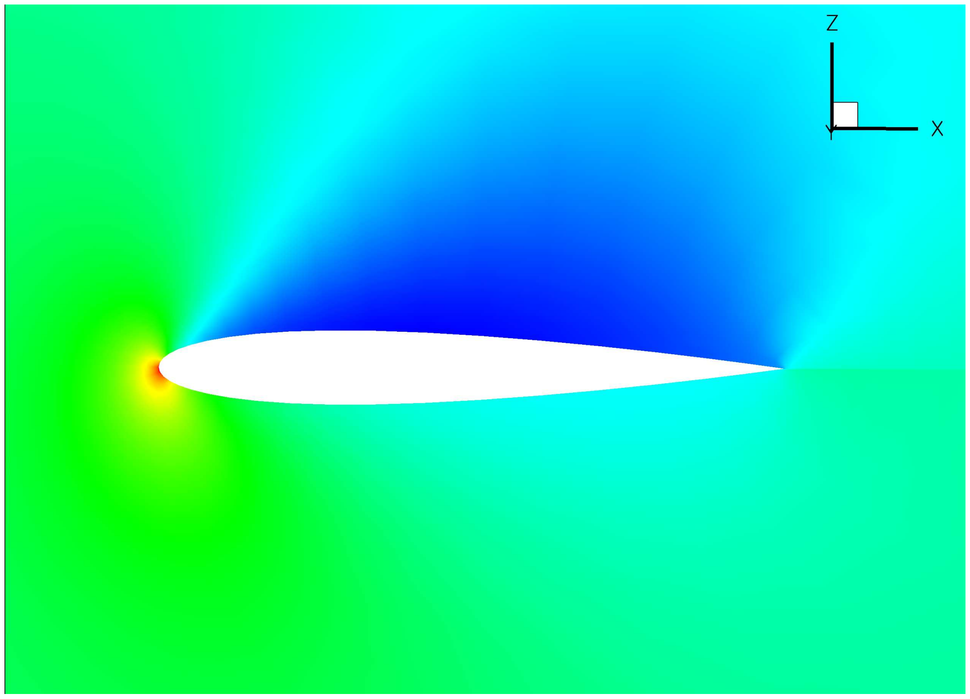

Figure 5.2: Density solution for a cross section of the infinite wing

The primary purpose of this test case is simply to act as a baseline reference case for the flow solver and its associated sensitivities. This test case is not setup to run with the geometry and mesh tools, so it is not capable of testing those portions of the framework. It is also a single block case, so it is not meant to test the multiblock or multiprocessor capabilities of the framework.

5.1.2 Oblique Wing: Coarse Mesh

The second test case used is an oblique wing test case, shown in Figure 5.3. This test case consists of a ten block multiblock mesh with 21,820 nodes. The model itself is an asymmetrically swept, flying wing. The planform is symmetrically tapered, starting with a chord of 1.0m at the tip of the forward swept portion of the wing, growing to a chord of 1.7m at mid span, and returning to a chord of 1.0m at full span. The overall span of the wing is 12.0m when the wing is in its baseline configuration with a mid-chord sweep of 45 degrees. Once again, the surface of the wing is modeled with inviscid wall boundary conditions. However, in this case, because the planform is asymmetric, the entire wing is modeled, so there are no symmetry planes in the mesh. The flow is modeled at Mach 1.2 and Mach 1.5 with angles of attack of 0 degrees and 0.572 degrees. Figure 5.4 shows a pressure solution for this test case with a Mach number of 1.2 and an angle of attack of 0 degrees.

This is a much more complex test case than the infinite wing test case. First of all, it is a multiblock test case, so it can be used to test all of the functionality of the framework for multiblock and multiprocessor cases. Further, it is setup to run with the geometry and mesh tools, so it can test the full aerodynamic framework.

5.2 Accuracy Results

One of the main focuses of this work was to develop a very accurate derivative implementation. Since the analysis tools are capable of achieving very accurate results, it is worthwhile having sensitivity information that is just as accurate, so that the gradient based optimizer used can take advantage of everything the analysis has to offer. The following tables show a series of comparisons between the sensitivities calculated using the framework and sensitivities calculated using the complex-step method with a step size of 10^{-20}, which can be considered to be numerically exact.

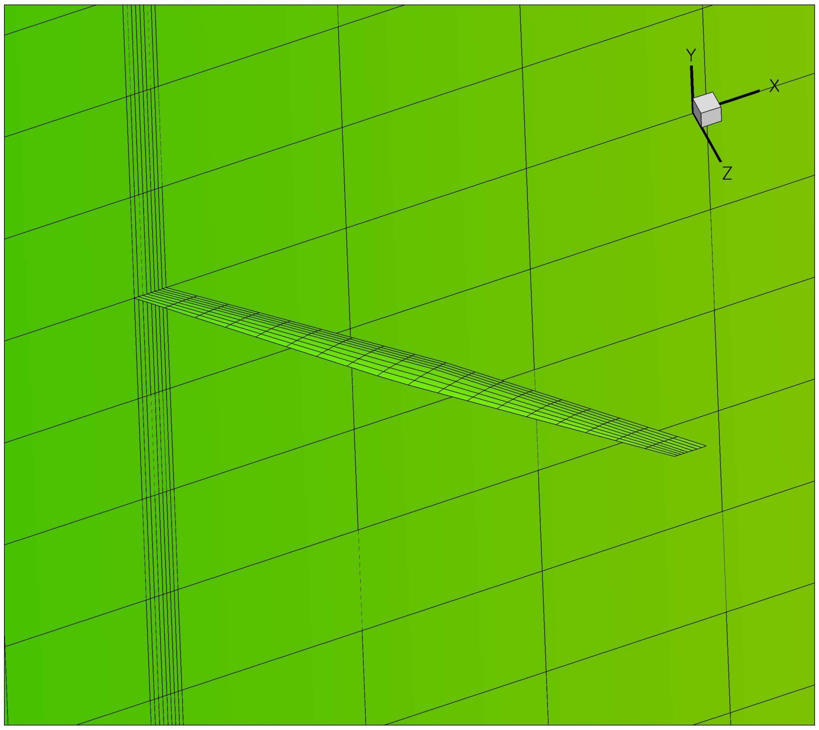

Figure 5.3: Oblique wing: coarse mesh computational domain

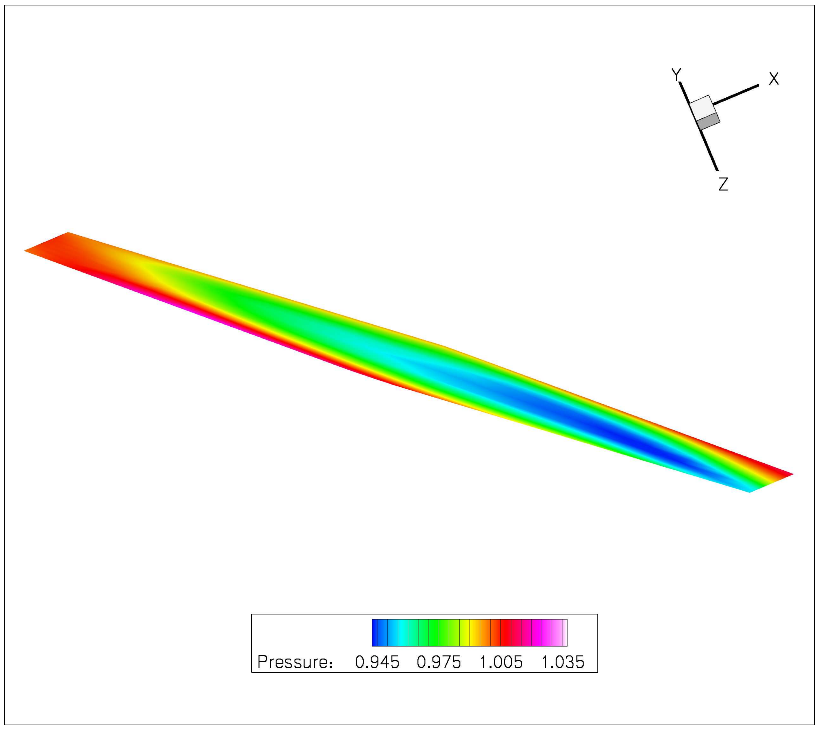

Figure 5.4: Pressure solution for the surface of the oblique wing

The first set of sensitivities, presented in Table 5.1, are from the single-block infinite wing test case. These values represent the ADjoint sensitivities inside the flow solver on its own, without any of the other framework components included. Since these derivatives have been developed for shape optimization, they represent the derivatives of the five force coefficients with respect to the coordinates of the CFD volume mesh. The results

Node index	Coefficient	ADjoint	Complex Step
10,1,3	C_L	1.03737827 *205*E-002	1.03737827 *112*E-2
	C_D	-4.2921761 *724*E-003	-4.2921761 *667*E-3
	C_{Mx}	-3.44421800 *547*E-002	-3.44421800 *306*E-2
	C_{My}	-3.9976406 *323*E-002	-3.9976406 *273*E-2
	C_{Mz}	-1.59291147 *470*E-002	-1.59291147 *259*E-2
10,1,5	C_L	1.03737827 *228*E-002	1.03737827 *112*E-2
	C_D	-4.2921761 *722*E-003	-4.2921761 *667*E-3
	C_{Mx}	-4.5239521 *327*E-002	-4.5239521 *281*E-2
	C_{My}	-3.9976406 *328*E-002	-3.9976406 *273*E-2
	C_{Mz}	-2.551081 *401*E-002	-2.551081 *398*E-2
30,1,3	C_L	1.70505774 *790*E-003	1.70505774 *414*E-3
	C_D	6.21953590 *692*E-003	6.21953590 *701*E-3
	C_{Mx}	-7.1419627 *827*E-003	-7.1419627 *7110*E-3
	C_{My}	-1.8257562646 *240*E-002	-1.8257562646 *434*E-2
	C_{Mz}	1.8512382304 *326*E-002	1.8512382304 *184*E-2
30,1,5	C_L	1.70505774 *756*E-003	1.70505774 *414*E-3
	C_D	6.219535907 *229*E-003	6.219535907 *011*E-3
	C_{Mx}	-1.07832034 *589*E-002	-1.07832034 *431*E-2
	C_{My}	-1.825756264 *730*E-002	-1.825756264 *643*E-2
	C_{Mz}	2.9865698889 *194*E-002	2.9865698889 *621*E-2

Table 5.1: CFD mesh coordinate sensitivity verification: $dJ/dX(i,j,k)$,in the Z coordinate direction.

presented represent four points on the surface of the wing, two on the top surface, two on the bottom surface. As is evident in the table, the agreement between the ADjoint values and the complex-step values is very good, varying between 7 and 11 digits of accuracy for the different cases, with most of the cases giving an 8-9 digit agreement. Considering the requested accuracy of the flow solver was 10^{-10}, this is exceptional agreement.

The next set of results, presented in Table 5.2, again represent the sensitivities of the flow solver alone. However, this table shows values from the multiblock, oblique

Node index	Coefficient	ADjoint	Complex Step
Block 7 4,10,10	C_L	-6.559636127 *405*E-3	-6.559636127 *395*E-3
	C_D	3.5547210577 *330*E-4	3.5547210577 *168*E-4
	C_{Mx}	5.98301242852 *85*E-2	5.98301242852 *43*E-2
	C_{My}	6.8540470166 *758*E-3	6.8540470166 *409*E-3
	C_{Mz}	-6.59618443996 *71*E-2	-6.59618443996 *08*E-2
Block 8 4,1,10	C_L	3.5464900865 *027*E-3	3.5464900865 *147*E-3
	C_D	5.923619700 *3941*E-4	5.923619700 *5154*E-4
	C_{Mx}	-2.855713856 *6782*E-2	-2.855713856 *7358*E-2
	C_{My}	9.58473674 *59268*E-3	9.58473674 *60812*E-3
	C_{Mz}	3.415759755 *6659*E-2	3.415759755 *7290*E-2

Table 5.2: Multiblock CFD mesh coordinate sensitivity verification: $dI/dX(i, j, k)$.

wing test case. The results are again shown for the five force coefficients with respect to the vertical mesh coordinate. Once again points on both the top and bottom of the wing are shown. However, in this case the two surfaces happen to be in different blocks. The results again show very good agreement, this time the agreement between the two methods is between 9 and 12 digits, even better than the single block case. However, in this case both the flow solver and adjoint solver tolerances were set to 10^{-12}, so again, the results are consistent with the requested tolerances.

The final set of accuracy results, shown in Table 5.3 represent the total sensitivity over the entire framework. These sensitivities are again calculated for the oblique wing test case, so these values are an extension of the values shown in Table 5.2. In this case the surface bump for which the sensitivities are shown is on the upper surface towards the leading edge on the forward swept tip of the wing.

The results shown in this table are slightly different than the results in the previous two tables. As in the previous tables, the right hand column lists results for the pure complex-step method, in this case over the entire framework. However, in this table, the left hand column represent results for the two different approaches that were implemented and tested for computing the geometry and mesh sensitivities. The first set of results in the table, labeled FD, represent the result when a finite-difference approach – a forward difference approximation with a step size of 10^{-8} – is used for the geometry and mesh sensitivities. The second set of results, labeled CS, represent the same sensitivities while using the complex-step method to calculate the geometry and mesh sensitivities. As the

Node index	Coefficient	ADjoint	Complex Step
Surface Bump FD	C_L	0.0891981 *19*	0.0891981 *35*
	C_D	0.00026879 *6*	0.00026879 *4*
	C_{Mx}	0.25447 *1920*	0.25447 *2050*
	C_{My}	-0.015799 *501*	-0.015799 *497*
	C_{Mz}	-0.263111 *799*	-0.263111 *927*
Surface Bump CS	C_L	0.0891981358 *57*	0.0891981358 *93*
	C_D	0.0002687945 *45*	0.0002687945 *50*
	C_{Mx}	0.254472050 *640*	0.254472050 *935*
	C_{My}	-0.015799497 *898*	-0.015799497 *751*
	C_{Mz}	-0.263111927 *282*	-0.263111927 *596*

Table 5.3: Multiblock shape variable sensitivity verification

table shows, the accuracy of the derivatives computed with the finite-difference approach is much worse than those computed with the complex-step method. In the case of the finite-differences, a variety of step sizes were tested, with no appreciable increase in accuracy. Thus, in the interest of attaining high accuracy, the complex-step approach was selected for the geometry and mesh sensitivities.

5.3 Timing Results

The second factor of importance when calculating sensitivities for use in gradient-based optimization is the efficiency of the computation. In a typical optimization run the sensitivity computation will be called many times. Thus, an efficient sensitivity calculation is necessary in order allow good performance.

Table 5.4 shows a timing breakdown for the two test cases outlined previously. In both cases, the sensitivity analysis takes less time than the flow solve itself. For the oblique wing test case, it is only slightly faster, with the adjoint solution taking 77% of the flow solution time. However, for the infinite wing test case, it is much faster, with the adjoint solution taking only 9% of the flow solution time. Given the alternatives available for sensitivity analysis, this is a remarkable achievement. With this approach, a large number of sensitivities can be calculated for essentially the cost of a single flow solve. The main reason that the cost of the method is not completely independent of the number of design variables is because the complex-step method is used to compute the geometry

	Oblique Wing	Infinite Wing
No. Nodes	21820	9471
Flow solution	198.78	241.57
ADjoint	152.45	22.08
Breakdown:		
Setup PETSc Variables	0.11	0.03
Compute Jacobian	12.49	5.36
Compute grid partial	13.55	5.73
Compute RHS	0.00	0.00
Solve the adjoint equations	126.22	10.93
Compute the total sensitivity	0.08	0.04

Table 5.4: ADjoint computational cost breakdown (times in seconds)

and mesh handling sensitivities. However, to compute the same set of total derivatives using just the complex-step method or finite differences would require $\mathcal{O}(N)$ flow solves, which would take considerable time to compute. This shows the overall efficiency of the ADjoint approach.

However, particularly in the case of the oblique wing test case, the cost of the linear adjoint solution is a significant portion of the overall sensitivity analysis time. Thus, it would be worthwhile improving the solution techniques used on the linear system to reduce this time. One particular method that has shown promise in this regard is preconditioning. Thus, some effort should be spent on exploring this option.

The other point to note is that the partial sensitivities for both the Jacobian and the mesh are computed in only a small fraction of the flow solution time. This clearly indicates that the AD approach used to generate these values is efficient.

5.4 Conclusions

The results in the previous two sections clearly show the accuracy and efficiency of the ADjoint sensitivities. The accuracy results progress from a simple single block test case for just the flow solver through to a complete multiblock test case for the entire framework. In each case the agreement with the pure complex-step results is excellent, essentially matching the requested precision of the flow solver. The one exception to this is the example where finite differences were used to compute the derivatives of the geometry and mesh sensitivities. As a result, it was decided that using the complex-step method for these portions of the framework was necessary. The timing results are also positive. In each case the ADjoint solutions in the flow solver portion of the framework take less time than the flow solver itself. In particular, the infinite wing test case shows excellent relative performance. In the case of the oblique wing test case, the performance can likely be improved with better preconditioning and matrix ordering. Thus, from the results shown, it is fair to conclude that the ADjoint approach to sensitivity analysis is both efficient and accurate.

Chapter 6

Conclusions and Future Work

In the discussion presented in the previous chapters, two implementations of the ADjoint were presented and discussed. In the first instance, a single block implementation was performed on the cell-centered, finite-volume code, SUmb. This implementation was performed for a few, very simple, derivatives for a limited number of cases, but served to prove the feasibility and accuracy of the ADjoint approach. The second implementation, conducted on pyNSSUS, is a more complete implementation, intended for use in aerodynamic shape optimization. This implementation includes infrastructure to handle multiblock and multiprocessor cases as well as a wide variety of shape variables for design optimization. The ADjoint sensitivities in this implementation compute the derivative of the force coefficients in the flow solver with respect to the CFD volume mesh node coordinate values. These sensitivities are combined with sensitivities from the geometry and mesh portions of the aerodynamic framework to form the total sensitivity matrix required for optimization. The geometry and mesh sensitivities are computed with the complex-step method for accuracy, as semi-analytic methods are not applicable to the relevant portions of the code. Once again, this implementation of the ADjoint is extremely accurate, with the results from the ADjoint matching pure complex-step sensitivity results to the same precision requested from the flow solver. The efficiency of the ADjoint approach in this implementation is also quite good, yielding an adjoint solution in less time than that required to obtain a flow solution. Thus, it is fair to conclude that the ADjoint approach to sensitivity analysis is both accurate and efficient and that it is a practical hybrid approach to computing sensitivity information.

Despite the success of this work, there are still many avenues to be explored regarding the ADjoint and many interesting new possibilities for research opened up by this approach. Specifically regarding the ADjoint, some further work needs to be done experimenting with the linear solution methods used within PETSc. While for the cases tested here, PETSc was able to produce efficient and accurate solutions, that success is by no means guaranteed. Some effort needs to be spent exploring the parameters and options inside PETSc to ensure that the solution methods used are both robust and suitable for a wide variety of test cases. In more general terms, however, there are now a wide variety of avenues open to exploration, primarily those involving optimization. Now that the shape variable sensitivities have been developed, it is possible to explore a wide variety of interesting optimization problems. The most obvious, of course, is aerodynamic shape optimization. This can now be performed for a wide variety of cases both conventional and unconventional. In addition to this, the door is now open to pursue the continued development of the multidisciplinary design optimization (MDO) framework. The aerodynamic derivatives now available are a key enabling component for the development of the coupled adjoint solver, which is itself the key element in performing MDO using a multidisciplinary feasible (MDF) approach. Finally, now that the ADjoint has been proved feasible using the Euler equations, the approach should be extended to the RANS equations. While the Euler equations provide a useful proving ground for new ideas, the additional physics present in a RANS solver is essential in the practical design of aircraft in a modern simulation environment. Thus, this extension is important in order to achieve the high-fidelity design goal originally espoused.

References

[1] Alonso, J. J., LeGresley, P., van der Weide, E., Martins, J. R. R. A., and Reuther, J. J. pyMDO: A framework for high-fidelity multi-disciplinary optimization. *AIAA Paper* 2004-4480, Aug. 2004.

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

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

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

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

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

[7] Carpenter, M. H., Gottlieb, D., and Abarbanel, S. 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 (Apr. 1994), 220–236.

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

[9] Dwight, R. P., and Brezillion, J. Effect of approximations of the discrete adjoint on gradient-based optimization. *AIAA Journal 44*, 12 (2006), 3022–3031.

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

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

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

[13] Hascoët, L. Tapenade: A tool for automatic differentiation of programs. In *Proceedings of 4^{th} European Congress on Computational Methods, ECCOMAS'2004, Jyvaskyla, Finland* (2004).

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

[15] Hicks, R. M., and Henne, P. A. Wing design by numerical optimization. *Journal of Aircraft 15*, 7 (1978), 407–412.

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

[17] Marta, A. C. *Rapid Development of Discrete Adjoint Solvers with Applications to Magnetohydrodynamic Flow Control.* PhD thesis, Stanford University, Stanford, California, 2007.

[18] Martins, J. R. R. A. *A Coupled-Adjoint Method for High-Fidelity Aero-Structural Optimization.* PhD thesis, Stanford University, Stanford, California, 2002.

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

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

[21] Martins, J. R. R. A., Sturdza, P., and Alonso, J. J. The connection between the complex-step derivative approximation and algorithmic differentiation. In *Proceedings of the 39th Aerospace Sciences Meeting* (Reno, NV, 2001). AIAA 2001-0921.

[22] Martins, J. R. R. A., Sturdza, P., and Alonso, J. J. The complex-step derivative approximation. *ACM Transactions on Mathematical Software 29*, 3 (2003), 245–262.

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

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

[25] Nemec, M., and Zingg, D. W. Newton—Krylov algorithm for aerodynamic design using the Navierr–Stokes equations. *AIAA Journal 40*, 6 (June 2002), 1146–1154.

[26] Nielsen, E. J., and Kleb, W. L. Efficient construction of discrete adjoint operators on unstructured grids using complex variables. *AIAA Journal 44*, 4 (2006), 827–836.

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

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

[29] Rall, L. B., and Corliss, G. F. An introduction to automatic differentiation. In *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. H. Bischof, G. F. Corliss, and A. Griewank, Eds. SIAM, Philadelphia, Penn., 1996, pp. 1–17.

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

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

[32] Ruo, S. Y., Malone, J. B., Horsten, J. J., and Houwink, R. The LANN program — an experimental and theoretical study of steady and unsteady transonic airloads on a supercritical wing. In *Proceedings of the 16th Fluid and PlasmaDynamics Conference* (Danvers, MA, July 1983). AIAA 1983-1686.

[33] Squire, W., and Trapp, G. Using complex variables to estimate derivatives of real functions. *SIAM Review 40*, 1 (1998), 110–112.

[34] van der Weide, E., Kalitzin, G., Schluter, J., and Alonso, J. J. Unsteady turbomachinery computations using massively parallel platforms. In *Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit* (Reno, NV, 2006). AIAA 2006-0421.

[35] Wengert, R. E. A simple automatic derivative evaluation program. *Commun. ACM 7*, 8 (1964), 463–464.

Appendix A

Generating an aerodynamic shape optimization test case

The following steps are required to generate an aerodynamic optimization test case from scratch for use with pyAerosurf, pyWarp and pyNSSUS.

- Use pyGeo.py – or some equivalent software – to generate .geo and .inp files to describe the desired geometry.

- Run pyAerosurf to generate a surface mesh, which will be stored in a .xyz file (this is in plot3D format)

- Copy a version of the .xyz file to aero.xy0. This will be used as a reference when aerosurf regenerates the surface mesh after the application of design variables. Note: A parameter must be enabled in the .inp file to use this feature.

- Use this surface file to generate a volume mesh in .cgns format for the flow solver. This can be done with any grid generation tool, however the one used in this work was IcemCFD, created by ANSYS. A description of the process required for generating a compatible grid file in IcemCFD is included in Appendix C

- Generate an input file for the CFD solver, based on the .cgns file created in the previous step.

- Test run the CFD solution to ensure convergence.

- Modify the initialization calls in aerodynamicoptimization.py to specify the files for this test case. Initialization calls for the geometry class and the SUmbSolution class should be modified.
- Run a test of the aerodynamicoptimization.py script by calling: mpirun -np * pyMPI aerodynamicoptimization.py, where * is the number of processors to use.
- Setup an optimization script – use lconst_dragmin.py as a template – using the Aero_optimization class.
- Run the optimization by calling: mpirun -np 1 pyMPI <filename>

Appendix B

List of input files for aerodynamic optimization using πADO

- Geometry – pyAerosurf
  - *.geo – Component section definitions
  - *.inp – Control parameter file
  - *.xy0 – Reference geometry in plot3d format. Dictates patch configuration and size.
- Mesh Warping – pyWarp
  - *.cgns – Same as flow solver
- Flow Solver – NSSUS
  - <filename> flow solver input file
  - *.cgns

Appendix C

Creating a CFD grid using ICEM CFD

- Generate a plot3D surface mesh using Aerosurf
- Open ICEMCFD and start a new project
- Import the the plot3D geometry information by clicking on File-> Import Geometry -> Formatted Point Data. Note: Make sure that the plot3D box at the bottom of the panel is selected.
- Clean up geometry by combining curves and surfaces. e.g. combine all leading edge curves for each wing into a single leading edge curve, combine surfaces together so that there is one surface each for the top and bottom of the wing, etc.. The cleaner the geometry definition is at this point the better/easier the mesh generation process will be. Also It is a good idea to generate features that will be easily associated with different portions of the intended blocking.
- Use ICEM's geometry tools to generate the far-field boundary for the mesh. This can be done by first generating points, connecting them with lines and then combining groups of lines into surfaces.
- Create the desired blocking scheme. This generally starts with one large block, which is then split multiple times to accommodate the desired geometry.
- Associate the vertices and edges of the blocking with the appropriate features on the plot3D surface. This is a very important stage. If these associations are not correct, the resulting mesh topology will end up unusable. Use the pre-mesh visualization to see what the associations that have been made cause the mesh to look like.

- Once satisfied with the association between the blocking and the geometry, add nodes to the various block edges. ICEM will ensure that the number of nodes across the various blocks stays consistent, so it is probably best to start with the partial line segments on the surface of the body(plot3D surface) and work outwards. ICEM has a number of linking and bunching aides to speed this process, but I have found them unreliable when returning to saved projects, so use them with care. When actually specifying the nodes, there are a number of distribution types available, so use your discretion when setting these options. I have had some success with the hyperbolic distribution option. Once again, check the pre-mesh visualization as you go through this process to ensure that the nodes are going where you want them to and to ensure that the mesh topology is what you expect.

- Create families of surfaces that will have like boundary conditions. This will likely include one for the wing/aircraft surface and one for each far-field/symmetry boundary condition. Also create a family for all of the blocks that are internal to the body being modeled and deactivate them. This will prevent this part of the mesh from being exported.

- The model is now ready for export. Under the File -> Blocking menu, select "save multiblock mesh", and save a volume mesh from your model.

- Select the output tab

- Set the solver type as CGNS, leave the common structural solver as NASTRAN.

- Set the boundary conditions for each of the surfaces using the create BC's tab. This will most likely be under surfaces -> mixed/unknown -> create new -> BCType. After creating the data, set the appropriate boundary condition type.

- Export the model. Click the "Write Input button". Set the grid type to structured. The default filenames should be correct. Select 'no' for the "create default BC patches" option and use CGNS library 2.4.

- Once created, use ADFViewer to concatenate the various boundary conditions to a single boundary condition per block face. This really shouldn't be necessary, however at this point it is required to ensure the accuracy of the ADjoint. Further exploration into the ICEM export process may help this.

- create an NSSUS input file to go with the *.cgns file and the mesh should be ready to go.

Appendix D

AD tools for Fortran

1. ADIFOR [6]
2. AD01
3. OPFAD/OPRAD
4. TAF [11],
5. TAMC [12]
6. Tapenade [14, 27]

For a detailed list of automatic differentiation tools for a wide variety of computer languages including Fortran, C/C++ and Matlab go to www.autodiff.org.

Appendix E

Modified Complex Code Sections

E.1 pyAerosurf

E.1.1 Original code segments

```
IF (VS .ceq. ZERO) CYCLE ! Next N

IF (DELTAV /= ZERO) THEN
```

E.1.2 Complexified code segments

```
IF ((VS .ceq. ZERO).and.(aimag(vs).ceq. ZERO)) CYCLE ! Next N

IF ((DELTAV .cne. ZERO).or.(aimag(deltav).cne. ZERO)) THEN
```

E.2 pyWarp

E.2.1 Original code segments

```
            DFACEI(1,J,K) = (ABS (DELJ) * DELJ + ABS (DELK) * DELK) /
     >                           MAX (ABS (DELJ) + ABS (DELK), EPS)
```

E.2.2 Modified code segments

```
              if ((aimag(ABS (DELJ) + ABS (DELK))/=0.0).or.
     >              (real(ABS (DELJ) + ABS (DELK))/=0.0) )then
                  DFACEI(1,J,K) = (ABS (DELJ) * DELJ +
     >                 ABS (DELK) * DELK) /
     >                 (ABS (DELJ) + ABS (DELK))

               else
                  DFACEI(1,J,K) = (ABS (DELJ) * DELJ +
     >                 ABS (DELK) * DELK) /
     >                 (EPS)

               endif
```