University of BRISTOL

Department of Aerospace Engineering
AENGM0032 Research Project
2018-2019

DEVELOPMENT OF A DISCRETE ADJOINT SOLVER

Charlie J. Anderson

Department of Aerospace Engineering, University of Bristol, Queen's Building, University Walk, Bristol, BS8 1TR, UK

Aerodynamic optimisation involves adjusting design parameters in order to find the best design, however this becomes increasingly difficult when many degrees of freedom are added. Adjoint solvers have a computational cost independent of the number of design variables but are complex to develop. This project uses automatic differentiation to simplify the development process. To do this backgrounds to adjoint equations and automatic differentiation are given before the processes of creating primal and adjoint solvers are explained. Results from the code detail the accuracy, temporal cost and memory usage of the solver.

Keywords: Discrete adjoint method, Automatic differentiation, Computational fluid dynamics, Aerodynamic shape optimisation

1 INTRODUCTION

With the high speed and low cost of today's computers it is easier than ever to run computational fluid dynamics (CFD) codes to analyse a given design. Through a combination of in-house, open-source and commercial codes CFD is used as the aerodynamic design tool of choice when optimising a given problem.

While aerodynamic shape optimisation in its most general form is an infinite-dimensional problem, for practical purposes it is reduced to n degrees of freedom and parameterised with a vector of design variables $\boldsymbol{\alpha} \in \mathbb{R}^n$. For example, if parameterising an aerofoil these could represent thickness and camber or be the coordinate points representing the surface. There must also be some way of measuring design quality, here represented by a scalar objective (or cost) function $J \in \mathbb{R}$. This can represent quantities such as lift, drag, range or any other desired metric. Equation 1 shows the standard form for optimisation problems, where $\mathbf{c}$ and $\mathbf{c}_{eq}$ represent vectors of inequality and equality constraints respectively.

$$\begin{aligned} \text{minimise} \quad & J(\alpha_i) \\ \text{s.t.} \quad & \mathbf{c}(\alpha_i) \leq \mathbf{0} \\ & \mathbf{c}_{eq}(\alpha_i) = \mathbf{0} \end{aligned} \tag{1}$$

Current research investigates how to best search a design space in order to find an optimum position. Out of the wide range of possible strategies gradient-based methods use the sensitivity $\boldsymbol{\mathcal{G}} \in \mathbb{R}^n$ (also known as the gradient vector) to move from the current design point to a better one. The explicit definition of this is given in Equation 2.

$$\boldsymbol{\mathcal{G}} = \frac{\partial J}{\partial \alpha_i} \tag{2}$$

While gradient-based methods have yielded good results, finding the sensitivity at each step remains an expensive process. Most current methods have a cost that scales linearly with the number of design variables which is a barrier to problems with many degrees of freedom. Adjoint methods propose a solution to this. By carefully finding sets of partial derivatives and combining them, the sensitivity can be found at a cost that is almost *independent of the number of design variables*. This would clearly increase the speed of many optimisation codes.

The primary drawback of adjoint solvers is the complexity of their development. While there are decades of publications on adjoint methods they are mainly posed as cutting-edge research with novel information as opposed to entry-level guides. The majority of people wanting to create adjoint codes will be researchers able

to extract the necessary information but even they would benefit from well explained documentation.

It is the view of this paper that creating an adjoint solver with today's available technology can be a task no more difficult than creating the primal CFD code, given that appropriate instruction is provided. For this reason, the objective of this final year project is to provide that guidance by including all the information relevant to the development of a discrete adjoint solver.

To do this, the paper will be split into several distinct sections. Section 2 will present a mathematical view of the discrete and continuous adjoint equations, followed by a literature review of their use in research. In order to create a working adjoint solver only the first subsection on discrete equations is needed as the adjoint method itself is very concise (Equations 8 and 9), however by going into more detail and including the continuous formulation the reader can gain a much deeper and intuitive understanding of the adjoint method. By using automatic differentiation (also known as AD, or algorithmic differentiation) implementing the discrete adjoint equations can be made into a quick and easy process, which is why Section 3 is devoted to explaining the topic. Both sections will include a review of the literature detailing both the history and current state of adjoint research in those areas.

The remaining sections of this paper will detail the practical development of a primal CFD code and its corresponding adjoint solver. Given a primal CFD code, this paper will explain how to prepare it for input into an AD engine to produce the required partial derivatives then suggest a range of ways to solve the resulting system. The major benefit of the adjoint method is its performance, so both the time cost and memory requirements will be discussed in detail. The advantages will be shown and drawbacks will have suggested solutions.

All of the code used to produce the results was developed by the author (relevant use of external software such as AD is made clear) so for the interest of the reader all of the source code has been made available online [1]. This will compile and run on an up-to-date Linux system. Also note that while all acronyms, mathematical symbols and novel technical language will be defined in text, a glossary is included in Appendix A as a centralised collection should the reader wish to reference information while reading.

2 ADJOINT EQUATIONS IN SENSITIVITY ANALYSIS

Any optimisation strategy using a gradient-based method requires the sensitivity $\boldsymbol{\mathcal{G}}$. This is usually the gradient vector of partial derivatives. The most common method (see Keane [2] for an overview of sensitivity analysis) of evaluating this is by the method of finite differences. The first-order approximation of the first derivative can be calculated with one extra cost evaluation per design variable (Equation 3) but a second-order approximation requires two extra evaluations per design variable.

$$\frac{\partial J}{\partial \alpha_i} = \frac{J(\alpha_i + \epsilon) - J(\alpha_i)}{\epsilon} + \mathcal{O}(\epsilon) \qquad (3)$$

This can be improved upon by using the complex-step derivative approximation (CSDA) [3] as shown in Equation 4. This can be implemented in a computer program by replacing every real variable declaration with a complex one. Not only does it give second-order accuracy with one function evaluation per design variable, it does not suffer from rounding error during the subtraction stage of the classical finite difference methods, giving almost machine precision for a small enough step size.

$$\frac{\partial J}{\partial \alpha_i} = \frac{\Im[J(\alpha_i + j\epsilon)]}{\epsilon} + \mathcal{O}(\epsilon^2) \qquad (4)$$

However, the CSDA has drawbacks. Not only does it require access to the entire source code, it can also be difficult when evaluating conditional statements and the functions being differentiated must obey the Cauchy-Riemann equations. Even then, the function evaluations are still proportional to the number of design

variables which can be prohibitive for large optimisation problems. These problems can be addressed by making use of adjoint methods.

2.1 Discrete adjoint equations

To form the discrete adjoint equations, introduce the state variable vector $\mathbf{w} \in \mathbb{R}^p$ and residual $\mathbf{R} \in \mathbb{R}^p$ of the governing fluid dynamics equations, where $\mathbf{w} = \mathbf{w}(\alpha_i)$. In this project the governing equations are the two-dimensional Euler equations, but in principle can be anything. For the cost function J to be feasible the fluid equations must be fully converged, represented by Equation 5. Optimisation constraints are not required for the adjoint derivation.

$$\begin{aligned} &\text{cost function} \quad J(\alpha_i, \mathbf{w}) \\ &\text{subject to} \quad \mathbf{R}(\alpha_i, \mathbf{w}) = \mathbf{0} \end{aligned} \tag{5}$$

To find the sensitivity take the *total* derivatives $\frac{dJ}{d\alpha_i} = \boldsymbol{\mathcal{G}}$ and $\frac{d\mathbf{R}}{d\alpha_i} = 0$ of Equation 5. To maintain consistency let $\mathbf{u}_i = \frac{d\mathbf{w}}{d\alpha_i}$, then the sensitivity can be written using Equations 6 and 7:

$$\frac{dJ}{d\alpha_i} = \frac{\partial J}{\partial \alpha_i} + \frac{\partial J}{\partial \mathbf{w}}\mathbf{u}_i \tag{6}$$

$$\frac{\partial \mathbf{R}}{\partial \mathbf{w}}\mathbf{u}_i = -\frac{\partial \mathbf{R}}{\partial \alpha_i} \tag{7}$$

This is known as the **direct method**. This gives the exact sensitivity by finding vector $\mathbf{u}_i$ using pre-calculated partial derivatives and substituting it back into the total derivative equation. However, Equation 7 involves a very expensive system solve which must be done for each α_i. As the number of design variables is often large this is inefficient. Now consider the **adjoint method** as Equations 8 and 9:

$$\frac{dJ}{d\alpha_i} = \frac{\partial J}{\partial \alpha_i} - \mathbf{v}^T \frac{\partial \mathbf{R}}{\partial \alpha_i} \tag{8}$$

$$\frac{\partial \mathbf{R}}{\partial \mathbf{w}}^T \mathbf{v} = \frac{\partial J}{\partial \mathbf{w}}^T \tag{9}$$

This new adjoint equation (Equation 9) with adjoint vector $\mathbf{v}$ is said to be *dual* to the original system in Equation 7 (see Giles and Pierce [4, 5] for further information). As this new adjoint equation has no reference to α_i the expensive system solve is independent from the number of design variables. The equivalence of the direct and adjoint methods can be shown by Equation 10:

$$-\mathbf{v}^T \frac{\partial \mathbf{R}}{\partial \alpha_i} = \mathbf{v}^T \frac{\partial \mathbf{R}}{\partial \mathbf{w}}\mathbf{u}_i = \frac{\partial J}{\partial \mathbf{w}}\mathbf{u}_i \tag{10}$$

This is the linear algebra approach, but the adjoint method can be interpreted in another way, through Lagrange multipliers. Equations 11 and 12 are equivalent to Equations 8 and 9.

$$\begin{aligned} \frac{dJ}{d\alpha_i} = &\left(\frac{\partial J}{\partial \alpha_i} + \frac{\partial J}{\partial \mathbf{w}}\frac{\partial \mathbf{w}}{\partial \alpha_i}\right) \\ &- \boldsymbol{\lambda}^T \left(\frac{\partial \mathbf{R}}{\partial \alpha_i} + \frac{\partial \mathbf{R}}{\partial \mathbf{w}}\frac{\partial \mathbf{w}}{\partial \alpha_i}\right) \end{aligned} \tag{11}$$

$$\frac{\partial J}{\partial \mathbf{w}} - \boldsymbol{\lambda}^T \frac{\partial \mathbf{R}}{\partial \mathbf{w}} = \mathbf{0} \tag{12}$$

The linear algebra approach is as favoured by Giles and Pierce [4] due to clear comparison of direct and adjoint systems, whereas the Lagrange multiplier formulation is preferred by Jameson [6] albeit implemented in the continuous adjoint fashion.

2.2 Further comments on the discrete adjoint equations

While the derivation of the adjoint method is straightforward there are a large variety of methods in which to implement it. The first step is to find the partial derivatives $\frac{\partial J}{\partial \alpha_i}$, $\frac{\partial J}{\partial \mathbf{w}}$, $\frac{\partial \mathbf{R}}{\partial \alpha_i}$ and $\frac{\partial \mathbf{R}}{\partial \mathbf{w}}$. This can be done using finite differences [7] for simplicity when the code structure is complicated or unavailable. Differentiation by hand would yeild a result with no truncation error, but is difficult and prone to mistakes. This method is sometimes used for small sections of adjoint codes, usually for turbulence models. However, one method that is gaining popularity is automatic differentiation [8], the

method used in this project. Due to the importance of this, it has its own devoted discussion covered in Section 3.

Readers may notice some abuse of notation used in the adjoint equations, which is wholly unaddressed in most literature. While most aerodynamicists will say that the sensitivity is the vector of partial derivatives $\boldsymbol{\mathcal{G}} = \frac{\partial J}{\partial \alpha_i}$, the adjoint equations use the vector of total derivatives $\boldsymbol{\mathcal{G}} = \frac{dJ}{d\alpha_i}$ which itself has a non-equivalent term $\frac{\partial J}{\partial \alpha_i}$. The reason for this difference is the completely arbitrary decision of whether $J = J(\alpha_i)$ or $J = J(\alpha, \mathbf{w}(\alpha_i))$. Both of these formulations return the exact same value, hence the $\frac{\partial J}{\partial \alpha_i}$ in Equation 3 is exactly the same as the $\frac{dJ}{d\alpha_i}$ in Equation 8. For a proper understanding of adjoint equations some mechanism must be introduced to highlight the difference between the $\frac{\partial J}{\partial \alpha_i}$ in Equation 3 and the $\frac{\partial J}{\partial \alpha_i}$ in Equation 8.

For the purposes of this paper, the term *inheritance* will be used when $\mathbf{w} = \mathbf{w}(\alpha_i)$, and the term *isolation* will be used when $\mathbf{w} \neq \mathbf{w}(\alpha_i)$. In the normal case $\mathbf{w}$ inherits α_i as the flow solution is a function of the design, meaning a finite difference method does not need to consider $\mathbf{w}$ explicitly. However, the adjoint equations assume that $\mathbf{w}$ is isolated from α_i, which is not the case in most CFD codes. This requires careful use of the $\frac{\partial}{\partial \alpha_i}$ and $\frac{\partial}{\partial \mathbf{w}}$ operators, discussed in more detail later.

At this stage it is useful to abstract the system in order to perform general analysis. The governing equations are written as $\frac{\partial \mathbf{w}}{\partial t} + \mathbf{R} = \mathbf{0}$ with the fluxes contained in the residual $\mathbf{R} = \mathbf{R}(\mathbf{w})$. If $L = \frac{\partial \mathbf{R}}{\partial \mathbf{w}}$ is defined as the Jacobian of fluxes, the residual can be written as $\delta\mathbf{R} = L\delta\mathbf{w} = \mathbf{0}$, or in the quasi-linear form displayed in Equation 13:

$$L\mathbf{w} = \mathbf{0} \tag{13}$$

If source terms $\mathbf{g}^T = \frac{\partial J}{\partial \mathbf{w}}$ and $\mathbf{f}_i = -\frac{\partial \mathbf{R}}{\partial \alpha_i}$ are defined, then the direct and adjoint systems (Equations 7 and 9) can be written as

$$L\mathbf{u}_i = \mathbf{f}_i \tag{14}$$

$$L^T\mathbf{v} = \mathbf{g} \tag{15}$$

Writing Equations 13, 14 and 15 in this form allows application of the theory of linear systems (Equation 16) to explain what before was a complex PDE system and gain an intuitive understanding of how they work.

$$\textit{dynamics} \times \textit{solution} = \textit{source} \tag{16}$$

The dynamics for all of these are dictated by the flux Jacobian L, which is the linearisation of the fluid equations. As matrix transposition does not affect the eigenvalues these linear systems will all have similar behaviour, which includes when they are being solved.

There exist other intuitive interpretations of the adjoint system. Note that the order of vectors $\mathbf{u}_i$ and $\mathbf{v}$ often have structures. This could be cell position in CFD or the solution over time for other PDE systems (this is often used as an example in adjoint literature). By transposing the matrix the solution-source mapping is reversed so the propagation of information in an adjoint system is the reverse of that in a direct one. This reversal is a common theme.

Now attention can return to the practical implementation of the adjoint method. Once the partial derivatives have been found they need to be applied to Equations 8 and 9. Equation 8 is a simple case of multiplying matrices and adding vectors, but the solution of the linear system in Equation 9 is a little trickier. The classical approach to solving the system is using pseudo-timestepping. In the same way that the governing equations are solved with time integration of the residual, the direct and adjoint systems can also be solved using iterative methods [4, 6, 9, 10] as shown in Equations 17 and 18 by driving down an *adjoint residual* and converging $\mathbf{v}$ to its true solution.

$$\frac{\partial \mathbf{u}_i}{\partial t} + L\mathbf{u}_i = \mathbf{f}_i \tag{17}$$

$$-\frac{\partial \mathbf{v}}{\partial t} + L^T\mathbf{v} = \mathbf{g} \tag{18}$$

Note that time is reversed in the adjoint formulation. While this is a common approach, there are simpler ways of solving Equation 9. As it is a matrix-vector linear system direct factorisation is possible but becomes prohibitively expensive as the mesh density (and hence the flux Jacobian) increases. This can be addressed by using methods designed to solve sparse matrices, such as a Krylov solver.

2.3 An introduction to the continuous adjoint equations

The discrete adjoint equations (DAEs) are derived by creating a dual system from equations that have already been discretised, whereas the continuous adjoint equations (CAEs) are formulated by creating a dual system from linearised PDEs first *then* discretising them. While the DAEs are the focus of this report an understanding of the CAEs gives a much deeper understanding of the theory of adjoint systems and are still used in research. For this reason the topic is introduced here. Note that while the discrete formulation creates a system dual to the *sensitivity* the usual continuous formulation studies an adjoint *flow solution* which can then be used to find the sensitivity. While this may not immediately seem useful the CAEs can be used as a more mathematically rigorous tool used in theory. The example studied here is taken from a paper by Giles and Pierce [5].

Define the inner product, denoted by the angle brackets $\langle , \rangle : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, as a function that takes the dot product of two vectors. Two new functions are then defined as volume and surface integrals over this inner product on domain Ω and boundary $\partial\Omega$ respectively. This is shown in Equation 19 for the volume integral case and is labelled by the subscripts.

$$\langle \mathbf{V}, \mathbf{U} \rangle_\Omega = \int_\Omega \mathbf{V}^T \mathbf{U} dV \tag{19}$$

Consider Equation 20, the primal representation of the cost function. Unlike Equation 6, this is not the derivative with respect to α_i. This formulation uses the primal state variable vector $\mathbf{w}$ flow solution rather than the differential $\mathbf{u}$. Boundary operators B and C are defined in addition to system operator L. In the general case, these are not matrices. f, e, g and h are the respective source and objective vectors.

$$\begin{aligned} \text{determine} \quad & J = \langle \mathbf{g}, \mathbf{w} \rangle_\Omega + \langle \mathbf{h}, C\mathbf{w} \rangle_{\partial\Omega} \\ \text{given that} \quad & L\mathbf{w} = \mathbf{f} \quad \text{in } \Omega \\ \text{and} \quad & B\mathbf{w} = \mathbf{e} \quad \text{on } \partial\Omega \end{aligned} \tag{20}$$

Now consider the dual system as shown in Equation 21, now with an introduction of the adjoint flow solution $\boldsymbol{\nu}$ and adjoint operators L^*, B^* and C^*.

$$\begin{aligned} \text{determine} \quad & J = \langle \boldsymbol{\nu}, \mathbf{f} \rangle_\Omega + \langle C^*\boldsymbol{\nu}, \mathbf{e} \rangle_{\partial\Omega} \\ \text{given that} \quad & L^*\boldsymbol{\nu} = \mathbf{g} \quad \text{in } \Omega \\ \text{and} \quad & B^*\boldsymbol{\nu} = \mathbf{h} \quad \text{on } \partial\Omega \end{aligned} \tag{21}$$

The process of finding the dual operators forms much of the work when using CAEs. For example, first derivatives become negative. This reversal in time and space means the direction of information propagation and convective fluxes are reversed. The equivalence of the two results can be shown using Equation 22:

$$\begin{aligned} &\langle L^*\boldsymbol{\nu}, \mathbf{w} \rangle_\Omega + \langle B^*\boldsymbol{\nu}, C\mathbf{w} \rangle_{\partial\Omega} \\ &\quad = \langle \boldsymbol{\nu}, L\mathbf{w} \rangle_\Omega + \langle C^*\boldsymbol{\nu}, B\mathbf{w} \rangle_{\partial\Omega} \end{aligned} \tag{22}$$

Given these definitions of dual systems, properties of the adjoint flow solution (for example, consider adjoint density compared to the primal density) can be explained. A highly studied example [4, 11, 12] is of the Euler equations through duct flow. In the case where shocks and contact discontinuities are produced, it is found that the adjoint solution is actually continuous through the shocks and discontinuous through the contact discontinuities. This requires an additional internal boundary condition to be enforced, which is difficult as knowledge of the shock location is required. Despite this continuity at shocks the adjoint solution can also contain singularities under certain conditions in places where the primal solution is

continuous, such as at the sonic point in a duct or the stagnation point on an aerofoil.

While the CAEs can provide insights into the behaviour of the adjoint equations they are much harder to implement and can have badly behaved solutions in areas of high non-linearity if improperly treated. As the DAEs are much easier to implement they have become far more commonplace in recent research.

2.4 Adjoint methods in research

Adjoint equations have been used for a long time to treat partial differential equations and over the last four decades they have become an increasingly common tool for scientific design. They are especially popular in the aerodynamics community, where adjoint methods have progressed from two-dimensional fluid mechanics problems to optimisations of entire aircraft configurations. A reader interested in this history is directed to the summary by Newman et al. [13].

Due to the complexity of the governing equations, computational fluid dynamics calculations have always been computationally expensive. In the 1980s computers became fast enough at low prices to be of use to researchers and the field of aerodynamic design (known today as optimisation) took off. Simple two-dimensional cases were handled without too much issue but implementing complex three-dimensional configurations was an issue. While these complex shapes can be handled by unstructured meshes, the flow solvers are more complex, and the flow solution is of lesser quality than a structured mesh. Finding the sensitivity was also a problem as finite differences were expensive and provided little insight into nonlinear phenomena. To address this researchers tried developing adjoint solvers to gain sensitivities for design.

The first use of the adjoint method in a fluid mechanics design problem was by Pironneau in 1974 [14], but the first application to aerodynamic design was by Jameson in 1988 [15]. These approaches used optimal control theory for systems governed by PDEs, as detailed by Lions [16], but resemble what is known today as the continuous adjoint method. After this initial groundwork, adjoint methods in CFD were thoroughly developed during the 1990s.

Eleshaky and Baysal did work on a 'quasi-analytical' alternative to finite differences in 1992 [17] on a simplified scramject-afterbody configuration solving two-dimensional Euler equations. Later they did work on a full three-dimensional multiblock mesh in 1994 [18], designing an axisymmetric nacelle near to a flat plate wing.

Many of the landmark papers developing the adjoint method were authored by Antony Jameson or James Reuther, often working together. After his initial 1988 paper on design for transonic flow, Jameson and Reuther released a similar work in 1994 [19], but using the adjoint method in a more recognisable form than the orinal version based off of control theory. Jameson uses continuous adjoint methods, but favours the Lagrange multiplier formulation as opposed to the dual system version. They also applied this method to a wing-body supersonic optimisation [20]. In 1996 they developed this method much further [21] and used a multiblock mesh to design complex aircraft configurations using the adjoint method. This was later extended to produce a study [22, 23] examining the effects of parallel computers.

In 2004, Nemec et al. [24] performed a multi-point and multi-objective aerodynamic shape optimisation using a discrete adjoint formulation. In the same year Martins et al. [25] released a paper examining aero-structural coupling.

While most of the previous examples used structured meshes in their work, research on unstructured meshes is vital due to their far greater ability to represent complex shapes. Based off of work from previously derived optimality conditions, Newman et al. [26] use the adjoint method on a two-dimensional unstructured grid. With other authors, Newman extended this approach to three dimensions in 1996 [27] and again in 1997 [28] demonstrating sensitivity analysis on a two-dimensional multielement aerofoil and a three-dimensional Boeing 747-200 aircraft.

A widely influential work on adjoint equations in unstructured meshes is a well written paper by Elliott and Peraire in 1996 [9] for the inverse design problem on a multielement aerofoil. This work was expanded on two years later [29] to include viscous effects.

While most of these works have been a 'means-to-an-end' application of the adjoint method to test cases, some authors have been pursuing a more mathematical research approach to understand the theoretical side of the adjoint equations. The most prolific of these have been Mike Giles and Niles Pierce releasing many influential works [4, 5, 10, 11] which have all been useful in this project. In 2012 Alauzet and Pironneau [12] also released a paper containing much useful theory comparing the continuous and discrete adjoints on the Euler equations.

The papers mentioned here have shown examples of the adjoint method applied to most CFD test cases, ranging from simple two-dimensional aerofoil sections to large complex three-dimensional configurations. However, there still remain problems where the adjoint method is still being researched.

Work on creating adjoint turbulence models is still ongoing. This is mainly because work on turbulence models themselves are not well understood, but other complications arise due to the fundamental nature of turbulence. Finding the derivative of code with many conditional statements can be demanding and the proving the feasibility of converging adjoint solutions is nontrivial due to the velocities on all time and length scales. In 1997 Anderson and Bonhaus differentiated a one-equation turbulence model [30]. Due to poor results using other methods, much of this was done by hand, resulting in tedious work but good results. Hand derivation for turbulence models is a common theme, with Driver and Zingg [31] also doing this in 2007, however in 2013 Lyu et al. used AD to find the adjoint of Reynolds-averaged Navier-Stokes that uses a Spalart-Allmaras turbulence model [32].

There is also ongoing work on unsteady adjoint codes, examples of which are presented here. In 2012 Krakos [33] wrote a Ph. D. thesis on the accuracy for adjoint solutions for unsteady and adaptive meshes. In 2015 a continuous adjoint formulation for an unsteady problem solving the RANS equations was derived by Economon et al. [34] and in 2019 Thomas and Dowell [35] used a discrete adjoint formulation for an unsteady aeroelastic problem.

This concludes the first literature review on adjoint methods in research, ranging from basic continuous adjoint methods with potential flow to complex three-dimensional shapes analysed with turbulence models. While these papers present good results, many of them require a deep knowledge of the adjoint theory and involve much manual intervention in the codes. This is the cause of the reputation given to adjoint methods for being complex to develop. There is, however, a solution: AD.

While AD has its own complications and is still the subject of intense research today, it does allow a researcher with very little knowledge of adjoint equations to create an adjoint solver with relatively little effort. Because of its importance to this project, it has been given its own section to deal with the theory, and neglected in this literature review in order to have a more focused review later.

3 AUTOMATIC DIFFERENTIATION

Automatic differentiation (also known as AD or algorithmic differentiation) is a process used to calculate the Jacobian of a given computer program. The core idea is that all computer programs are just series of simple operations such as $+, -, *, /$ or $\exp(), \sin(), \log()$ which can be analytically differentiated no matter how complex the final result. For this reason it is most effective when applied to a completely self-contained program with no external 'black-box' functions which cannot be perfectly differentiated. AD has two primary modes - the forward mode, which is most efficient when a function has many outputs, and the reverse mode, which is most efficient when a function has many inputs. This latter effect is called the cheap gradient result.

There are two ways of automatically differen-

tiating a program. The most common method is source code transformation. This takes in the primal source code and returns new source code that produces the Jacobian of the specified variables. While development of the AD engine itself is difficult, it is very easy for the user to use. The other method of AD is operator overloading which is done by defining a new variable type that contains both the original number and its derivative. As all of the work is done by the compiler operator overolading can result in much simpler implementation, however it is currentlly much slower than source code transformation, especially for reverse mode.

AD has been developed for decades. In 1964, Wengert published a thesis considered to be the first publication on forward mode AD [36] with work on reverse mode published by Speelpenning in 1980 [37]. Both of these works are based off efforts by other researchers in the wider community. These were groundbreaking research, but a seminal work by Rall was published in 1983 [38]. This, however, still had no reference to reverse mode as it was still being developed. Today most researchers who develop or use AD reference the works of Griewank [39, 40].

While the theory has been developed for many years use of AD outside of computer science departments was limited initially due to a lack of readily available AD software. However in the 1990s this was changed with the introduction of practical AD engines. Here the focus is on those that use FORTRAN code due to its prevalence in the CFD community. Examples include TAF, ADIFOR and Odyssée.

The AD engine used in this report is Tapenade [41], the successor to Odyssée, developed at INRIA since 1999. Tapenade uses source code transformation to find the Jacobian of code written in FORTRAN 77, FORTRAN 95, C and C++.

AD is useful not only for calculating an exact derivative with no truncation errors, but for its high speed. If the number of input variables is very different to the number of output variables then the correct use of the *forward* or *reverse* mode can increase the speed by calculating multiple derivatives *at the same time*. This is discussed in the next section.

3.1 Front End: How to use AD

To introduce AD a *front end* description will be given for knowledge of how to use AD and what the effects are, with a *back end* description to explain how it works.

Consider a function $\mathbf{F}(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with n inputs and m outputs. The goal of AD is to calculate the system Jacobian $\mathcal{J} \in \mathbb{R}^{n \times m}$, as shown in Equation 23. Also define direction vectors (also known as *seeds*) $\boldsymbol{\mathcal{U}}_i \in \mathbb{R}^n$ and $\boldsymbol{\mathcal{V}}_j \in \mathbb{R}^m$, where every element is zero except the i^{th} or j^{th} component, which is one.

$$\mathcal{J} = \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \tag{23}$$

When measuring the efficiency of computers calculating the Jacobian there are two useful measures as defined by Verma [42]: Temporal complexity $\omega()$ also known as *work cost* and spatial complexity $S()$ also known as *memory cost*.

When calculated via a first-order finite difference calculation of the Jacobian is expensive as it is proportional to the product nm as seen in Equation 24. Even in the best case scenario of one input or output variable it is quite expensive.

$$\begin{aligned} \omega(\mathbf{F}, \mathcal{J}) = &\ (n \cdot m + 1) \cdot \omega(\mathbf{F}) \\ S(\mathbf{F}, \mathcal{J}) = &\ S(\mathbf{F}) \end{aligned} \tag{24}$$

The most intuitive mode of AD is the forward mode, which calculates the product $\mathcal{J}\boldsymbol{\mathcal{U}}_i$ as shown in Equation 25, which is equivalent to a column of the Jacobian. Forward mode AD calculates the derivative of all of the outputs with respect to one of the inputs *at the same time*, meaning the cost is independent of the number of output variables (Equation 26, proportionality constant α_1 ranges from one to three).

$$\begin{bmatrix} \dot{F}_1 \\ \dot{F}_2 \\ \dot{F}_3 \end{bmatrix} = \begin{bmatrix} . & \frac{\partial F_1}{\partial x_2} & . & . \\ . & \frac{\partial F_2}{\partial x_2} & . & . \\ . & \frac{\partial F_3}{\partial x_2} & . & . \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{25}$$

$$\begin{aligned} \omega(\mathbf{F}, \mathcal{J}) = &\ \alpha_1 \cdot n \cdot \omega(\mathbf{F}) \\ S(\mathbf{F}, \mathcal{J}) = &\ S(\mathbf{F}) \end{aligned} \tag{26}$$

This is also known as the tangent mode. For a parameterised curve $\mathbf{r}(t) \in \mathbb{R} \rightarrow \mathbb{R}^m$ the tangent vector is defined as $\mathbf{T}(t) = \frac{d\mathbf{r}}{dt}$, which is clearly equivalent to a single row of the Jacobian.

The other mode of AD is the reverse mode, which calculates the product $\mathcal{J}^T \boldsymbol{\mathcal{V}}_j$ as shown by Equation 27, which is equivalent to a row of the Jacobian. Similarly to its counterpart, reverse mode calculates the derivative of one output with respect to all of the inputs *at the same time*. Despite a larger proportionality constant from the costs in Equation 28 (α_2 is in practice five to ten) the temporal complexity of reverse mode is *independent of the number of input variables*. The major drawback of this is the large memory requirements. Reverse mode requires storage of all of the intermediate variables, so $\beta =$ `#_inter_vars`.

$$\begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \bar{x}_3 \\ \bar{x}_4 \end{bmatrix} = \begin{bmatrix} . & . & . & . \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \frac{\partial F_2}{\partial x_3} & \frac{\partial F_2}{\partial x_4} \\ . & . & . & . \end{bmatrix}^T \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{27}$$

$$\begin{aligned} \omega(\mathbf{F}, \mathcal{J}) = &\ \alpha_2 \cdot m \cdot \omega(\mathbf{F}) \\ S(\mathbf{F}, \mathcal{J}) = &\ \beta \cdot S(\mathbf{F}) \end{aligned} \tag{28}$$

A row of the Jacobian is equivalent to the gradient of a given output variable ∇F. In the case of most aerodynamic optimisation problems, not only is the gradient a much more useful value than the tangent but $n \gg m$, meaning this can be expensive. For this reason, reverse mode shows a lot of potential despite its prohibitive memory requirements.

$$\langle \mathcal{J}\boldsymbol{\mathcal{U}}_i, \boldsymbol{\mathcal{V}}_j \rangle = \langle \boldsymbol{\mathcal{U}}_i, \mathcal{J}^T \boldsymbol{\mathcal{V}}_j \rangle \tag{29}$$

It is also known as the adjoint mode as the inner product equivalence shown in Equation 29 is in the same form as Equations 10 and 22.

3.2 Back End: Theory of how AD works

Although the details important to the user have been discussed in the previous section, knowledge of the inner workings of AD can be useful, especially if technical work is being done.

Equation 30 shows $\mathbf{F}$ as a composite function, made of smaller elemental functions ϕ. In Tapenade, each one of these corresponds to an instruction of the computer program as traced in the run-time [41].

$$\mathbf{F} = \phi_p \circ \phi_{p-1} \circ \ldots \circ \phi_1 \tag{30}$$

By using the chain rule it is found that the differential of this function (Equation 31) is simply the product of the elemental Jacobians, which are matrices in the general case.

$$\mathcal{J} = \phi'_p \cdot \phi'_{p-1} \cdot \ldots \cdot \phi'_1 \tag{31}$$

In AD the Jacobian is never formed directly, but instead the product with a directional vector. In forward mode the product $\mathcal{J}\boldsymbol{\mathcal{U}}_i$ is needed. Here, the primal solution is run in parallel with the derivative, as seen in Figure 1. More details of this process can be found in Hoffmann [43]. In order to reduce cost, the product between $\boldsymbol{\mathcal{U}}_i$ and the Jacobian in Equation 31 is evaluated from right-to-left in order to do vector-matrix products rather than the more expensive matrix-matrix products.

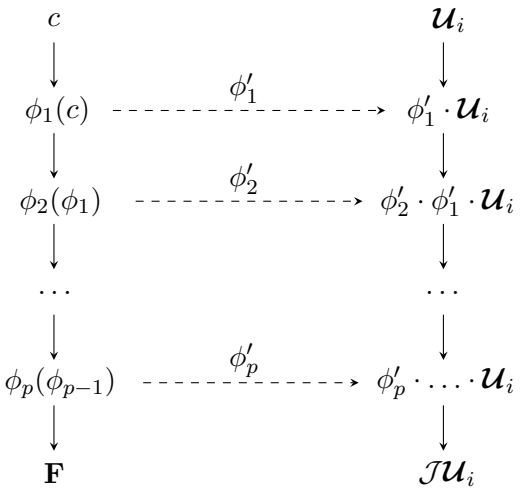

Figure 1: Propagation of information in forward mode. Adapted from [43]

In contrast, the reverse mode runs the primal calculation to completion, storing all of the intermediate variables as seen in Figure 2. The product $\boldsymbol{\mathcal{V}}_j^T \mathcal{J}$ is then evaluated left-to-right in order to perform the same favourable vector-matrix products. While reverse mode shows great potential, an efficient practical implementation remains a challenge as the values needed for calculation are required in an *inverse* order in which they are calculated. There are currently two main strategies [41] of retrieving these values.

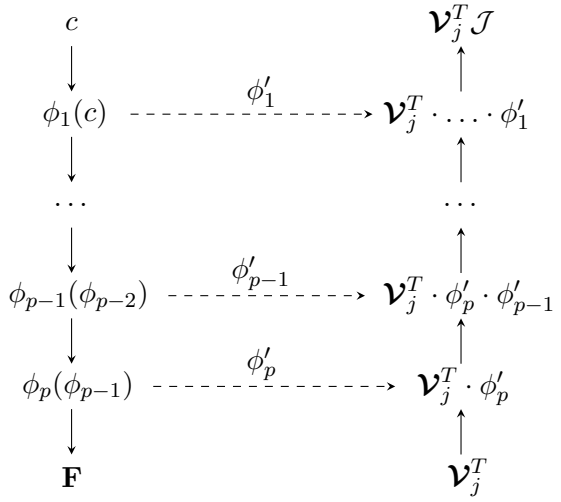

Figure 2: Propagation of information in reverse mode. Adapted from [43]

In the *recompute all* strategy, each value is recomputed every time it is needed, however this has a quadratic temporal complexity with respect to number of instructions p. The *store all* strategy stores all values and retrieves them when needed, with a linear spatial complexity with respect to p. Currently the most efficient implementation is a combination of these two approaches, a method known as *checkpointing* [44].

3.3 AD in adjoint research

The properties of AD are very beneficial to researchers developing adjoint code. Not only is the process quicker than finite differences due to the information propagation, the automatic process returns nearly exact derivatives. When using AD the converged solution can be used to find the partial derivatives with a single function evaluation, rather than another iteration process as with finite differences. AD is especially beneficial during the implementation process of the discrete adjoint equations (despite being possible to use with the continuous formulation). This is because a primal CFD code would only need minor modifications before being converted to a discrete adjoint solver as AD would do all of the work to do with calculating partial derivatives.

While many of the 'landmark' adjoint solvers mentioned in the previous section were first developed in the 1990s and early 2000s the majority of research using AD has lagged behind until the relevant software became more freely available. Examples of early uses of AD in aerodynamic sensitivity analysis include Sherman et al. in 1996 [45] and Mohammadi in 1997 [46]. In the following years many more researchers would also use AD due to the many advantages. A selection of works shall be dicussed here to highlight the trends and current state of the research field with use of AD-derived adjoint CFD codes.

A current researcher who authors and contributes to adjoint research with a focus on AD is Jens-Dominik Müller of Queen Mary University of London. After his initial work on discrete adjoint algorithms with Giles and Pierce in 2003 [10], Müller released work in 2005 detailing the performance of AD-derived discrete adjoint CFD codes [47].

He has also co-authored much work to do with AD in adjoint methods. In 2011 Müller et al. released a set of papers on how to prepare, assemble and verify AD-derived discrete adjoint codes [48, 49]. These papers detail how to use Tapenade with FORTRAN code in detail, making them especially relevant to this project. In more recent years has Müller co-authored papers covering advanced work. In 2015 Xu et al. [50] released research on the stabilisation of steady discrete adjoint solvers derived using AD, allowing the method to be applied to applied to a wider range of flow regimes. In 2019 Hückelheim and Müller released several papers, using AD on an unstructured parallel CFD solver using the reverse mode [51] and addressing the large storage requirements for unsteady flows [52] by using interpolation on selectively stored data.

While most of the papers referenced here aided the project, the most useful paper by far was the publication detailing the ADjoint (**A**utomatic **D**ifferentiation Ad**joint**) approach, written by Mader and Martins in 2008 [8]. The ADjoint method details an overview of the process in which AD is selectively applied to the primal code in order to produce the partial derivatives, which was also the approach taken in this project. Later Mader and Martins would use the ADjoint approach [53] to calculate stability of arbitrary aircraft configurations.

Out of all the different forms of adjoint research, this project will most closely follow the papers as referenced in this AD section.

4 IMPLEMENTATION: CFD CODE

The aim of this report is to describe in detail the development of the adjoint method such that a reader unfamiliar with the concept could go on to create one for themselves. However before an adjoint solver can be developed a primal CFD code must first be created. This is covered in the next section for a number of reasons. First of all a CFD code can be written specifically for an easy conversion to an adjoint code at a later stage. Second of all some readers may be want to create an adjoint code but be inexperienced with creating a CFD code in the first place.

For this project a CFD code was written independently by the author to provide a basis for the following adjoint solver. The code solves the 2D Euler equations using a finite volume scheme with JST [54] dissipative terms, resulting in inviscid results with compressible effects. The source code is available online [1] for any interested readers and compiles and runs on a Linux system.

This code had to both be efficient enough to solve the Euler equations in a reasonable time and be simple enough to convert to adjoint code. A detailed overview of this can be found in a paper by Jones et al. [48]. For this reason some advanced features such as dynamic memory allocation were avoided in order to make the AD process work better.

The code was written in FORTRAN 77, allowing compatibility with pre-existing CFD codes and useful advanced techniques such as the CSDA should any future development be done. A modular program structure was used for clean code writing and a straightforward application to AD.

4.1 The Euler equations

The Euler equations (Equations 32 and 33) are a first-order system of hyperbolic non-linear partial differential equations. Here they have been solved in the conservation formulation.

$$\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = 0 \tag{32}$$

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho E\end{bmatrix} + \frac{\partial}{\partial x}\begin{bmatrix}\rho u\\ \rho u^2 + p\\ \rho uv\\ u(\rho E + p)\end{bmatrix} + \frac{\partial}{\partial y}\begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2 + p\\ v(\rho E + p)\end{bmatrix} = 0 \tag{33}$$

Flux terms $\mathbf{F}$ and $\mathbf{G}$ are functions of the state vector $\mathbf{w}$. The pressure can be found in terms of $\mathbf{w}$ using Equation 34.

$$p = (\gamma - 1)\left[\rho E - \frac{\rho}{2}(u^2 + v^2)\right] \qquad (34)$$

4.2 Meshing

A suitable combination of the flow solver and the necessary boundary conditions can simulate the aerodynamics of many problems, but some test cases are more suited to demonstrating the adjoint method than others. The best test case would be where a large number of design variables can create a shape with little complexity. For this reason it was decided to use flow through a duct as a test case as opposed to flow over an aerofoil. In order to create a simple flow solver and to improve flow capture a structured mesh was used.

Nominally the cross sectional area, referred to here as height due to the two-dimensional equations, was set as a $\cos^2()$ shape. This gave a mesh a throat that introduced compressible flow effects. While it is possible to achieve this by specifying the height as an analytical function this would not allow the introduction of many design variables. To solve this problem, the height was defined by a cubic spline with control points initially distributed in the desired shape. This allowed a large number of design variables to locally control the mesh shape.

$$\phi(x) = \frac{1}{h(x)} + e^{-(x-x_0)^2} \qquad (35)$$

This defined the height of the mesh but there was also an attempt to improve the distribution of the x-positions of the nodes. For a specified number of cells in the i direction it is desired to have a greater concentration of points at the position where a shock was located in order to improve the solution capture and also where the duct area is smaller to keep a consistent aspect ratio. To do this, a probability density function ϕ was created as seen in Equation 35. This was then integrated into a cumulative distribution function between zero and one then seeded in order to recover the x-positions of the points.

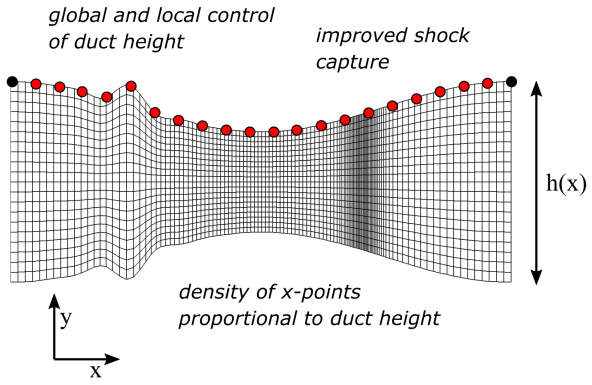

Figure 3: Mesh, show: h(x), spline CPs, local perturbations, x-spacing at throat and shock

An example of the final mesh can be seen in Figure 3 with local control over the height and distributed x-points.

4.3 Boundary conditions

As with all PDE problems the solution is defined by the boundary conditions. In this situation there are three types of boundary conditions: inflow, outflow and solid wall. All variables were non-dimensionalised for simplicity and two layers of halo cells were needed for the dissipative terms.

The inflow and outflow boundary conditions were chosen to be nonreflecting. As this is internal duct flow these were different. A constant pressure ratio was enforced between either end with an inflow Mach number specified. All other states at the outflow were the transient values as calculated by the scheme.

As a two-dimensional inviscid problem a no normal flow condition was enforced at the wall. This was done by specifying a new velocity in the halo cells such that the component normal to the solid wall is equal and opposite to the normal component in the nearby body cells, while the tangential components are identical.

4.4 Residual calculation

A cell-centred finite-volume scheme was used to calculate the convective Euler fluxes. For simplicity the value of the state variable vector at each face was the average from each neighboring cell (Equation 36 shows the calculation for

a given face), which was then used to calculate the fluxes F_k and G_k at each face.

$$w_{i+\frac{1}{2},j} = \frac{1}{2}(w_{i+1,j} + w_{i,j}) \tag{36}$$

However this scheme is implicitly unstable so requires some dissipative terms to stabilise the scheme, as seen in Equation 37. The cell volume is represented by Ω and the operator $Q()$ calculates the residual via the normal finite volume method in Equation 38. This integrates the fluxes around a cell with N faces for each equation of the Euler system.

$$\Omega\frac{\partial w}{\partial t} + Q(w) - D(w) = 0 \tag{37}$$

$$Q(w) = \sum_{k=1}^{N} \begin{bmatrix} F \\ G \\ 0 \end{bmatrix}_k \cdot \begin{bmatrix} \Delta y \\ -\Delta x \\ 0 \end{bmatrix}_k \tag{38}$$

The dissipative terms used are from the Jameson-Schmidt-Turkel (JST) paper [54]. The timestepping scheme used was first-order Euler time integration instead of the Runge-Kutta scheme mentioned in the paper. The dissipation operator is shown in Equation 39 and requires terms to be calculated at each face.

$$D(w) = d_{i+\frac{1}{2},j} + d_{i,j+\frac{1}{2}} - d_{i-\frac{1}{2},j} - d_{i,j-\frac{1}{2}} \tag{39}$$

These terms have a second difference of the flow solution which is scaled to be present only in the presence of strong pressure gradients (i.e. shocks) and a fourth difference everywhere else. These terms are sufficient to stabilise the scheme and allow convergence.

4.5 Code building

For efficiency and good practice the code building process performed the compiling and linking stages separately using `gfortran` to avoid recompiling unaltered source code files during every update.

To speed up the code the optimisation flag `-O3` was used. As FORTRAN 77 is an old code many of the compiler warning options are turned off by default, so `-Wall` and `-fbounds-check` were used for the development.

This was all managed with a Makefile. All object files were stored in a separate directory to avoid clutter.

5 IMPLEMENTATION: THE ADJOINT SOLVER

Many approaches for creating discrete adjoint codes exist with different advantages and disadvantages. The approach taken for the adjoint solver of this project is adapted from the ADjoint method [8]:

1. Run CFD code to convergence to solve for state variable vector w
2. Find partial derivatives $\frac{\partial J}{\partial \alpha_i}$, $\frac{\partial J}{\partial \mathbf{w}}$, $\frac{\partial \mathbf{R}}{\partial \alpha_i}$ and $\frac{\partial \mathbf{R}}{\partial \mathbf{w}}$
3. Solve adjoint equation for vector $\mathbf{v}$
4. Form total derivative to get the sensitivity

This method was chosen because it has the best trade-off of simplicity and good results.

5.1 Partial derivatives

There are several different ways of calculating the required partial derivatives: finite differences, analytically or by automatic differentiation. It was decided to use AD due to the exact derivatives yielded as a result of an easy process.

If AD is immediately applied to unmodified code it may not produce the correct results. For example, consider a computer program where pressure (calculated from the flow solution vector $\mathbf{w}$) is integrated over a surface defined by design variables α_i.

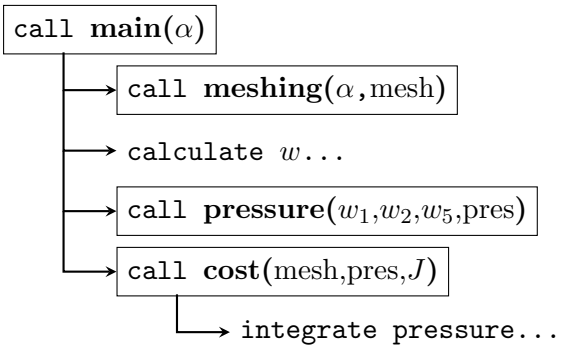

Figure 4: Function call graph displaying inheritance

Figure 4 shows a computer program where $\mathbf{w}$ inherits α_i. Many codes may be structured like this because information can be re-used. However if an AD engine is requested to find a variation $\frac{\partial}{\partial \mathbf{w}}$ at the same time as $\frac{\partial}{\partial \alpha_i}$ it can fail as it considers $\mathbf{w}$ to be a dependent variable. A value of $\frac{\partial}{\partial \alpha_i}$ would not be a true partial derivative and instead a total derivative.

As an alternative, Figure 5 shows a program that does exactly the same calculation but where $\mathbf{w}$ is isolated from α_i. This may be a less efficient process but now both $\frac{\partial}{\partial \mathbf{w}}$ and $\frac{\partial}{\partial \alpha_i}$ can be requested from the AD software at the same time with no conflict.

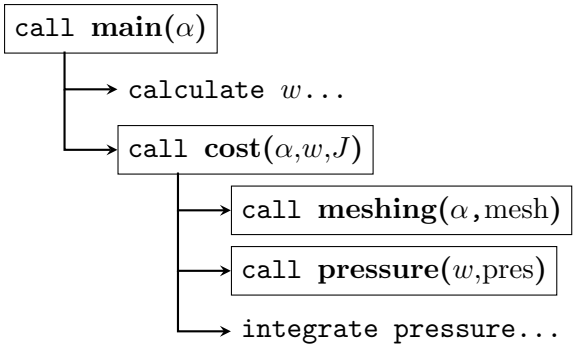

Figure 5: Function call graph displaying isolation

So there are two different approaches to designing functions, both with different advantages. For the primal code functions with inheritance were designed in order to be efficient while in loops. These kept their regular names `*.f`. For the adjoint solver functions with isolated variables were designed to be prepared for automatic differentiation, labelled `a*.f` to indicate they were destined for the adjoint solver. During development, these were compared to ensure correct values.

Rather than upload many files to Tapenade for the function dependencies they were all concatenated into a single file called `aM*.f` to denote merging. After differentiation these were called `aM*_d.f` in the case of forward differentiation. Careful management of these files was needed as there are a lot of redundant intermediate files not included in the final code.

This process needs to be done to both the cost function and residual function in order to yield $\frac{\partial J}{\partial \alpha_i}$, $\frac{\partial J}{\partial \mathbf{w}}$ and $\frac{\partial \mathbf{R}}{\partial \alpha_i}$, $\frac{\partial \mathbf{R}}{\partial \mathbf{w}}$ respectively. Possible examples for the cost function are shown in Figures 4 and 5 however the residual function is more complicated. For this boundary conditions and the meshing function must be successfully integrated.

It is also worth noting that the adjoint equations require the flow and residual variables to be the storage vectors of each state at each point as shown in Equations 40 and 41 rather than the individual conservative variables as seen in Equation 33. This of course means the flux Jacobian $\frac{\partial \mathbf{R}}{\partial \mathbf{w}}$ is a large square matrix which happens to be sparse, making the linear system solve very expensive.

$$\mathbf{w} = [\rho_1, \rho u_1, \rho v_1, \rho w_1, \rho E_1, ..., \rho_n, \rho u_n, \rho v_n, \rho w_n, \rho E_n] \tag{40}$$

$$\mathbf{R} = [R_1^{(1)}, R_1^{(2)}, R_1^{(3)}, R_1^{(4)}, R_1^{(5)}, ..., R_n^{(1)}, R_n^{(2)}, R_n^{(3)}, R_n^{(4)}, R_n^{(5)}] \tag{41}$$

This forms the most difficult part of the adjoint calculations. It is assumed that because the state variable vector $\mathbf{w}$ is sufficiently converged and the AD code returns close enough to an exact derivative that the required partial derivatives are also sufficiently close to the real values. This is different from creating iterative schemes as discussed in papers such as [10].

For this project all of these functions were differentiated in forward mode for the sake of simplicity, however the reverse mode of AD would almost certainly be quicker for $\frac{\partial J}{\partial \alpha_i}$ and $\frac{\partial J}{\partial \mathbf{w}}$ due to the large ratio of input variables to output

variables. The remaining two partial derivatives are more complex. The dependant variables here are the residual for every conservative variable in every cell, indicating a preference for forward mode. In this case the residual routine is very simple but this is not always the situation as there can be dissipative and viscous fluxes. The flux Jacobian is especially difficult due to its large size for what is a very sparse matrix. In fact, some iterative schemes try to avoid calculation of this in the first place. Mader et al. [8] deal with the flux Jacobian in great detail, using reverse mode AD in each cell to calculate it more efficiently than brute-force forward mode.

5.2 Forming the sensitivity

For practical use in the primal solver the flow variables are manipulated in two-dimensional arrays. However, the state variable vector stores these in a one-dimensional array. The halo cells for all of the conservative variables are stored first, then all of the main body cells and finally all of the 'corner cells'. As the halo cells are overwritten in each iteration with the boundary conditions, only the body cells should be used in the adjoint calculations. After the AD functions return the partial differentials, all of the rows and columns containing anything other than body cells are deleted.

By this point all of the relevant partial derivatives have been obtained and can be applied to Equations 8 and 9. This can be done using the iterative methods discussed in subsection 2.2, but there are several alternatives using external code.

For this project the linear system was solved using direct factorisation to attempt and immediate solution. This was done using LAPACK [55], enabled in the Makefile by adding the `-llapack` flag. This should be available from most Linux repositories and will need to be downloaded before building. The linear system is solved via LU decomposition, which has a cost of order $\mathcal{O}(n^2)$ given a flux Jacobian of size n. This however became prohibitively expensive for larger mesh densities and can solve with errors for badly conditioned matrices.

A much better alternative is to use a Krylov subspace method which is specifically dedicated to solving linear systems governed by sparse matrices. The generalised minimal residual method (GMRES) attempts to minimise a residual in the form $|A\mathbf{x} - \mathbf{b}|$ to solve for vector $\mathbf{x}$. This can also be referred to as the adjoint residual, similar to the pseudo-timestepping scheme. The Portable, Extensible Toolkit for Scientific Computation [56] (PETSc) is a freely available set of routines written in C, C++ and FORTRAN. Among many things this contains a function performing the GMRES method.

5.3 Validation of code

Despite the great complexity of the adjoint development the result is easy to validate. Finite differences will aid debugging and check the correct aswer up to a few digits. For checking validity for up to machine precision, an approach such as the CSDA will be needed as finite differences suffer from rounding errors for small step sizes.

A more difficult task is checking the intermediate stages, which is useful for debugging during development. While the final sensitivity is easy to check it may be difficult to validate the individual partial derivatives as the differentiated codes have complex structures and the resultant arrays are large. This is one of the greatest and weakest strengths of AD: while it may be difficult to solve any problems thrown up by the AD process, the automatic nature prevents any human errors from occurring in the first place. This is why it is advantageous to be able to prepare code that works well with AD the first time.

Due to the large size of the flux Jacobian the linear system can sometimes be difficult to solve, so checking that $L^T\mathbf{v} - \mathbf{g} = \mathbf{0}$ would be advantageous.

Another possible approach to solving intermediate errors is to have an already working adjoint code. While the approach described here may not be the most efficient, it can be a checking tool for the development of an adjoint solver that is more advanced.

6 RESULTS & DISCUSSION

The developed code was executed on a computer with an Intel i7-7500U CPU at 3.5 GHz with 16000 MiB memory, running on a `5.0.4-arch1-1-ARCH` Linux kernel release.

Unless otherwise stated the CFD code was run with the parameters shown in Table 1. This contains values needed to induce a shock in the flow in order to demonstrate transonic effects. The JST parameters were used to stabilise the flow. The need for a low CFL number is unknown but still allows the solution to converge at a reasonable rate.

Table 1: Standard flow parameters

Parameter	Symbol	Value
Inflow Mach number	M_{in}	0.8
Pressure ratio (in/out)	P_{rat}	0.3
Ratio of specific heats	γ	1.4
Throat ratio	A_{in}/A^*	2
JST parameter	$\kappa^{(2)}$	1/16
JST parameter	$\kappa^{(4)}$	1/32
CFL number	σ	0.3
Residual tolerance	R_{tol}	10^{-8}

6.1 Primal results

After the solution produces a suitably converged solution vector $\mathbf{w}$ of conservative variables this was converted into primal variables for presentation. Figures 6 and 7 show examples of this for the density and velocities.

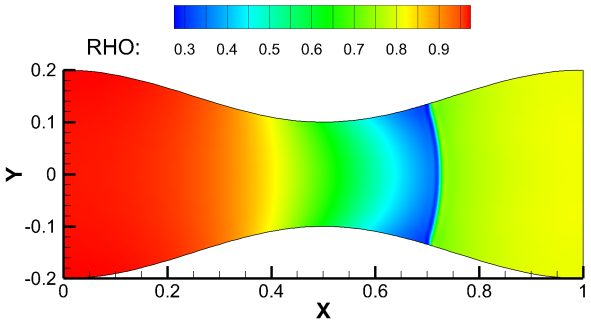

Figure 6: Density distribution through the duct

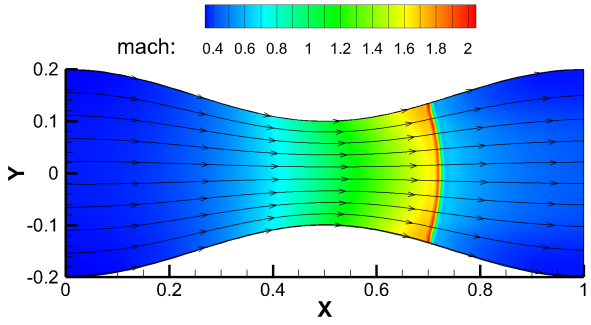

Figure 7: Mach number distribution through the duct with streamtraces

While the value of the output variables are the important results it is useful to know how accurate these are. The escape conditions are set to when the root mean square average of the results reach the residual tolerance but this is only an average; while some cells are converged more than this some are converged much less.

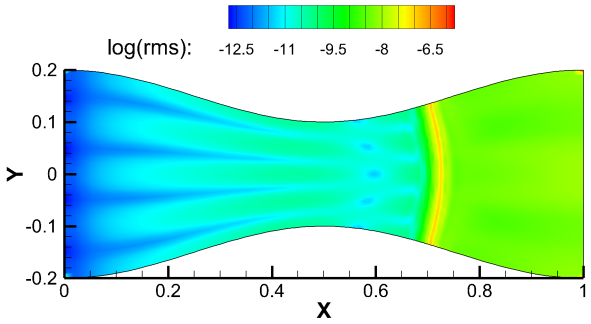

Figure 8: Distribution of RMS error through the duct

Figure 8 shows the local RMS error throughout the duct. As expected smooth regions such as the inflow show low amounts of error but highly nonlinear phenomena such as the shock display high RMS error.

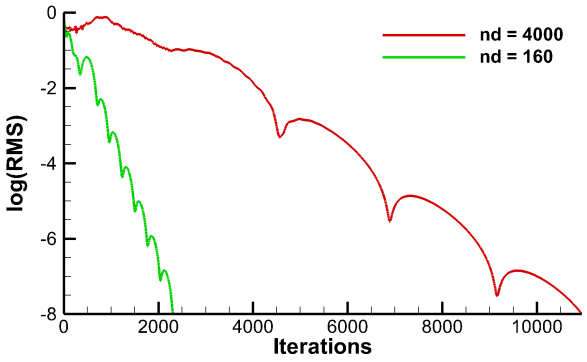

Figure 9: Convergence plot for different mesh densities: `nd` = 160 and `nd` = 4000

When executing the code different meshes have different convergence rates, as demonstrated in Figure 9. Mesh density `nd` is defined as the number of cells in the mesh, in this case the product of `x` and `y` cells as shown in Equation 42.

$$\texttt{nd} = \texttt{nx} \times \texttt{ny} \tag{42}$$

$$\texttt{nw} = 4 \times \texttt{nx} \times \texttt{ny} \tag{43}$$

The number of elements `nw` in solution vector **w** is four times this due to all of the conservative variables stored in each cell, shown in Equation 43.

6.2 Adjoint results

For the adjoint results a cost function J was introduced. In order to have relevance to common aerodynamics studies this was defined as a pressure integral evaluated over the wall regions $\mathcal{W}$ as shown in Equation 44. While this is not equivalent to lift or drag due to no vertical or horizontal component being taken it shares many similarities in terms of the adjoint equations.

$$J = \int_{\mathcal{W}} p(s) \cdot ds \tag{44}$$

Finally some design variables α_i are required. These are defined as the y-position of the spline control points as shown in Figure 3, which control the height and therefore the shape of the duct.

In general the adjoint code produced is effective at evaluating sensitivities however the linear solver used produces errors for large matrix sizes. As every other part of the code works as intended the only remaining step for future work is to replace the current linear solver with one of the suggested alternatives as laid out in subsection 5.2, after which the results should match to several decimal places. Due to the scope of this project it is not worth doing these so the following results are all affected. To mitigate this, all adjoint results will be produced using a mesh density of `nd` = 60 unless otherwise stated.

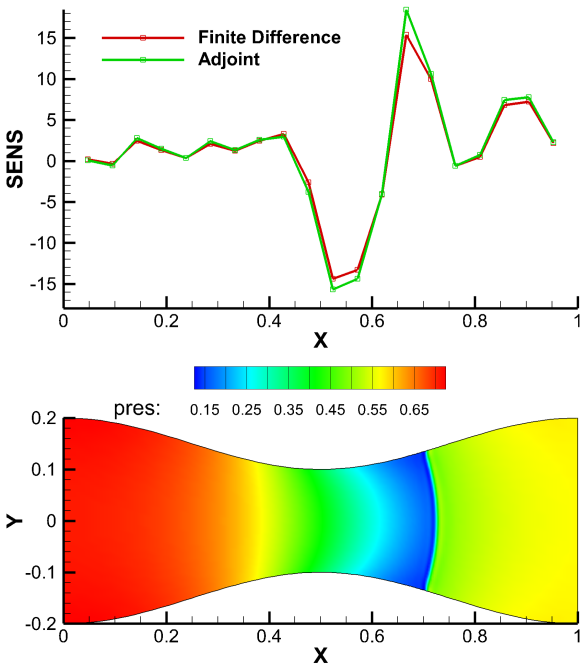

Figure 10: Sensitivity $\frac{dJ}{d\alpha_i}$ of the cost function J with respect to the control points at their respective x-positions. This is shown above a plot of the pressure distribution through the duct

A comparison of the sensitivity $\boldsymbol{\mathcal{G}}$ evaluated via both finite differences and the adjoint method is shown in Figure 10. This is for 20 design variables. As the cost function is a pressure integral the sensitivities are shown alongside the pressure distribution through the duct. As expected the greatest changes in J are at the throat and shock. A table of the numerical values shown in Figure 10 can be found in Appendix B for greater detail.

One of the most important aspects of the adjoint method is computational cost. Unless otherwise stated *cost* shall refer to the temporal cost, measured in seconds. This was determined using the `call CPU_TIME(t)` subroutine in FORTRAN. While the linear solver will be replaced, changing the following time results, the replacement methods are likely to be much faster based off adjoint literature.

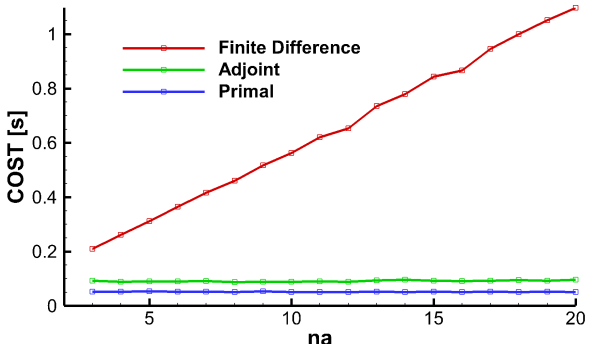

Figure 11: Computational cost in seconds for a variation of number of design variables `na`

The most widely known property of the adjoint method is the independence of the cost from the number of design variables. Figure 11 shows the computational cost, or elapsed time for the code to calculate the sensitivity for `na` design variables. This figure confirms that the time it takes to calculate the sensitivity via finite difference increases linearly with the number of design variables whereas the adjoint method has an independent cost. Note that the adjoint method includes a full primal run before the sensitivity is calculated.

$$\frac{\partial \mathbf{R}}{\partial \mathbf{w}} \in \mathbb{R}^{\mathtt{nw}\times\mathtt{nw}} \tag{45}$$

While the independence of cost from the number of design variables is extremely useful, there is a drawback. Solving the linear system is very expensive with an increase in the mesh density `nd` as the flux Jacobian has $16 \times \mathtt{nd}^2$ elements as shown in Equation 45.

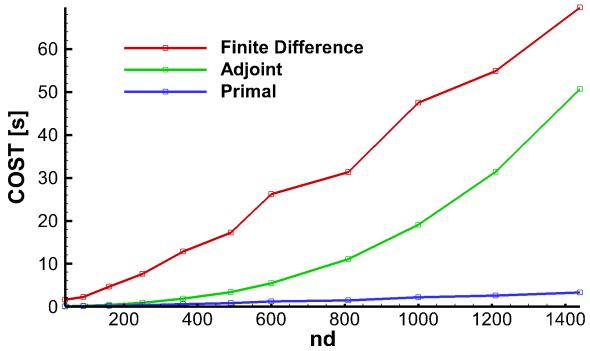

Figure 12: Computational cost in seconds for a variation in mesh density `nd` for `na = 20`

Figure 12 shows the computational cost required to calculate the sensitivity for a given mesh density. The finite difference method costs `na` times more than the primal. This is seen to have an approximately linear trend. However the adjoint method has a quadratic cost with respect to the mesh density, mostly because of the linear solver. A different linear solver would likely have a much cheaper relationship than this but would still become more expensive at a faster rate than the method of finite differences.

Table 2: Time breakdown for the adjoint method for different mesh densities

`na`	20	20
`nd`	250	2250
Primal run	0.604370 s	18.14444 s
Adjoint run	0.945372 s	179.4507 s
Runtime increase	$\times$1.6	$\times$9.9
Calculation of $\frac{\partial J}{\partial \alpha_i}$	.0636 %	.0031 %
Calculation of $\frac{\partial J}{\partial \mathbf{w}}$	10.4942 %	1.3832 %
Calculation of $\frac{\partial \mathbf{R}}{\partial \alpha_i}$	.3867 %	.0173 %
Calculation of $\frac{\partial \mathbf{R}}{\partial \mathbf{w}}$	67.0492 %	11.6530 %
System solution	22.8650 %	86.9432 %
Final sensitivity	.0047 %	.0001 %

These plots show the overall time that the adjoint method takes to run compared to other parts of the code, but a cost breakdown of the individual sections of the adjoint solver can be seen in Table 2. The solution of the linear system is by far the most expensive part. This takes up 23% of the adjoint solver for a mesh density of $\mathtt{nd} = 250$, but 87% for a higher mesh density of $\mathtt{nd} = 2250$. The calculation of the partial derivatives take up varying amounts of time. The flux Jacobian is generally the second most expensive part of the adjoint solver behind the linear system solution due to its large size. All of the partial derivatives were calculated using forward mode AD, so despite having more elements ($\mathtt{nw} \times \mathtt{na}$), $\frac{\partial \mathbf{R}}{\partial \alpha_i}$ is far quicker than $\frac{\partial J}{\partial \mathbf{w}}$ with $(1 \times \mathtt{nw})$ elements to calculate as it has a large amount of outputs rather than inputs which befits the forward mode. Of course, the converse would have been true of reverse mode had it been used. If these had been cal-

culated with finite differences then this would have taken much longer than the AD subroutines.

It has been shown that the most expensive operations in the adjoint metehod all deal with the flux Jacobian, so it is worth investigating the structure of this matrix. Figure 13 displays the internal structure of the matrix with all non-zero elements in black. This image was made by calling a library written in C [57] from the FORTRAN code.

This is a large, sparse matrix. For a mesh density of `nd` $= 60$ this is a matrix of order 240 and has a sparsity of 0.900, where the sparsity of a matrix is defined as the ratio of elements equal to zero over the total number of elements in the matrix. For a mesh density of `nd` $= 1000$, the sparsity increases to 0.993.

The matrix is also badly conditioned, with the absolute values of matrix extries usually ranging from the order of 10^2 on the lead diagonal to the order of 10^{-20} on the thick 'fuzzy' diagonal lines. The bad conditioning combined with the sparsity makes this difficult to solver numerically and must have a specialised solver. As the results show, direct LU decomposition is not sufficient.

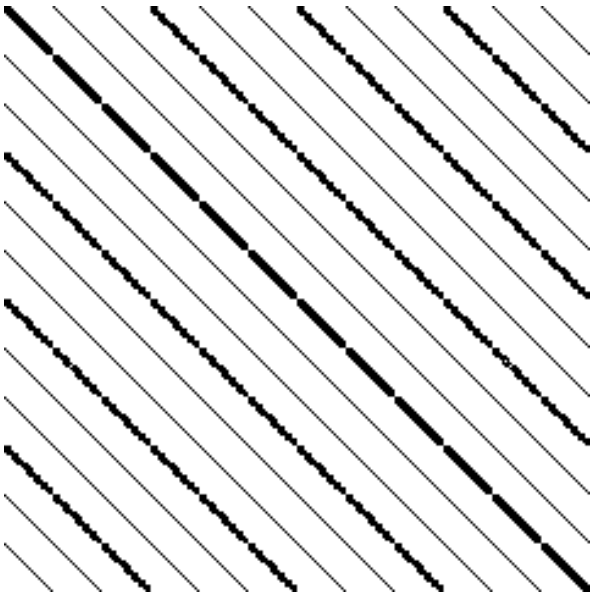

Figure 13: Visualisation of the flux Jacobian $\frac{\partial \mathbf{R}}{\partial \mathbf{w}}$ for the standard mesh density. All non-zero-elements are shown in black

Some of the other adjoint partial derivatives have structures that can be explained. For example, each element in the $\frac{\partial J}{\partial \mathbf{w}}$ term corresponds to a specific mesh cell for each conservative variable. For this term, every element in the array is equal to zero except at the boundary cells where the cost integration is performed. By contrast, the flux Jacobian is independent of the cost function as it is a linearisation of the Euler equations.

Many of the techniques discussed in the report are known to have high memory requirements. Because of this it is important to be able to predict the amount of memory needed by the code.

Many of the Linux system files are ASCII plaintext making it very easy to read status information. Once the process ID of the adjoint solver has been found using `pid = GETPID()`, another program can be written to read the status file `/proc/[pid]/status`. There are several different ways of measuring memory usage. Here the resident set size (also written as RSS or VmRSS) is used which is measured in kilobytes. This represents the non-swapped physical memory being used by a process.

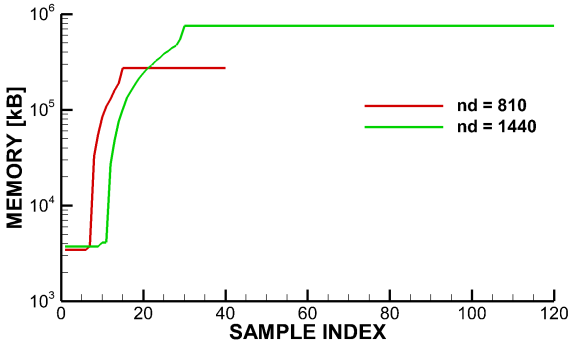

Figure 14: Program memory usage for different mesh densities

A computer program was written to measure the memory usage of the adjoint solver. This read the RSS value from the relevant status file at regular intervals. The results are shown in Figure 14. The small usage at the beginning is the primal flow solve. The section that rises is where the partial derivatives are calculated and the majority of time and memory usage is the matrix factorisation. By using a different linear system solver both of these things can be

significantly reduced.

7 CONCLUSIONS & FUTURE WORK

The objective of this paper was to make the creation of an adjoint solver a quick and easy process. Due to the successful evaluation of sensitivities as shown in the results section this can be said to have been achieved. It has been frequently stated in literature that AD can make the development of a discrete adjoint solver an almost automatic process the results presented in this project confirm this to be true.

An introductory account of adjoint equations was presented, moving from a practical derivation of the discrete adjoint equations to the more abstracted continuous equations and their mathematical properties. An overview of AD was then presented to highlight the many advantages it has when developing an adjoint solver.

The results extracted from the solver all display information as expected and sometimes provide valuable new insights. The exception to this is the performance of the linear solver. While only a simple method was required in accordance with the scope of this project the negative effects far exceeded those expected.

While this report has satisfied its objectives there are several areas which would benefit from more work. As has been mentioned the linear solver in use needs replacing with a dedicated Krylov solver, but this has been discussed in depth. In addition to this some of the partial derivatives currently computed via forward mode AD could be calculated much faster by use of the reverse mode. The physical process of differentiating the code could be improved as well. Currently this is done by uploading the files to an online web server, but Tapenade can be downloaded and integrated into a Makefile [48].

There are other avenues for improvement. While AD has been used as the primary method for an easy development of an adjoint solver, it has a limited scope for extension. Many CFD codes make calls to external 'black box' functions such as CAD packages for meshing. As the source code would not be available to the AD engine this could not be differentiated. While this is a fundamental problem with AD, a more fixable problem would be finding the Hessian, a metric used by many optimisation strategies. While repeated application of first difference AD would result in many duplicate calculations when finding the Hessian, a dedicated mode is being developed.

Finally, for the code to be truly novel research it would need to address an area of cutting edge research. As seen from the literature reviews this could be turbulence modelling, unsteady CFD, theoretical continuous adjoint equations or making the reverse mode of AD more efficient.

REFERENCES

[1] C. J. Anderson, "Discrete adjoint solver source code." https://github.com/charlie-j-white/discrete-adjoint-solver, 2019.

[2] A. Keane and P. Nair, *Computational approaches for aerospace design: the pursuit of excellence*. John Wiley & Sons, 2005.

[3] J. R. Martins, P. Sturdza, and J. J. Alonso, "The complex-step derivative approximation," *ACM Transactions on Mathematical Software (TOMS)*, vol. 29, no. 3, pp. 245–262, 2003.

[4] M. B. Giles and N. A. Pierce, "An introduction to the adjoint approach to design," *Flow, turbulence and combustion*, vol. 65, no. 3-4, pp. 393–415, 2000.

[5] M. Giles, N. Pierce, M. Giles, and N. Pierce, "Adjoint equations in cfd-duality, boundary conditions and solution behaviour," in *13th Computational Fluid Dynamics Conference*, p. 1850, 1997.

[6] S. Nadarajah and A. Jameson, "A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization," in *38th Aerospace Sciences Meeting and Exhibit*, p. 667, 2000.

[7] P. He, C. A. Mader, J. R. Martins, and K. J. Maki, "An aerodynamic design op-

timization framework using a discrete adjoint approach with openfoam," *Computers & Fluids*, vol. 168, pp. 285–303, 2018.

[8] C. A. Mader, J. RA Martins, J. J. Alonso, and E. V. Der Weide, "Adjoint: An approach for the rapid development of discrete adjoint solvers," *AIAA journal*, vol. 46, no. 4, pp. 863–873, 2008.

[9] J. Elliott and J. Peraire, "Aerodynamic design using unstructured meshes," in *Fluid Dynamics Conference*, p. 1941, 1996.

[10] M. B. Giles, M. C. Duta, J.-D. Muacute, and N. A. Pierce, "Algorithm developments for discrete adjoint methods," *AIAA journal*, vol. 41, no. 2, pp. 198–205, 2003.

[11] M. B. Giles and N. A. Pierce, "On the properties of solutions of the adjoint euler equations," *Numerical Methods for Fluid Dynamics VI. ICFD*, pp. 1–16, 1998.

[12] F. Alauzet and O. Pironneau, "Continuous and discrete adjoints to the euler equations for fluids," *International Journal for Numerical Methods in Fluids*, vol. 70, no. 2, pp. 135–157, 2012.

[13] J. C. Newman III, A. C. Taylor III, R. W. Barnwell, P. A. Newman, and G. J.-W. Hou, "Overview of sensitivity analysis and shape optimization for complex aerodynamic configurations," *Journal of Aircraft*, vol. 36, no. 1, pp. 87–96, 1999.

[14] O. Pironneau, "On optimum design in fluid mechanics," *Journal of Fluid Mechanics*, vol. 64, no. 1, pp. 97–110, 1974.

[15] A. Jameson, "Aerodynamic design via control theory," *Journal of scientific computing*, vol. 3, no. 3, pp. 233–260, 1988.

[16] J. L. Lions, *Optimal Control of Systems Governed by Partial Differential Equations (Grundlehren der Mathematischen Wissenschaften)*, vol. 170. Springer Berlin, 1971.

[17] O. Baysal and M. E. Eleshaky, "Aerodynamic design optimization using sensitivity analysis and computational fluid dynamics," *AIAA journal*, vol. 30, no. 3, pp. 718–725, 1992.

[18] M. Eleshaky and O. Baysal, "Design of 3-d nacelle near flat-plate wing using multiblock sensitivity analysis (ados)," in *32nd Aerospace Sciences Meeting and Exhibit*, p. 160, 1994.

[19] A. Jameson and J. Reuther, "Control theory based airfoil design using the euler equations," in *5th Symposium on Multidisciplinary Analysis and Optimization*, p. 4272, 1994.

[20] J. Reuther and A. Jameson, "Supersonic wing and wing-body shape optimization using an adjoint formulation," 1995.

[21] J. Reuther, A. Jameson, J. Farmer, L. Martinelli, and D. Saunders, "Aerodynamic shape optimization of complex aircraft configurations via an adjoint formulation," in *34th Aerospace Sciences Meeting and Exhibit*, p. 94, 1996.

[22] J. J. Reuther, A. Jameson, J. J. Alonso, M. J. Rimlinger, and D. Saunders, "Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers, part 1," *Journal of aircraft*, vol. 36, no. 1, pp. 51–60, 1999.

[23] J. J. Reuther, A. Jameson, J. J. Alonso, M. J. Rimlinger, and D. Saunders, "Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers, part 2," *Journal of aircraft*, vol. 36, no. 1, pp. 61–74, 1999.

[24] M. Nemec, D. W. Zingg, and T. H. Pulliam, "Multipoint and multi-objective aerodynamic shape optimization," *AIAA journal*, vol. 42, no. 6, pp. 1057–1065, 2004.

[25] J. R. RA Martins, J. J. Alonso, and J. J. Reuther, "High-fidelity aerostructural design optimization of a supersonic business jet," *Journal of Aircraft*, vol. 41, no. 3, pp. 523–530, 2004.

[26] J. Newman, III, I. A. Taylor, and G. Burgreen, "An unstructured grid approach to

sensitivity analysis and shape optimization using the euler equations," in *12th Computational Fluid Dynamics Conference*, p. 1646, 1995.

[27] J. Newman III and A. Taylor III, "Three-dimensional aerodynamic shape sensitivity analysis and design optimization using the euler equations on unstructured grids," in *14th Applied Aerodynamics Conference*, p. 2464, 1996.

[28] J. Newman, III, R. Barnwell, A. Taylor, III, J. Newman, III, R. Barnwell, and A. Taylor, III, "Aerodynamic shape sensitivity analysis and design optimization of complex configurations using unstructured grids," in *15th Applied Aerodynamics Conference*, p. 2275, 1997.

[29] J. Elliott and J. Peraire, "Aerodynamic optimization on unstructured meshes with viscous effects," in *Computational Fluid Dynamics Review 1998: (In 2 Volumes)*, pp. 542–559, World Scientific, 1998.

[30] W. K. Anderson and D. L. Bonhaus, "Aerodynamic design on unstructured grids for turbulent flows," 1997.

[31] J. Driver and D. W. Zingg, "Numerical aerodynamic optimization incorporating laminar-turbulent transition prediction," *AIAA journal*, vol. 45, no. 8, pp. 1810–1818, 2007.

[32] Z. Lyu, G. K. Kenway, C. Paige, and J. Martins, "Automatic differentiation adjoint of the reynolds-averaged navier-stokes equations with a turbulence model," in *21st AIAA Computational Fluid Dynamics Conference*, p. 2581, 2013.

[33] J. A. Krakos, *Unsteady adjoint analysis for output sensitivity and mesh adaptation.* PhD thesis, Massachusetts Institute of Technology, 2012.

[34] T. D. Economon, F. Palacios, and J. J. Alonso, "Unsteady continuous adjoint approach for aerodynamic design on dynamic meshes," *AIAA Journal*, vol. 53, no. 9, pp. 2437–2453, 2015.

[35] J. P. Thomas and E. H. Dowell, "Discrete adjoint approach for nonlinear unsteady aeroelastic design optimization," *AIAA Journal*, pp. 1–9, 2019.

[36] R. E. Wengert, "A simple automatic derivative evaluation program," *Communications of the ACM*, vol. 7, no. 8, pp. 463–464, 1964.

[37] B. Speelpenning, "Compiling fast partial derivatives of functions given by algorithms," tech. rep., Illinois Univ., Urbana (USA). Dept. of Computer Science, 1980.

[38] L. B. Rall, "Differentiation and generation of taylor coefficients in pascal-sc," in *A New Approach to Scientific Computation*, pp. 291–309, Elsevier, 1983.

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

[40] A. Griewank, "A mathematical view of automatic differentiation," *Acta Numerica*, vol. 12, pp. 321–398, 2003.

[41] L. Hascoët and V. Pascual, "The Tapenade Automatic Differentiation tool: Principles, Model, and Specification," *ACM Transactions On Mathematical Software*, vol. 39, no. 3, 2013.

[42] A. Verma, "Structured automatic differentiation," tech. rep., Cornell University, 1998.

[43] P. H. W. Hoffmann, "A hitchhiker's guide to automatic differentiation," *Numerical Algorithms*, vol. 72, pp. 775–811, Jul 2016.

[44] M. Fagan and A. Carle, "Reducing reverse-mode memory requirements by using profile-driven checkpointing," *Future Generation Computer Systems*, vol. 21, no. 8, pp. 1380–1390, 2005.

[45] L. L. Sherman, A. C. Taylor III, L. L. Green, P. A. Newman, G. W. Hou, and V. M. Korivi, "First-and second-order aerodynamic sensitivity derivatives via automatic differentiation with incremental iterative methods," *Journal of Computational Physics*, vol. 129, no. 2, pp. 307–331, 1996.

[46] B. Mohammadi, "Optimal shape design, reverse mode of automatic differentiation and turbulence," in *35th Aerospace Sciences Meeting and Exhibit*, p. 99, 1997.

[47] J.-D. Müller and P. Cusdin, "On the performance of discrete adjoint cfd codes using automatic differentiation," *International journal for numerical methods in fluids*, vol. 47, no. 8-9, pp. 939–945, 2005.

[48] D. Jones, J.-D. Müller, and F. Christakopoulos, "Preparation and assembly of discrete adjoint cfd codes," *Computers & Fluids*, vol. 46, no. 1, pp. 282–286, 2011.

[49] F. Christakopoulos, D. Jones, and J.-D. Müller, "Pseudo-timestepping and verification for automatic differentiation derived cfd codes," *Computers & Fluids*, vol. 46, no. 1, pp. 174–179, 2011.

[50] S. Xu, D. Radford, M. Meyer, and J.-D. Müller, "Stabilisation of discrete steady adjoint solvers," *Journal of Computational Physics*, vol. 299, pp. 175–195, 2015.

[51] J. Hückelheim, P. Hovland, M. M. Strout, and J.-D. Müller, "Reverse-mode algorithmic differentiation of an openmp-parallel compressible flow solver," *The International Journal of High Performance Computing Applications*, vol. 33, no. 1, pp. 140–154, 2019.

[52] J. C. Hückelheim and J.-D. Müller, "Checkpointing with time gaps for unsteady adjoint cfd," in *Advances in Evolutionary and Deterministic Methods for Design, Optimization and Control in Engineering and Sciences*, pp. 117–130, Springer, 2019.

[53] C. A. Mader and J. R. RA Martins, "Computation of aircraft stability derivatives using an automatic differentiation adjoint approach," *AIAA journal*, vol. 49, no. 12, pp. 2737–2750, 2011.

[54] A. Jameson, W. Schmidt, and E. Turkel, "Numerical solution of the euler equations by finite volume methods using runge kutta time stepping schemes," in *14th fluid and plasma dynamics conference*, p. 1259, 1981.

[55] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[56] S. Abhyankar, J. Brown, E. M. Constantinescu, D. Ghosh, B. F. Smith, and H. Zhang, "Petsc/ts: A modern scalable ode/dae solver library," *arXiv preprint arXiv:1806.01437*, 2018.

[57] lvandeve, "Lodepng." https://github.com/lvandeve/lodepng, 2019.

A NOMENCLATURE & GLOSSARY

As this report contains many figures and equations a centralised list of the recurring terms is presented here. This was placed in an appendix as all of the terms have been defined within the body text.

Table 3: List of technical terms used in this report

Symbol	Meaning
$\boldsymbol{\alpha}$	Vector of design variables
α_i	Component of the design variable vector
$\boldsymbol{\lambda}$	Lagrange multiplier
$\boldsymbol{\nu}$	Adjoint flow solution
ϕ	Intermediate elemental function
$\omega()$	Temporal complexity operator
$\mathbf{c}$	Inequality constraints
$\mathbf{c}_{eq}$	Equality constraints
$\mathbf{f}_i$	Source term
$\mathbf{F}$	Test vector of outputs
$\mathbf{g}$	Source term
J	Objective function
L	Flux Jacobian
m	Number of outputs
n	Number of inputs
$\mathbf{R}$	Residual
$S()$	Spatial complexity operator
$\mathbf{u}_i$	Counterpart to adjoint vector
$\mathbf{v}$	Adjoint vector
$\mathbf{w}$	Flow solution / state variable vector
$\mathbf{x}$	Test vector of inputs
$\boldsymbol{\mathcal{G}}$	Sensitivity
$\mathcal{J}$	Jacobian
$\boldsymbol{\mathcal{U}_i}$	Direction vector
$\boldsymbol{\mathcal{V}_j}$	Direction vector
`na`	Number of design variables
`nd`	Mesh density
`nw`	Storage vector size
$\Im[]$	Imaginary component operator
inheritance	The case when $\mathbf{F} = \mathbf{F}(\mathbf{x})$
isolation	The case when $\mathbf{F} \neq \mathbf{F}(\mathbf{x})$

B FURTHER ADJOINT RESULTS

Table 4: Full list of the sensitivities displayed in Figure 10

i	x	FD	Adjoint	% ERROR
1	0.04761905	0.17556207	0.08364249	109.895802
2	0.09523810	-0.34508147	-0.54942970	-37.192789
3	0.14285714	2.45640193	2.78782770	-11.888316
4	0.19047619	1.31177250	1.49082965	-12.010571
5	0.23809524	0.36758136	0.35579467	03.312779
6	0.28571429	2.06001568	2.41531922	-14.710418
7	0.33333333	1.25036148	1.33145968	-06.090924
8	0.38095238	2.46453985	2.55994116	-03.726699
9	0.42857143	3.28509347	2.95187168	11.288492
10	0.47619048	-2.62516514	-3.79131666	-30.758484
11	0.52380952	-14.36855704	-15.67111046	-08.311813
12	0.57142857	-13.31000723	-14.37579600	-07.413772
13	0.61904762	-4.10746061	-4.04381756	01.573836
14	0.66666667	15.39530706	18.45308498	-16.570551
15	0.71428571	9.98747439	10.63133381	-06.056243
16	0.76190476	-0.58904983	-0.59187320	-00.477023
17	0.80952381	0.47987536	0.72190838	-33.526834
18	0.85714286	6.81818534	7.44634066	-08.435759
19	0.90476190	7.20934603	7.76603407	-07.168241
20	0.95238095	2.16823062	2.29992037	-05.725839