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

Adjoint methods have the potential to greatly increase the speed of gradient based optimisation codes but are very complex to develop. This paper attempts to make the development easier through concise explanation of theory and relevant implementation comments. First a review of adjoint equations are presented followed by an overview of automatic differentiation, both including a literature review of related adjoint research techniques. The second half of the report details how to make a primal flow solver with an extension to an adjoint flow sensitivity solver. Finally results are presented with a discussion.

Keywords: Adjoint method, Automatic differentiation, CFD, Aerodynamic Optimisation

1 INTRODUCTION

With the high speed and low cost of today's computers it is easier than ever to run CFD codes to analyse a given design. Current research investigates how to best search a design space in order to find an optimum position. 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 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. **The objective of this paper is to leave the reader confident that they can not only create an adjoint solver, but understand much of the surrounding literature.** 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 6 and 7). 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, as well as be prepared for technical terms presented in any literature.

The final half of the theory will be presented in Section 3 which will cover automatic differentiation, or AD, used as a technique of finding the partial derivatives. Again, this is written so the concept can be understood by the reader quickly, but with more detail and further references for deeper understanding. This has another literature review detailing the use of AD in adjoint research.

The remaining sections of this paper will detail the practical development of a primal CFD code and its corresponding adjoint solver. Fitting in with this paper's theme these sections will attempt to give relevant insights on the process to help the reader overcome common problems as opposed to just an outline of the theory.

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 (LINK). This will compile and run on an up-to-date Linux system.

2 ADJOINT EQUATIONS IN SENSITIVITY ANALYSIS

Any optimisation strategy using a gradient based method requires the sensitivity of the cost function at that point with respect to the design variables. This is usually the gradient vector of partial derivatives. The most common method (see Keane [1] 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 1) but a second-order approximation requires two extra evaluations per design variable. For this work, consider a scalar cost function $J \in \mathbb{R}$ as a function of design variables $\alpha_i \in \mathbb{R}^n$.

$$\frac{\partial J}{\partial \alpha_i} = \frac{J(\alpha_i + \epsilon) - J(\alpha_i)}{\epsilon} + \mathcal{O}(\epsilon) \tag{1}$$

This can be improved upon by using the complex-step derivative approximation [2] as shown in Equation 2. By replacing every real variables declaration with a complex ones the CSDA can be used. 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.

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

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

Introduce state variable vector $\mathbf{w} \in \mathbb{R}^p$, where $\mathbf{w} = \mathbf{w}(\alpha)$. This is subject to the constraint that the fluid equations have been solved, represented by the residual $\mathbf{R} \in \mathbb{R}^p$ being equivalent to zero.

$$\begin{aligned} \text{cost function} \quad & J(\alpha_i, \mathbf{w}) \\ \text{subject to} \quad & \mathbf{R}(\alpha_i, \mathbf{w}) = \mathbf{0} \end{aligned} \tag{3}$$

To find the sensitivity of J with respect to design variables α, take the total derivatives $\frac{dJ}{d\alpha_i}$ and $\frac{d\mathbf{R}}{d\alpha_i}$, where the latter term can be shown to equal zero. To maintain consistency let $\mathbf{u}_i = \frac{d\mathbf{w}}{d\alpha_i}$, then the sensitivity can be written as:

$$\frac{dJ}{d\alpha_i} = \frac{\partial J}{\partial \alpha_i} + \frac{\partial J}{\partial \mathbf{w}}\mathbf{u}_i \tag{4}$$

$$\frac{\partial \mathbf{R}}{\partial \mathbf{w}}\mathbf{u}_i = -\frac{\partial \mathbf{R}}{\partial \alpha_i} \tag{5}$$

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, not all terms are equally easy to calculate: Equation 5 involves a very expensive system solve which must be done for each α_i. As the number of design variables is often greater than the number of cost function, this is inefficient. Now consider the **adjoint method**:

$$\frac{dJ}{d\alpha_i} = \frac{\partial J}{\partial \alpha_i} - \mathbf{v}^T\frac{\partial \mathbf{R}}{\partial \alpha_i} \tag{6}$$

$$\frac{\partial \mathbf{R}}{\partial \mathbf{w}}^T \mathbf{v} = \frac{\partial J}{\partial \mathbf{w}}^T \tag{7}$$

By introducing an adjoint vector $\mathbf{v}$ we can construct a new linear system (Equation 7) which is dual to the system in Equation 5. In this new adjoint equation there is no reference to the design variables, meaning finding the total sensitivity is a process almost independent of the number of design variables. The equivalence of the direct and adjoint methods can be proved by Equation 8:

$$-\mathbf{v}^T \frac{\partial \mathbf{R}}{\partial \alpha_i} = \mathbf{v}^T \frac{\partial \mathbf{R}}{\partial \mathbf{w}} \mathbf{u}_i = \frac{\partial J}{\partial \mathbf{w}} \mathbf{u}_i \tag{8}$$

This is the linear algebra approach, but the adjoint method can be interpreted in another way, through Lagrange multipliers.

$$\begin{aligned} \frac{dJ}{d\alpha_i} = \left(\frac{\partial J}{\partial \alpha_i} + \frac{\partial J}{\partial \mathbf{w}} \frac{\partial \mathbf{w}}{\partial \alpha_i} \right) \\ - \boldsymbol{\lambda}^T \left(\frac{\partial \mathbf{R}}{\partial \alpha_i} + \frac{\partial \mathbf{R}}{\partial \mathbf{w}} \frac{\partial \mathbf{w}}{\partial \alpha_i} \right) \end{aligned} \tag{9}$$

$$\frac{\partial J}{\partial \mathbf{w}} - \boldsymbol{\lambda}^T \frac{\partial \mathbf{R}}{\partial \mathbf{w}} = \mathbf{0} \tag{10}$$

The linear algebra approach is as favoured by Giles and Pierce [3] due to clear comparison of direct and adjoint systems, whereas the Lagrange multiplier formulation is preferred by Jameson [4] 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 [5] 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 [6], the method used in this project. Due to the importance of this, it has its own devoted discussion covered in Section 3.

Obesrvant 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 $\frac{\partial J}{\partial \alpha_i}$ the adjoint equations use the vector of total derivatives $\frac{dJ}{d\alpha_i}$, which itself has a term $\frac{\partial J}{\partial \alpha_i}$ with another non-zero component. The reason for 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 1 is exactly the same as the $\frac{dJ}{d\alpha_i}$ in Equation 6. So what is the difference between the $\frac{\partial J}{\partial \alpha_i}$ in Equation 1 and the $\frac{\partial J}{\partial \alpha_i}$ in Equation 6?

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 quasi-linear form

$$L\mathbf{w} = \mathbf{0} \tag{11}$$

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 5 and 7) can be written as

$$L\mathbf{u}_i = \mathbf{f}_i \tag{12}$$

$$L^T \mathbf{v} = \mathbf{g} \tag{13}$$

Writing these last three equations in this form allows us to apply the theory of linear systems (Equation 14) to what was before a complex PDE and gain an intuitive understanding of how they work.

$$\textit{dynamics} \times \textit{solution} = \textit{source} \tag{14}$$

The dynamics for all of these is 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 6 and 7. Equation 6 is a simple case of multiplying matrices and adding vectors, but the solution of the linear system in Equation 7 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 [3, 4, 7, 8] by driving down an *adjoint residual* and converging $\mathbf{v}$ to its true solution.

$$\frac{\partial \mathbf{u}_i}{\partial t} + L\mathbf{u}_i = \mathbf{f}_i \tag{15}$$

$$-\frac{\partial \mathbf{v}}{\partial t} + L^T\mathbf{v} = \mathbf{g} \tag{16}$$

Note that time is reversed in the adjoint formulation. While this is a common approach, simpler ways of solving Equation 7. As it is a matrix-vector linear system direct factorisation is possible but becomes prohibitively expensive as the mesh density (hence the flux Jacobian) increases. This can be addressed by using methods designed to solve sparse matrices, such as a Krylov solver.

2.3 Continuous adjoint equations

The discrete adjoint equations (DAEs) involve creating a dual system from equations that have already been discretised, but the continuous adjoint method (CAEs) involves 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 adjoint equations as a whole.

An example of creating a dual system (of the cost function, not the sensitivity) can be found in the paper by Giles and Pierce [9]. This is recreated here to highlight properties of adjoint systems.

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 the inner product over domain Ω and boundary $\partial\Omega$ respectively. This is shown in Equation 17 for the volume integral case and is labelled by the subscripts.

$$\langle \mathbf{V}, \mathbf{U} \rangle_\Omega = \int_\Omega \mathbf{V}^T \mathbf{U} dV \tag{17}$$

Consider Equation 18, the primal representation of the cost function. Unlike Equation 4, 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 $\mathbf{u}$. This uses boundary operators B and C 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.

$$\begin{aligned} \text{determine} \quad & J = \langle \mathbf{g}, \mathbf{w} \rangle_\Omega + \langle \mathbf{h}, C\mathbf{w} \rangle_{\partial\Omega} \\ \text{given that} \quad & L\mathbf{w} = \mathbf{f} \quad \text{in } \Omega \\ \text{and} \quad & B\mathbf{w} = \mathbf{e} \quad \text{on } \partial\Omega \end{aligned} \tag{18}$$

Now consider the dual system as shown in Equation 19, now with an introduction of the adjoint flow solution $\boldsymbol{\nu}$ and adjoint operators L^*, B^* and C^*.

$$\begin{aligned} \text{determine} \quad & J = \langle \boldsymbol{\nu}, \mathbf{f} \rangle_\Omega + \langle C^* \boldsymbol{\nu}, \mathbf{e} \rangle_{\partial\Omega} \\ \text{given that} \quad & L^* \boldsymbol{\nu} = \mathbf{g} \quad \text{in } \Omega \\ \text{and} \quad & B^* \boldsymbol{\nu} = \mathbf{h} \quad \text{on } \partial\Omega \end{aligned} \tag{19}$$

The equivalence of the two results can be proved using

$$\begin{aligned} &\langle L^* \boldsymbol{\nu}, \mathbf{w} \rangle_\Omega + \langle B^* \boldsymbol{\nu}, C\mathbf{w} \rangle_{\partial\Omega} \\ &\quad = \langle \boldsymbol{\nu}, L\mathbf{w} \rangle_\Omega + \langle C^* \boldsymbol{\nu}, B\mathbf{w} \rangle_{\partial\Omega} \end{aligned} \tag{20}$$

After this dual system has been created the equations can be linearised then discretised. A comparison of the discrete and adjoint systems is also performed by Nadarajah and Jameson [4].

Section to be completed later

2.4 Adjoint methods in research

Section to be completed later

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. [10].

A summary of pre-adjoint methods in design is in Hicks [11].

The first use in a fluid mechanics design problem was by Pironneau in 1974 [12], but the first application to aerodynamic design was by Jameson in 1988 [13]. These approaches used optimal control theory for systems governed by PDEs, as detailed by Lions [14], 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.

Due to the complexity of the governing equations, computational fluid dynamics calculations have always been computationally expensive. In the 1980s computers became fast at low prices, enough to be of use to researchers and the field of aerodynamic design, or optimisation, took off. Simple two-dimensional cases were handled without too much issue but implementing complex three-dimensional configurations was an issue. Using a structured mesh was an issue, often handled by multiblock or overlapping (sometimes known as CHIMERA) meshes, if not unstructured. To avoid using finite differences on these problems, researchers tried developing adjoint solvers to gain sensitivities for design.

Despite their complexity, structured meshes are often preferred for their superior flow capture. Several authors favour these:

Eleshaky and Baysal did work on a ‘quasi-analytical’ alternative to finite differences in 1992 [15] on a simplified scramject-afterbody configuration solving two-dimensional Euler equations. Later they did work on a full three-dimensional multiblock mesh in 1994 [16], 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 [17], 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. They also applied this method to a wing-body supersonic optimisation [18]. In 1996 the develop this method much further [19] and use a multiblock mesh to design complex aircraft configurations using the adjoint method. This was later extended to produce a study [20, 21] examining the effects of parallel computers.

In 2004, Nemec et al. [22] performed a multi-point and multi-objective aerodynamic shape optimisation using a discrete adjoint formulation. There is also work done by Martins et al. in this period. In 2004 a paper was released [23] examining aero-structural coupling. More of Martins' work shall be reviewed later.

Despite the lower quality solution capture, use of unstructured meshes in CFD is also of great importance due to the much greater flexibility in shapes. Based off of work from previously derived optimality conditions, Newman et al. [24] use the adjoint method on a two-dimensional unstructured grid. With other authors, Newman extended this approach to three dimensions in 1996 [25] and again in 1997 [26] 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 [7] for the inverse design problem on a multielement aerofoil. This work was expanded on [27] to include viscous effects.

Gaining adjoint sensitivities for Euler equations and even Navier-Stokes equations have become well established, but work on turbulence models is still ongoing. This is mainly because work on turbulence models themselves are not well understood, but there are other complications. Finding the derivative of code with many conditional statements can be demanding, but the question of feasibility is also present. Many adjoint formulations require the existence of a smooth solution, whereas turbulence has velocities on all time and length scales, making this more challenging.

In 1997 Anderson and Bonhaus differentiated a one-equation turbulence model [28]. Due to poor results using other methods, much of this was done by hand, resulting in tedious work but good results. In 2007 Driver and Zingg [29] also did this.

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.

There are two ways of automatically differentiating 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 overoloading 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 [30] with work on reverse mode published by Speelpenning in 1980 [31]. 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 [32]. 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 [33, 34].

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 [35], 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 $\mathbf{F}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$ 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 21. 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.

$$\mathcal{J} = \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \tag{21}$$

When measuring the efficiency of computers calculating the Jacobian there are two useful measures as defined by Verma [36]: 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 22. Even in the best case scenario of one input or output variable it is quite expensive.

$$\begin{aligned} \omega(\mathbf{F},\mathcal{J}) = &\ (n\cdot m+1)\cdot\omega(\mathbf{F}) \\ S(\mathbf{F},\mathcal{J}) = &\ S(\mathbf{F}) \end{aligned} \tag{22}$$

The most intuitive mode of AD is the forward mode, which calculates the product $\mathcal{J}\boldsymbol{\mathcal{U}}_i$ as shown in Equation 23, 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 24, proportionality constant α_1 ranges from one to three).

$$\begin{bmatrix} \dot{F}_1 \\ \dot{F}_2 \\ \dot{F}_3 \end{bmatrix} = \begin{bmatrix} ? & \frac{\partial F_1}{\partial x_2} & ? & ? \\ ? & \frac{\partial F_2}{\partial x_2} & ? & ? \\ ? & \frac{\partial F_3}{\partial x_2} & ? & ? \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{23}$$

$$\begin{aligned} \omega(\mathbf{F},\mathcal{J}) = &\ \alpha_1\cdot n\cdot\omega(\mathbf{F}) \\ S(\mathbf{F},\mathcal{J}) = &\ S(\mathbf{F}) \end{aligned} \tag{24}$$

This is also known as the tangent mode. For a parameterised curve $\mathbf{r}(t)\in\mathbb{R}\to\mathbb{R}^m$ 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 25, 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 26 (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 $\beta = \texttt{\#\_inter\_vars}$.

$$\begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \bar{x}_3 \\ \bar{x}_4 \end{bmatrix} = \begin{bmatrix} ? & ? & ? & ? \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \frac{\partial F_2}{\partial x_3} & \frac{\partial F_2}{\partial x_4} \\ ? & ? & ? & ? \end{bmatrix}^T \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{25}$$

$$\begin{aligned} \omega(\mathbf{F},\mathcal{J}) = &\ \alpha_2\cdot m\cdot\omega(\mathbf{F}) \\ S(\mathbf{F},\mathcal{J}) = &\ \beta\cdot S(\mathbf{F}) \end{aligned} \tag{26}$$

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.

$$\left\langle \mathcal{J}\boldsymbol{\mathcal{U}}_i, \boldsymbol{\mathcal{V}}_j \right\rangle = \left\langle \boldsymbol{\mathcal{U}}_i, \mathcal{J}^T\boldsymbol{\mathcal{V}}_j \right\rangle \tag{27}$$

It is also known as the adjoint mode as the inner product equivalence shown in Equation 27 is in the same form as Equations 8 and 20.

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 28 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 [35].

$$\mathbf{F} = \phi_p \circ \phi_{p-1} \circ \ldots \circ \phi_1 \tag{28}$$

By using the chain rule it is found that the differential of this function (Equation 29) is simply the product of the elemental Jacobians, which are matrices in the general case.

$$\mathcal{J} = \phi'_p \cdot \phi'_{p-1} \cdot \ldots \cdot \phi'_1 \tag{29}$$

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 [37]. In order to reduce cost, the product between $\boldsymbol{\mathcal{U}}_i$ and the Jacobian in Equation 29 is evaluated from right-to-left in order to do vector-matrix products rather than the more expensive matrix-matrix products.

$$\begin{tikzcd}
c \arrow[d] & & \boldsymbol{\mathcal{U}}_i \arrow[d] \\
\phi_1(c) \arrow[d] \arrow[rr, dashed, "\phi'_1"] & & \phi'_1 \cdot \boldsymbol{\mathcal{U}}_i \arrow[d] \\
\phi_2(\phi_1) \arrow[d] \arrow[rr, dashed, "\phi'_2"] & & \phi'_2 \cdot \phi'_1 \cdot \boldsymbol{\mathcal{U}}_i \arrow[d] \\
\dots \arrow[d] & & \dots \arrow[d] \\
\phi_p(\phi_{p-1}) \arrow[d] \arrow[rr, dashed, "\phi'_p"] & & \phi'_p \cdot \dots \cdot \boldsymbol{\mathcal{U}}_i \arrow[d] \\
\mathbf{F} & & \mathcal{J}\boldsymbol{\mathcal{U}}_i
\end{tikzcd}$$

Figure 1: Propagation of information in forward mode. Adapted from [37]

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, the practical implementation of an efficient 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 [35] of retrieving these values.

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.*

$$\begin{tikzcd}
c \arrow[d] & & \boldsymbol{\mathcal{V}}_j^T \mathcal{J} \\
\phi_1(c) \arrow[d] \arrow[rr, dashed, "\phi'_1"] & & \boldsymbol{\mathcal{V}}_j^T \cdot \dots \cdot \phi'_1 \arrow[u] \\
\dots \arrow[d] & & \dots \arrow[u] \\
\phi_{p-1}(\phi_{p-2}) \arrow[d] \arrow[rr, dashed, "\phi'_{p-1}"] & & \boldsymbol{\mathcal{V}}_j^T \cdot \phi'_p \cdot \phi'_{p-1} \arrow[u] \\
\phi_p(\phi_{p-1}) \arrow[d] \arrow[rr, dashed, "\phi'_p"] & & \boldsymbol{\mathcal{V}}_j^T \cdot \phi'_p \arrow[u] \\
\mathbf{F} & & \boldsymbol{\mathcal{V}}_j^T \arrow[u]
\end{tikzcd}$$

Figure 2: Propagation of information in reverse mode. Adapted from [37]

An example of this process can be found in Appendix A, where the forward and reverse modes are applied to a simple function. This is the explanation given by most introductions to AD, but does not work in the general case as is needed for most practical research.

3.3 AD in adjoint research

Section to be completed later

2008 Martins, ADjoint: An approach for the rapid development of discrete adjoint solvers [6]

2011 JD Mueller, Verification of AD derived CFD codes [38]

Conditional statements for AD

Reducing reverse-mode memory requirements with checkpointing [39]

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 dissipative terms, resulting in inviscid results with compressible effects. The source code is available online (LINK) for any intersted 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. [40]. 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 complex-step derivative approximation 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 30 and 31) are a first-order system of hyperbolic non-linear partial differential equations. Here they have been solved in the conservation formulation.

$$\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = 0 \tag{30}$$

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho E\end{bmatrix} + \frac{\partial}{\partial x}\begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ u(\rho E+p)\end{bmatrix} + \frac{\partial}{\partial y}\begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ v(\rho E+p)\end{bmatrix} = 0 \tag{31}$$

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 32.

$$p = (\gamma - 1)\big[\rho E - \frac{\rho}{2}(u^2+v^2)\big] \tag{32}$$

While solving the RANS equations would have yielded a better solution they would have been far harder to solve for and create a more difficult creation of adjoint code. The Euler equations still give a good flow solution of compressible effects including shocks and other transonic effects.

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. This was not difficult to do for the case of a duct.

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 specify 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.

$$\phi(x) = \frac{1}{h(x)} + e^{-(x-x_0)^2} \tag{33}$$

This defined the height of the mesh but there was also an attempt to improve the distribution of the x-position 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 33. This was then integrated into a cumulative distribution function between zero and one then seeded in order to recover the x position of points.

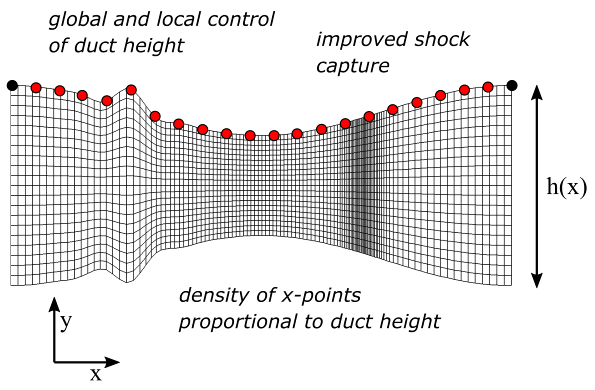
global and local control of duct height
improved shock capture
h(x)
density of x-points proportional to duct height
y
x

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

$$w_1 = w_{i+\frac{1}{2},j} = \frac{1}{2}(w_{i+1,j} + w_{i,j}) \tag{34}$$

However this scheme is implicitly unstable so requires some dissipative terms to stabilise the scheme, as seen in Equation 35. The cell volume is represented by Ω and the operator $Q()$ calculates the residual via the normal finite volume method in Equation 36. This integrates the fluxes around a cell with N faces for each equation of the Euler system.

$$\Omega\frac{\partial w}{\partial t} + Q(w) - D(w) = 0 \tag{35}$$

$$Q(w) = \sum_{k=1}^{N} \begin{bmatrix} F \\ G \\ 0 \end{bmatrix}_k \cdot \begin{bmatrix} \Delta y \\ -\Delta x \\ 0 \end{bmatrix}_k \tag{36}$$

The dissipative terms used are from the Jameson-Schmidt-Turkel (JST) paper [41]. 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 37 and requires terms to be calculated at each face.

$$D(w) = d_{i+\frac{1}{2},j} + d_{i,j+\frac{1}{2}} - d_{i-\frac{1}{2},j} - d_{i,j-\frac{1}{2}} \tag{37}$$

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 adjoint code produced for this project was developed with simplicity in mind rather than efficiency. Despite this the code produced adequate results.

Rather than use adjoint pseudo-timestepping (as discussed in subsection 2.2) to solve the adjoint equations as is traditionally done [8, 38] the approach as performed by Mader et al. [6] is used:

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 approach was taken as the process in this paper is very well explained, making it easy to emulate, and use of automatic differentiation is a simple but accurate process.

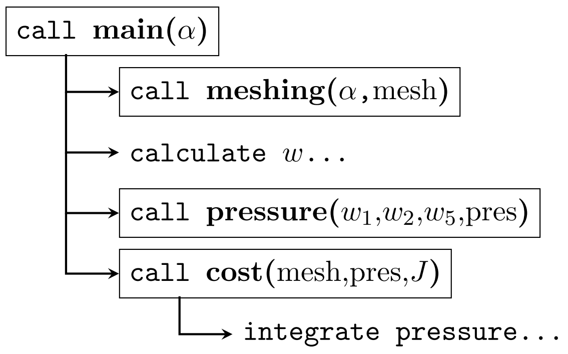
call **main**(α)
call **meshing**(α,mesh)
calculate w...
call **pressure**(w_1,w_2,w_5,pres)
call **cost**(mesh,pres,J)
integrate pressure...

Figure 4: Function call graph displaying inheritance

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 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.

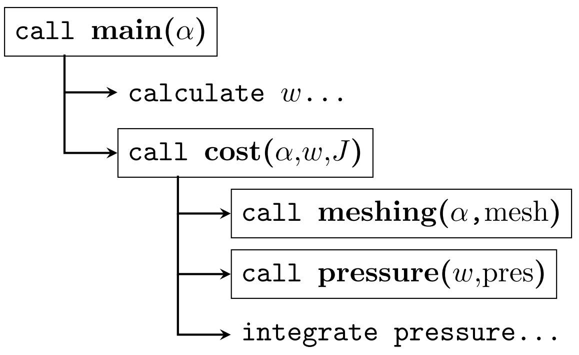
call **main**(α)
calculate w...
call **cost**(α,w,J)
call **meshing**(α,mesh)
call **pressure**(w,pres)
integrate pressure...

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 entire process can be summarised in Figure 6.

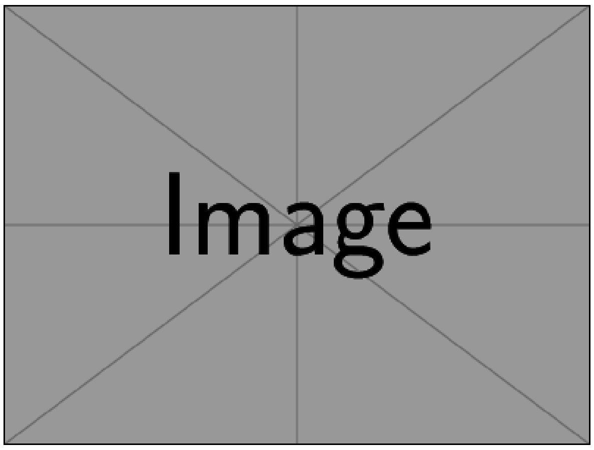

Figure 6: Flow chart of the differentiation process

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 38 and 39 rather than the individual conservative variables as seen in Equation 31. 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.

$$\begin{aligned}\mathbf{w} = [\rho_1, \rho u_1, \rho v_1, \rho w_1, \rho E_1, ..., \\ \rho_n, \rho u_n, \rho v_n, \rho w_n, \rho E_n]\end{aligned} \tag{38}$$

$$\begin{aligned}\mathbf{R} = [R_1^{(1)}, R_1^{(2)}, R_1^{(3)}, R_1^{(4)}, R_1^{(5)}, ..., \\ R_n^{(1)}, R_n^{(2)}, R_n^{(3)}, R_n^{(4)}, R_n^{(5)}]\end{aligned} \tag{39}$$

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 value. This is different from creating iterative schemes as discussed in papers such as [8].

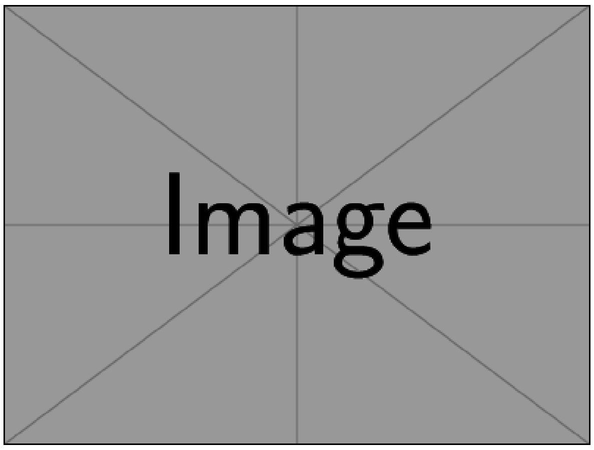

Figure 7: Movement from 2D cells to 1D storage arrays

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. [6] 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 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 6 and 7. This can be done using the methods discussed in subsection 2.2.

Initially the linear system was solved using direct factorisation to ensure a perfect solution. This was done using LAPACK, 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.

If neither of these techniques are to be used then it is possible to use Krylov subspace methods which are 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 reffered to as the adjoint residual, similar to the pseudo-timestepping scheme.

5.3 Validation

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.

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.

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 fun 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 CFL number is low for unexplained reasons but 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}

First the results of the primal code will be shown to show what the standard result looks like before the adjoint results will be displayed.

6.1 Primal results

After the solution produces a suitably converged solution vector of conservative variables $\mathbf{w}$ this was converted into primal variables for presentation. Figures 8 and 9 show examples of this for the density and velocities.

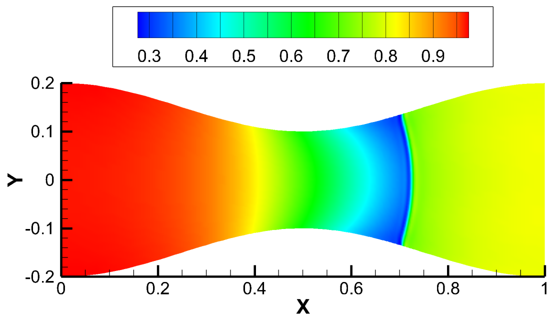

Figure 8: Density distribution through the duct

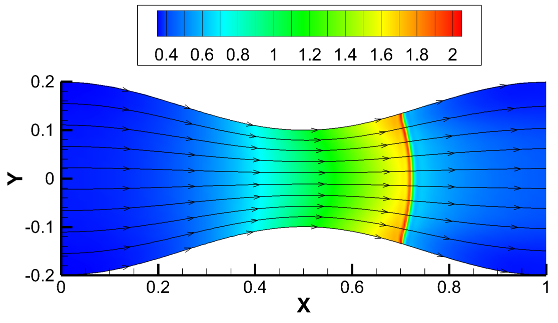

Figure 9: Mach number distribution through the duct with streamtraces

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

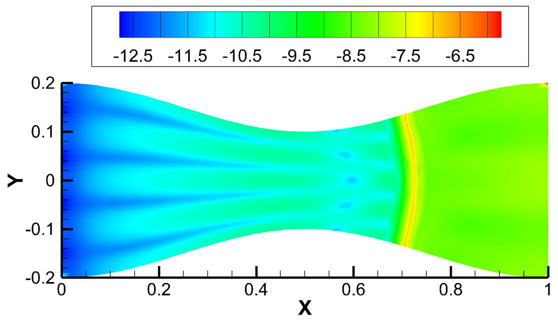

Figure 10: Distribution of RMS error through the duct

Figure 10 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.

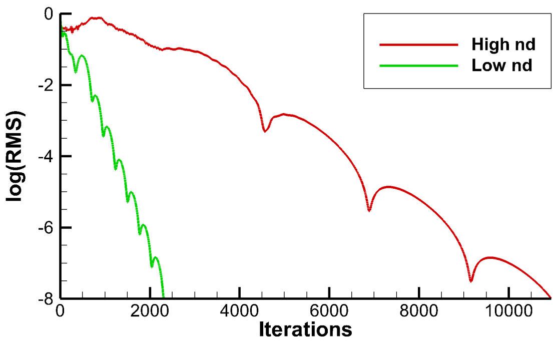

Figure 11: Convergence plot for different mesh densities: `nd` = 160 and `nd` = 4000

When executing the code different meshes have different convergence rates, as demonstrated in Figure 11. 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 40.

$$\texttt{nd} = \texttt{nx} \times \texttt{ny} \tag{40}$$

$$\texttt{nw} = 4 \times \texttt{nx} \times \texttt{ny} \tag{41}$$

The number of elements in solution vector **w** is four times this due to all of the conservative variables stored in each cell, shown in Equation 41.

6.2 Adjoint results

For the adjoint results a cost function was J 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 42. While this is not equivalent to lift or drag due to no horizontal or component being taken it shares many similarities in terms of the adjoint equations.

$$J = \int_{\mathcal{W}} p(s) \cdot ds \tag{42}$$

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 $\texttt{nd} = 60$ unless otherwise stated.

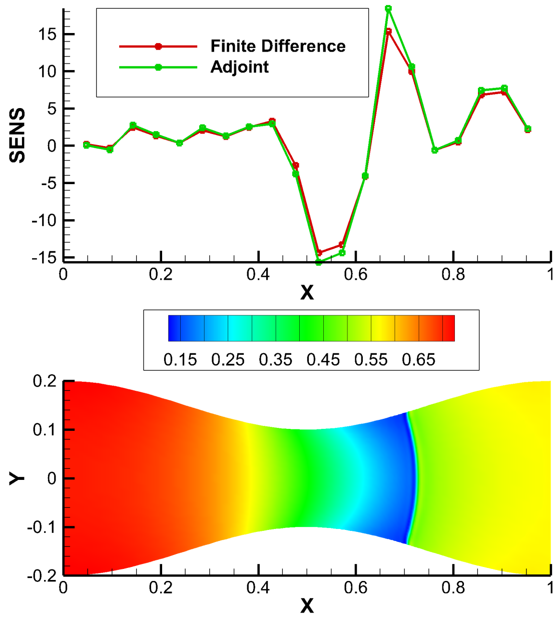

Figure 12: Sensitivity $\frac{dJ}{d\alpha_i}$ of the cost function J with respect to the control points at their respective x-positions. This is shown above a plot of the pressure distribution through the duct

A comparison of the sensitivity $\frac{dJ}{d\alpha_i}$ evaluated via both finite differences and the adjoint method is shown in Figure 12. 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 12 can be found in Appendix C for greater detail. Due to the large data sets used in this project it can be difficult to present all of the information.

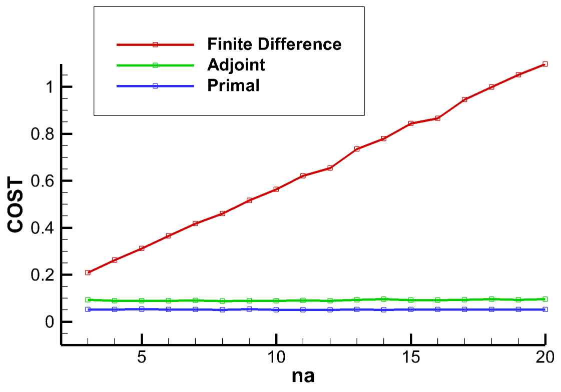

Figure 13: Computational cost in seconds for a variation of number of design variables `na`

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, these are known to be similar to other methods.

The most widely known property of the adjoint method is the independence of the cost from the number of design variables. Figure 13 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.

$$\frac{\partial \mathbf{R}}{\partial \mathbf{w}} \in \mathbb{R}^{\texttt{nw}\times\texttt{nw}} \tag{43}$$

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 as the mesh density increases as the flux Jacobian has $16 \times \texttt{nd}^2$ elements as shown in Equation 43.

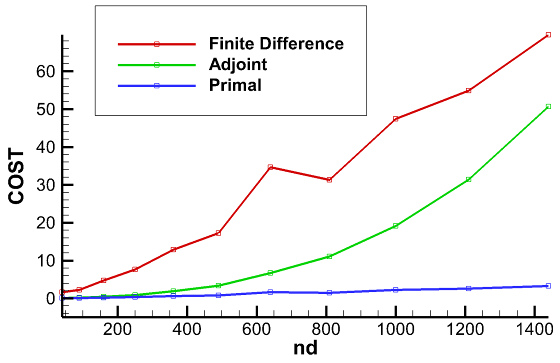

Figure 14: Computational cost in seconds for a variation in mesh density `nd` for `na = 20`

Figure 14 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 cheaper relationship than this but would suffer from similar problems.

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 methods can be found in Appendix C. The solution of the linear system is by far the most expensive part. This takes up 22% of the adjoint solver for a mesh density of `nd` = 250, but 87% for a higher mesh density of `nd` = 2250. The calculation of the partial derivatives take up varying amounts of time. The flux Jacobian is mostly 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. Despite having more elements ($\texttt{nw} \times \texttt{na}$). $\frac{\partial \mathbf{R}}{\partial \alpha_i}$ is far quicker than $\frac{\partial J}{\partial \mathbf{w}}$ with $1 \times \texttt{nw}$ 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 calculated with finite differences then this would have taken much longer than the AD subroutines.

6.3 Bonus results

It has been shown that the most expensive operations in the adjoint metehod all deal with the flux Jacobian, so it is worth noting the structure of this matrix. Figure 15 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 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. While the true condition number of the ratio of eigenvalues is not calculated, it is estimated by dividing the absolute value of the largest non-zero element by the absolute value of the smallest non-zero element. This value is usually equal to approximately 10^{22} for most mesh densities. 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.

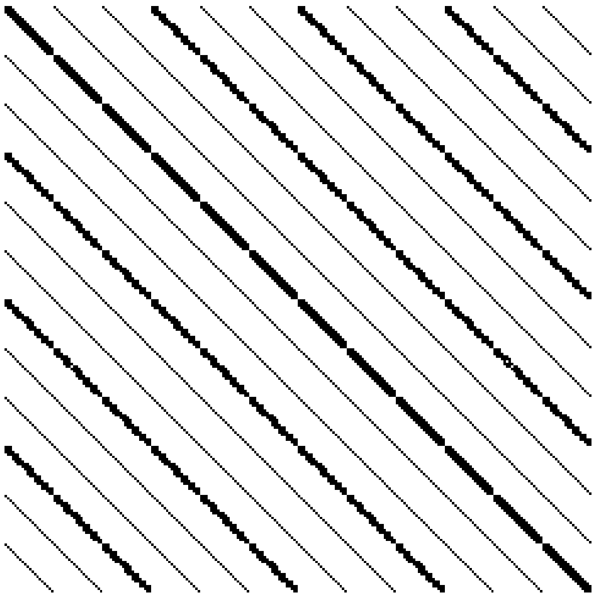

Figure 15: Visualisation of the flux Jacobian $\frac{\partial \mathbf{R}}{\partial \mathbf{w}}$ for the standard mesh density. All non-zero-elements are shown in black

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.

Section to be completed later

Memory over time, RSS or something. Describe process so others can do it.

7 CONCLUSIONS & FUTURE WORK

The objective of this paper was to make the creation of an adjoint solver a quick and easy process. Given that the relevant concepts and theory are understood this is the case. Although a reader completely new to adjoint methods will have a learning curve a researcher creating subsequent adjoint solvers will find it simple as most of the work is done by the AD engine and the linear solver, requiring almost no new code to be written if the primal solver is in the correct form. This is not the case when adjoint codes are done by hand.

Processes that allow the almost automatic creation of adjoint codes have been around for years. This report attempts to reduce that initial learning curve for the new user by reporting more details of the creation of adjoint solvers that research papers take for granted.

“ Draw together the most important results and their consequences, and list any reservations or limitations.

Do not duplicate the Abstract as the Conclusions or vice versa. The Abstract is an overview of the entire paper. The Conclusions are a summing up of the advances in knowledge that have emerged from it. It is acceptable to present conclusions as a bulletpointed list. ”

The objective of this report was to make the creation of an adjoint code very easy, with very little effort for the user. If the goal of the user is simply to find a faster way of calculating the sensitivity than by finite differences this is fine however researchers may want to somehow improve on the work that has been done here. In order to do that a deeper knowledge of adjoint equations than a simple combination of partial derivatives as per the discrete adjoint method. This is why there was an attempt to cover more of the mathematical side in Section 2.

One situation where this might be useful is in the creation of adjoint turbulence models. While AD engines have no issue with differentiating conditional statements most of the time there are situations when it is best to differentiate by hand. In this situation a deep knowledge of how adjoint equations affect nonlinear phenomena and singularities would be very useful, especially given the complicated nature of cutting-edge turbulence models. The researcher could apply AD to the code as usual, then replace the AD turbulence model subroutine with the hand differentiated one and there would be no problem.

While this report has satisfied its objectives there are several areas which would benefit from more work.

The ease of creating the adjoint method has been due to the use of AD, however this would not work on many current primal CFD codes in use. Many of these codes use `allocate()` statements which many AD engines find difficult to find the derivative of. The greater problem is that AD engines by definition cannot differentiate external or ‘black-box’ functions that are not source code written in the primary programming language. So any CAD applications or third-party analysis methods could not be differentiated.

Another issue is finding the Hessian. All gradient-based methods require the grad vector, which is covered in this report, but many optimisers make use of the Hessian matrix as well. It is possible to calculate this using AD in a brute force approach but this is very inefficient, resulting in many unnecessary duplicate calculations. In fact, it is possible for AD to calculate the Hessian very efficiently but this is done by the AD engine and not the user. It is currently an active area of AD research and is slowly being implemented in AD software.

REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] P. He, C. A. Mader, J. R. Martins, and K. J. Maki, "An aerodynamic design optimization framework using a discrete adjoint approach with openfoam," *Computers & Fluids*, vol. 168, pp. 285–303, 2018.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] R. M. Hicks and P. A. Henne, "Wing design by numerical optimization," *Journal of Aircraft*, vol. 15, no. 7, pp. 407–412, 1978.

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

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

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

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

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

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

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

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

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

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

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

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

A AUTOMATIC DIFFERENTIATION: A SIMPLE EXAMPLE

Section to be completed later

$$\begin{aligned} \text{independent variables: } &= t_i, \quad 1 < i < n \\ \text{intermediate variables: } &= t_i, \quad n+1 < i < n+r \\ \text{dependent variables: } &= t_i, \quad n+r+1 < i < n+r+m \end{aligned} \tag{44}$$

$$\frac{\partial t_i}{\partial t_j} = \sum_{k=j}^{i-1} \frac{\partial t_i}{\partial t_k} \frac{\partial t_k}{\partial t_j} \tag{45}$$

B CFD CODE

Section to be completed later

Function call graph

Makefile?

C FURTHER ADJOINT RESULTS

Table 2: Full list of the sensitivities displayed in Figure 12

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

Table 3: Time breakdown for the adjoint method

`na`	20		20	
`nd`	250		2250	
Primal run	0.604370		18.14444	
Adjoint run	0.945372	$\times 1.6$	179.4507	$\times 9.9$
Calculation of $\frac{\partial J}{\partial \alpha_i}$	0.000599	.0636	0.005558	.0031
Calculation of $\frac{\partial J}{\partial \mathbf{w}}$	0.098864	10.4942	2.478229	1.3832
Calculation of $\frac{\partial \mathbf{R}}{\partial \alpha_i}$	0.003643	.3867	0.031062	.0173
Calculation of $\frac{\partial \mathbf{R}}{\partial \mathbf{w}}$	0.631658	67.0492	20.87816	11.6530
Matrix factorisation	0.205986	21.8650	155.6723	86.8878
Matrix solution	0.001287	.1366	0.099270	.0554
Final sensitivity	0.000044	.0047	0.000231	.0001