Imperial College London

Aerodynamic shape optimisation using algorithmic differentiation

Giacomo Sammons
Imperial College London
Department of Aeronautics

Supervised by Dr. Rafael Palacios Nieto

September 15th, 2017

This thesis is submitted in partial fulfilment of the requirement for the MSc in Advanced Computational Methods

Abstract

This investigation focuses on the capabilities of the discrete adjoint solver that is implemented in SU^2, the open-source CFD solver, with a focus on optimisation of aerodynamic shapes in fluid and fluid-structure interaction (FSI) cases. A fluid-only discrete adjoint sensitivity output of a transonic case simulation is demonstrated to match nearly exactly the continuous adjoint output. A shape optimisation routine is used to evaluate the performance of the discrete adjoint when compared to the continuous adjoint using time-efficient convergence criteria. This revealed that the discrete adjoint evaluates gradients poorly when the direct solution is badly converged, but is highly effective afterwards, leading to a superior airfoil within the optimisation constraints. The structural model implemented in SU^2 is evaluated against analytical solutions, revealing some issues with the linear elastic model, but confirming the validity of the nonlinear model. An aerodynamically relevant thin-walled airfoil case is created for use in an FSI cases and optimisation. While the direct solution is valid, attempting to run discrete adjoints on the FSI case proved unsuccessful, impeding opportunities of performing shape optimisation. Further work is suggested in the structural solver to include node positional sensitivity and a deeper investigation into how the algorithmic differentiation tool derives the discrete adjoint equations.

Acknowledgements

I would like to extend my gratitude to my supervisor, Dr. Rafael Palacios, for following me through this project and guiding me when necessary and all the university staff for the great teaching I have received this year.

I would also like to thank my colleagues, friends, but most importantly, family, who have trustingly supported me for so many years of University.

Table of Contents

1. Introduction

1.1 Motivation

The continuous strive to gain a greater understanding of the interaction of fluids with objects has been a driving force behind the development of more accurate, precise and repeatable forms of testing. The aeronautical engineer, in a development environment, is effectively limited to two forms of testing, wind tunnel and computational fluid dynamics (CFD) [1] [2]. Wind tunnel testing has the advantage of being accurate to its non-dimensional testing conditions and has benefitted from upgrades like particle image velocimetry (PIV) [3] [4], which allows one to capture volume data. However, it still suffers from being expensive, has poor repeatability, and is prohibitive to measure all of the flow primaries over the whole flow field.

In contrast, CFD offers to perform the study and evaluation of a flow completely in the computational domain through the discretisation of the governing partial differential equation (PDE) over the geometrical domain. The simulations are, by definition, perfectly repeatable, and all flow primaries are inherently available as part of the simulation.

While in wind tunnel testing every design iteration must be tested individually, simulation poses itself very well for automatic optimisation. There are many optimisation algorithms available, each with their own advantages and limitations. Most commonly, these involve the definition of a cost or objective function that must be minimised within some design constraints. Many of these methods work by parametrising a shape with some design variables and then evaluate them using either direct iterative [5], genetic algorithm [6] or other algorithm to evaluate the function over variations of variables.

While these methods have proven very effective in fields like structural engineering [7], these methods often require orders of magnitude greater computational cost than the simulation alone [8]. Unfortunately, these methods cannot simply be used within the field of CFD, as the direct simulation alone is, for three-dimensional high-Reynolds flows, a compromise between accuracy and computational cost [9]. This makes iteration-based optimisation methods very difficult as they would either be exceedingly computationally expensive, or a reduced resolution simulation must be used. This may develop an optimisation around an inaccurate simulation, and thus a sub-optimal solution [10].

Recent advances in computational fluid dynamics have brought more advanced methods of optimisation which allow the evaluation of gradients of design variables through use of adjoint equations. These methods have been developed and integrated into leading commercial and open-source codes like STAR-CCM+ [11], OpenFOAM [12] and, the code of focus of this paper, SU^2 [13].

Within the field of aviation, the normal practices of aircraft wing design have been to produce a wing around a minimum thickness that ensured structural strength and rigidity [14]. The wing from the structural side would then be designed in such a way that it was stiff enough for the airfoil profile not to deform too much. The wing must also not to suffer from excessive spanwise rotations and deflections. When designing a wing using this method, the downside is that the airfoil is being optimized around an objective, be it lift, drag, or efficiency, while constrained with a specific thickness, which forces it to avoid thinner, but potentially higher-performing airfoils. At the same time, the structure may need to be reinforced due to wing or airfoil deformation being deemed too high and causing a degradation in the performance of the device.

In order to evaluate the effects of the structure on the fluid-dynamic problem, it is possible to add the simulation of the structure in the loop, which is referred to as fluid-structure interaction (FSI) and has been another development in the field of CFD.
There are currently many loosely-coupled methods [15] which perform, in alternation, a number of iterations of the fluid domain, derive the pressure forces from the surfaces of interest, and apply them as load conditions to the structure. At completion of the solution of the structural model, the FSI iteration routine obtains the surface displacements, morphs the fluid domain to reflect these changes and returns to the fluid simulation, for a number of fluid-structure loops until equilibrium is reached. Research [16] has been done on the development of tightly-coupled fluid-structure interaction, but it is not currently available in common solvers as per knowledge of the author.

The existence of fluid-structure interaction models mean that it is possible to evaluate the performance of the wing, thus it is possible to design an optimisation strategy. Due to the even-greater computational cost, it can be concluded that only the most efficient methods of optimisation can be used here. The issue with FSI optimisation is that a fluid-only optimisation is insufficient as changes to the aerodynamic surface may cause changes to the structure, causing changes in the amount of deflection perceived by the aerodynamic device.

1.2 Objectives

The scope of this paper is to investigate the capabilities of the adjoint solver in SU^2, and compare the two different adjoint solvers currently available in SU^2, which are the continuous and discrete adjoint solvers. A comparison between the performance of each adjoint will be measured and the computational cost as well as the accuracy of the calculated sensitivities.

Additionally, an optimisation loop based on the test cases provided with SU^2 will be run using finite difference, continuous adjoint and discrete adjoint, and the resulting design's performance over a fixed number of iterations, or computational time compared.

An investigation into the current state of the structural and fluid-structure interaction implementation in SU^2 is to be performed by running structural and FSI cases against analytical solutions. An evaluation of the change in objective function value of a fluid-only optimisation when evaluated in an FSI case is performed with a typical airfoil construction.

Finally, an attempt will be made to perform optimisation on a FSI case through the use of the discrete adjoint solver.

1.3 Report Outline

Chapter two will focus on the introduction to SU^2 and the workflow used throughout this project. Chapter three will cover the fluid-only simulations, observe and compare the differences between the continuous and discrete adjoint. In chapter four, the structural solver and its discrete adjoint are investigated and some FSI cases are shown and discussed, where attempts are made to perform FSI adjoints. Chapter five discusses in detail the results recorded for the discrete adjoint for the different physics simulations performed. Chapter six briefly reflects on the content explored and the evaluated current state of SU^2 and proposes further work needed to progress on FSI optimisation.

2. The SU^2 solver, meshing and simulation workflow

SU^2, the Stanford University Unstructured CFD code, was originally developed in Stanford for the purpose of solving partial differential equation problems on general, unstructured meshes. The focus of the code is on the solution of Reynolds Averaged Navier-Stokes (RANS), the code has been designed around accepting any partial differential equation (PDE) problem, for the expansion of the code capabilities to other problems. The code is capable of simulating compressible and turbulent flow.

The code is also built around the constrained-optimisation problem, with particular focus in aerodynamic shape optimisation through adjoint-based shape optimisation [17]. The shape optimisation is achieved internally through avoidance of re-meshing by morphing of the mesh to a different shape.

The code has been recently expanded to include thermodynamic simulations, and, as part of an effort by the aeroelastics department at Imperial College London, a structural solver and a Fluid-Structure interaction solver. The FSI solver in SU^2 is developed as a loosely-coupled FSI problem, with the fluid-simulation based on a Eulerian reference frame and the structural simulation on a Lagrangian frame that must be linked to one another via a Lagrangian-Eulerian formulation of the problem.

2.1 The adjoint solver

At the core of the optimisation algorithms in SU^2 is the use of adjoint partial differential equations. The primal equations are derived such that it is possible to calculate the gradient of a function with respect to a certain quantity of interest. This can be done by either the continuous adjoint method, which involves manual derivation of the equation and then the discretisation for implementation into the CFD code. Alternatively, the adjoint may be obtained through the calculation of the adjoint on the discretized equations. This poses a great opportunity from the point of view of complex, multiphysics problems, where the code developers must only worry about the discretisation of the PDE problem and the adjoint equations can be compiled later.

In order to better understand the mechanics of the adjoint, it is possible to take as an example a one-dimensional advection equation. The domain is discretized into four nodes, placed at x = 0, 1, 2, 3. A positive advection velocity is used in the domain such that the propagation of information is only in the positive direction. The matrix representing this discretisation would appear as:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ c_1 & c_2 & 0 & 0 \\ c_3 & c_4 & c_5 & 0 \\ c_6 & c_7 & c_8 & c_9 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

where *c* are coefficients of the matrix due to the upwind scheme used. As it can be observed, the dependence of each node's value is dependent only on the upwind values, hence the lower-triangular matrix.

If we were to take the adjoint of the domain with respect to some function of interest *F*, the resulting system of equations is just:

$$\begin{pmatrix} 1 & c_1 & c_3 & c_6 \\ 0 & c_2 & c_4 & c_7 \\ 0 & 0 & c_5 & c_8 \\ 0 & 0 & 0 & c_9 \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \begin{pmatrix} \partial F/\partial u_0 \\ \partial F/\partial u_1 \\ \partial F/\partial u_2 \\ \partial F/\partial u_3 \end{pmatrix}$$

which is the transpose of the original matrix. The matrix is now upper-diagonal, meaning that the information is being propagated in the opposite direction to the flow. It is important to understand that the adjoint here is the tangent linear system based around the solution, so it is always linear in λ. As such, even if the direct simulation may be nonlinear, the computation of the adjoint is going to be linear, meaning it can be much quicker to solve than the direct simulation. For example, the forward model needs n iterations to reach a solution close to machine accuracy, the adjoint will only need a single iteration to achieve the same machine accuracy, thanks to being linear.

The solution must have converged in order for the adjoint to have validity. If an unconverged solution is used to determine the adjoint, the adjoint solution will take far longer to converge to solution, hence it is important to converge the direct problem for both accuracy and quick adjoint.

2.2 Meshing and simulation workflow

SU^2 is an open-source, free project, with the most accessible, active repository hosted on github.com, a Git-based remote repository for software. Git is version-control software that monitors and tracks changes in computer files and code through intelligent detection of the changes made and stores every "committed" change made as a series of differences to the original commit. This allows developers to simultaneously work on the same code base or files and then merge their changes when they have completed their tasks. On a greater scale, teams may "branch" the code to focus on long-term development of a specific feature, like a thermal solver and work independently on the code until the feature is complete and then merge it back. In most projects, there will be many branches, with in-progress features, a development branch with the latest features integrated together, and a master branch, which

gets updates from the development branch once the developed features have been proven to be stable and useable.

The installation of SU^2 is simple for individuals who are proficient with the terminal, a UNIX-based shell. The installation instructions are available on the SU^2 Github wiki page (https://github.com/su2code/SU2/wiki/Simple-Build) and can be compiled in different ways, depending on the requirements of the user. These include the integration or not of the CoDi algorithmic differentiation tools that are necessary for the performing of discrete adjoint flow solutions.

SU^2 also comes with a number of example cases showcasing the different capabilities of the software, which include SU^2's native mesh files (.su2) and a configuration file (.cfg). The geometry files from which the mesh was generated are not available, meaning that any change to the mesh resolution or the modification of the geometry cannot be performed. For this reason, a method of generation of meshes from geometry was necessary.

The open-source, free software GMSH [18] was chosen for its 3D unstructured and structured meshing capabilities, as well as its integrated mesh output directly to SU^2 native mesh. The open-source nature of the code means that if some functionality were to be desirable for those using it in conjunction with SU^2 could code it into the program. GMSH is also free, meaning the community can easily access the same tools available to the professionals and academics, improving repeatability of experiments. The code is also compatible with major operating systems like Windows, MacOS and Linux distributions.

GMSH utilises a primitive method of definition of geometries, via a text-based sequence of commands that define the entities via dimensional progression of definitions. This file (which uses a .geo extension) can be created either via the use of the graphical interface or programmatically. In order to quickly mesh varying airfoils and shapes, a Python program was written, which read raw point coordinates from a comma-separated file, which contains the x and y coordinates of the airfoil, or shape. In its final iteration, the code can produce a two-dimensional mesh, with a specified boundary layer structured mesh within a rectangular domain as well as fully defined physical surfaces for inlet, outlet, walls and airfoil. The code also included the functionality of performing a spline fit of the points and then a resampling of the points, in order to obtain a smoother surface on the mesh. Meshing was mostly done locally but could also be performed fully programmatically through terminal. A mesh output of the transonic RAE2822 is shown below in figures 2.1 and 2.2, which was upsampled from 120 to 300 surface points. These were used as surface mesh for the Mach 0.75 run, with a very fine volume mesh of approximately 0.005mm in height at the wall, increasing by a factor of 1.2 as it gets farther from the wall, to a total distance of 0.1m away from the surface. From there, an unstructured mesh was utilised to mesh the rest of the domain.

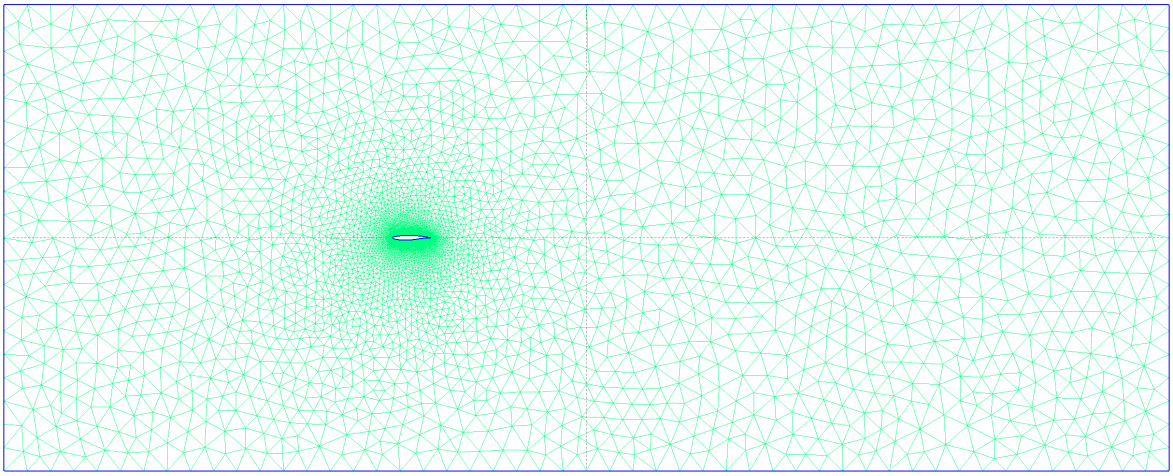

Figure 2.1: Far-field view of fluid mesh around RAE2822

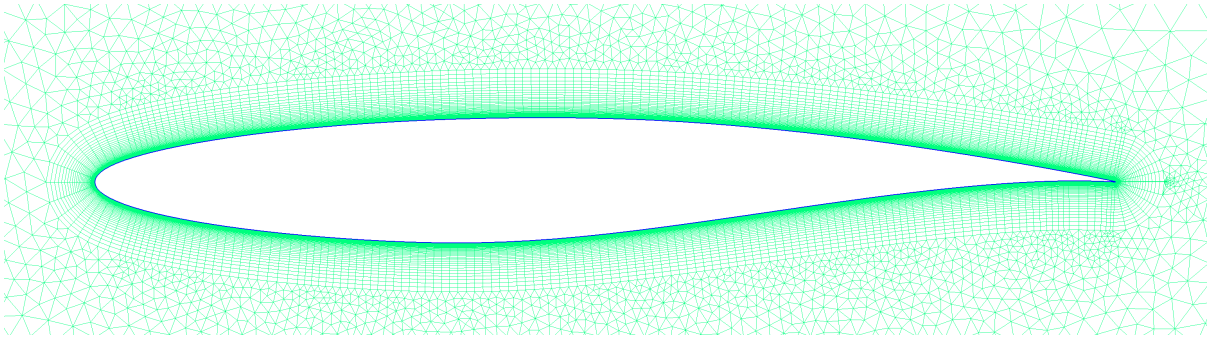

Figure 2.2: Close-up view of airfoil and boundary layer in RAE2822 fluid mesh

The structural mesh was generated as a thin wall airfoil structure by offsetting the airfoil surface inwards, and defining the 2D airfoil structure as a hollow airfoil with a reinforcement spar at 0.2 lengths from the leading edge. For use in 2D structural cases, a clamping point was defined at the location of the spar. In order to save on computational time, the mesh was only produced for the trailing edge as an assumption was made that deformation only occurs on the trailing part of the wing skin. The structural solver in SU^2 supports only quadrilateral mesh, so the mesh was defined accordingly, in order to produce a semi-structured mesh made wholly of quad elements, as shown in figure 2.3. In order to use a structural and a fluid mesh together in an FSI case, the meshes must match for the transfer of forces between the fluid and the structure. A current development in SU^2 at the time of writing addresses the issue by integration of a radial basis function into the code base, but it was decided against the use of it due to its in-development status.

The mesh resolution was chosen as a mix between sufficient structural resolution and good shape factor and acceptable fluid convergence time. The fluid mesh was approximately 30'000 elements and the structural was around 2'000 elements.

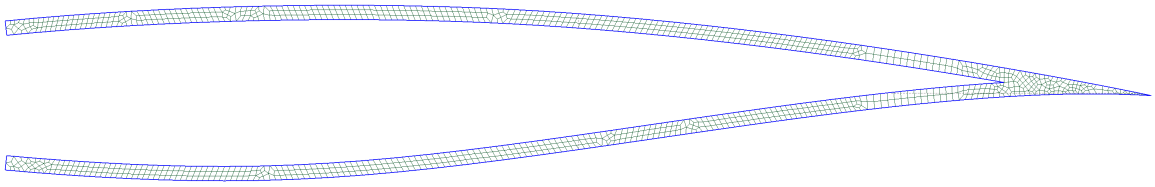

Figure 2.3: Structural mesh for RAE2822

Once the geometries were meshed, the jobs were run on remote servers. All cases were run on Linux machines, as the CoDi library used as part of SU^2 discrete adjoint implementation was only possible on Linux operating systems.

All job submissions and running was performed through secure shell (ssh) sessions, on either the “spitfire” 44 CPU-core server or “linux1” 32 CPU-core server. None of the simulations were run at higher than 16 cores, as these computational resources were shared amongst students and staff alike.

Post-processing was performed on ParaView [19] [20], as SU^2 only supports Tecplot and ParaView output, and ParaView is open-source, free and available for major operating systems, as well as possessing advanced features like remote, headless rendering and complex manipulation algorithms for unstructured meshes.

Calculation of the computational time used will be performed via the use of the `time` command in Linux prepending the command being run. The output includes the real time that has passed as well as the number of core-seconds utilized. This has the advantage of improving repeatability on a computer with varying CPU load over the day, as when the computer is heavily loaded, the real time is affected but the core-seconds are not, because they represent the cumulative time that all cores have spent performing tasks on the CPU.

In order to ensure repeatability of the results, meshes and configuration files from the SU^2 codebase were used as much as possible. The cases utilized in the paper and their referred name are listed below, in table 2.1, with a brief description of the flow configuration used, when not specified otherwise.

Table 2.1 – Summary of SU² cases used and file location

Description	**Location of files**	**Case description**
Transonic NACA airfoil	/Quickstart/	2D compressible, steady euler NACA0012 airfoil at 1.25° angle of attack, with freestream inlet velocity at Mach 0.8, and ISA pressure and temperature
Transonic RAE2822	/TestCases/rans/rae2822/	2D compressible, steady Reynolds-averaged Navier-Stokes of RAE2822 airfoil at Mach 0.729, angle of attack 2.31°, ISA pressure and temperature
Square-cylinder	/TestCases /fea_fsi/SquareCyl_beam	2D FSI case with unsteady compressible flow around a square cylinder with a trailing edge beam. Trailing edge beam structural case with induced vibration by vortex shedding.
Choked pipe flow	/TestCases /fea_fsi/WallChannel_2d	2D FSI case with steady compressible flow of channel with a "dam" restriction. Dam structural case, clamped at the channel wall
2D static beam	*Self-generated*	2D structural case of clamped beam under either tension or end-load bending
RAE2822 FSI [19]	*Self-generated*	2D FSI case with fluid-side as per "Transonic RAE2822" but 0.01*chord thick walls clamped at 20% chord position for structural case
NACA0012 FSI [20]	*Self-generated*	2D FSI case of NACA0012, with incompressible flow and structural case with .01*chord thick walls clamped at 20% chord position
Rhombus FSI	*Self-generated*	2D FSI case of high-aspect ratio rhomboidal shape at small angle of attack in compressible subsonic flow. Structural case clamped at the leading portion of the top surface.

3. SU^2 optimisation

3.1 Comparison of the adjoint techniques

The optimisation loop in SU^2 is based on the definition and manipulation of design variables through measurements of sensitivity of the objective function to any of the input variables. These may be calculated through adjoints or finite difference. Given the focus of this study, an investigation into the computational cost and differences between continuous and discrete adjoint is performed.

The first case that will be observed is the transonic NACA airfoil case. In figure 3.1, it is possible to observe the field Mach, with the presence of a shock on the upper and lower surface. Some shockwave dilution is visible due to the low mesh refinement, but the case is still valid. The airfoil surface is coloured by the continuous adjoint output channel "Surface_Sensitivity" which captures the sensitivity of the surface node in the normal direction of the surface, with respect to the objective function. The objective function set in this case was drag. A positive number represents an increase in the objective function, so an increase in drag. As it can be observed, drag is very sensitive to changes at the suction surface near the leading edge, where the optimisation points to a 'flattening' of the area. The surface sensitivity colour range was restricted to -100/+100 as the data range was dominated by the trailing edge nodes, with a sensitivity in the order of 10^5

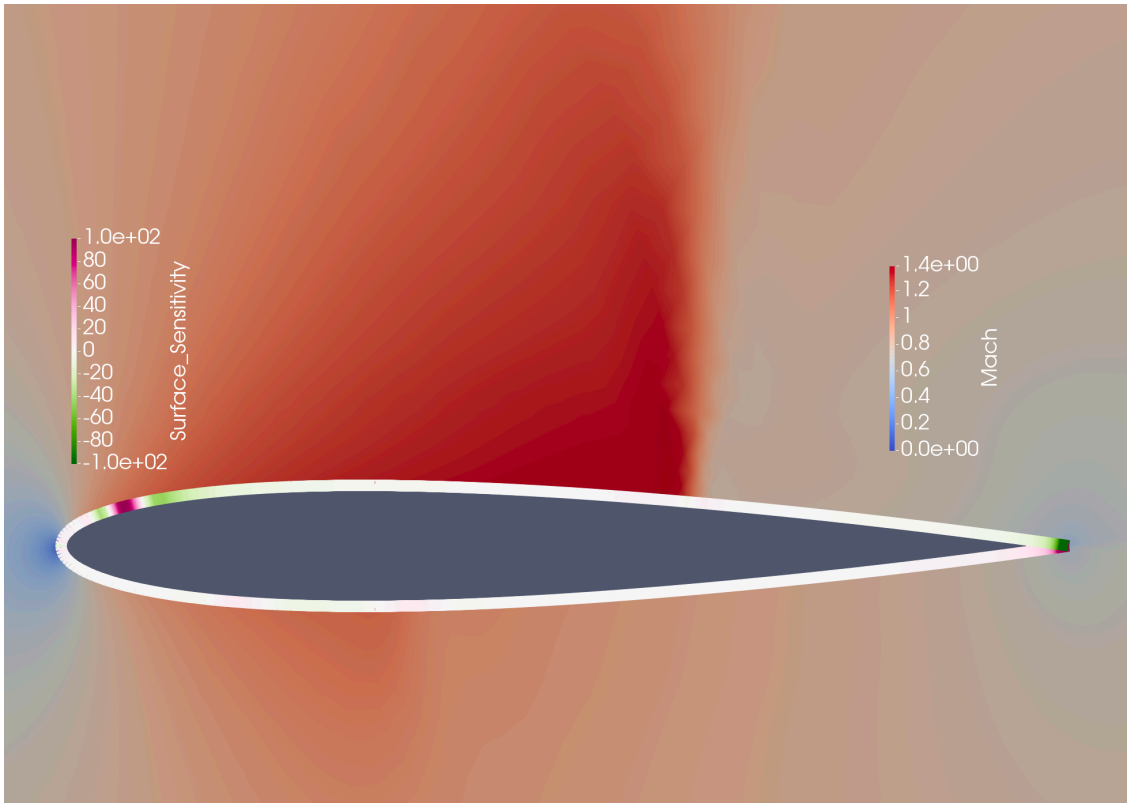

Figure 3.1: transonic NACA airfoil – field output for Mach and surface sensitivity to drag (surface sensitivity range constrained between -100 and 100)

In order to compare the accuracy of the continuous and discrete adjoint flows, the surface sensitivity for the discrete and continuous adjoint is calculated by running the direct case until full convergence. Continuous and discrete adjoints were then run for 2500 iterations, where both showed full convergence with residuals below 10^{-12}. The plot of the two surface sensitivities can be observed in figure 3.2, with chord position on the x-axis. It can be observed that there is a small difference between the two results, including a more accentuated sensitivity being measured by the continuous adjoint at the leading edge, but otherwise an agreement on the location and direction of the surface-normal sensitivity. While both the simulation and their adjoints were run until full convergence to machine error, the presence of differences between them can be accounted to the different mathematical steps involved, and hence the propagation of machine error.

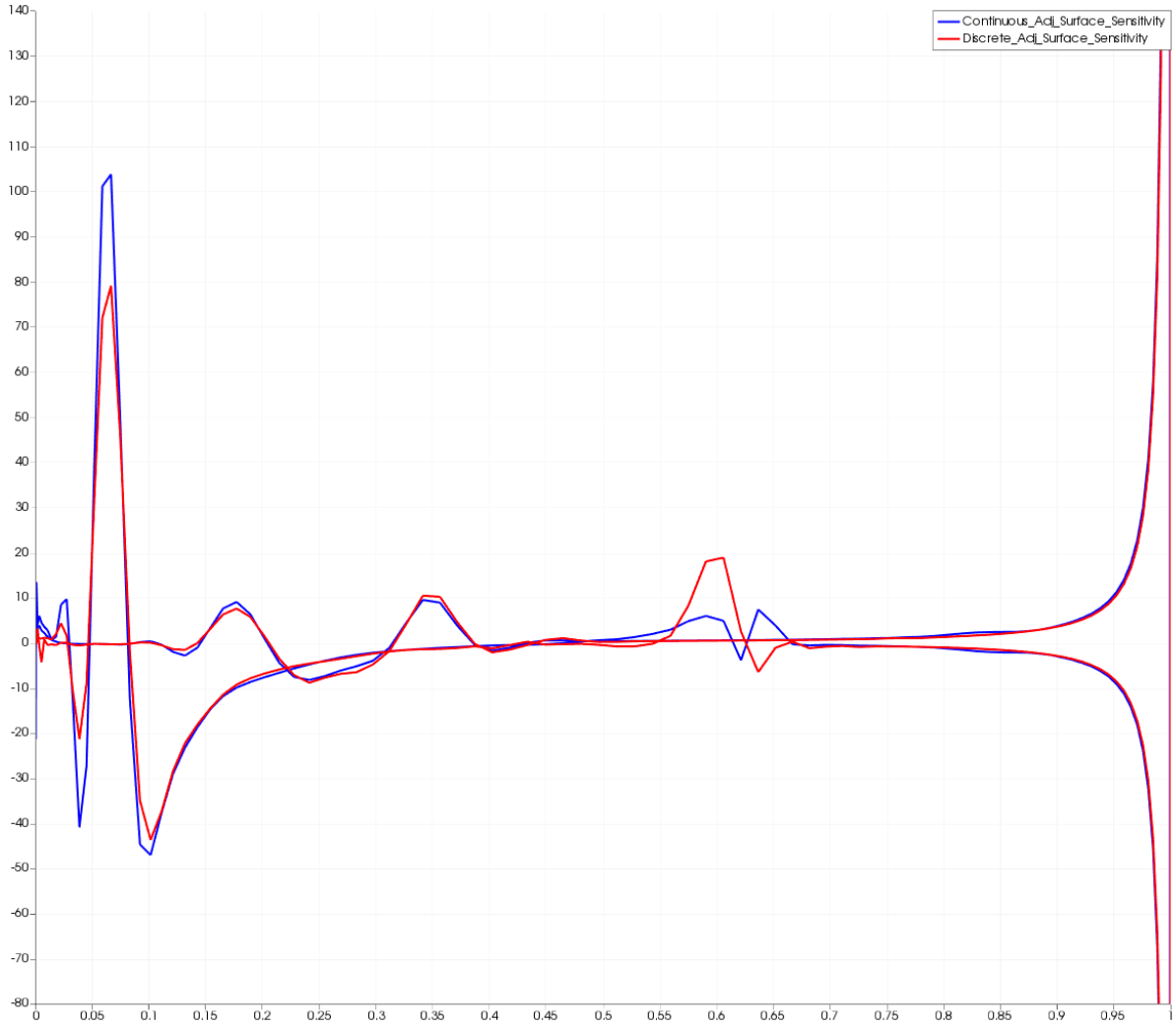

Figure 3.2: surface sensitivity (y-axis) of transonic NACA case of the continuous and discrete adjoint with respect to chord position (x-axis)

A study on the number of iterations necessary to achieve machine accuracy on the discrete adjoint can be performed to understand the target iteration count, as each iteration's computational cost is slightly greater than 2 for the case being considered. Figure 3.3 shows a plot of the surface sensitivity for the fully converged case (650 iterations), with the discrete adjoint run for 25, 250 and 2500 iterations. The discrete adjoint shows full convergence at 1800 iterations so the 2500-iteration case is the same one as that shown in figure 3.2. It is clear that the 25-iteration case is far from correctly capturing the sensitivity, but the 250-iteration is sufficiently accurate.

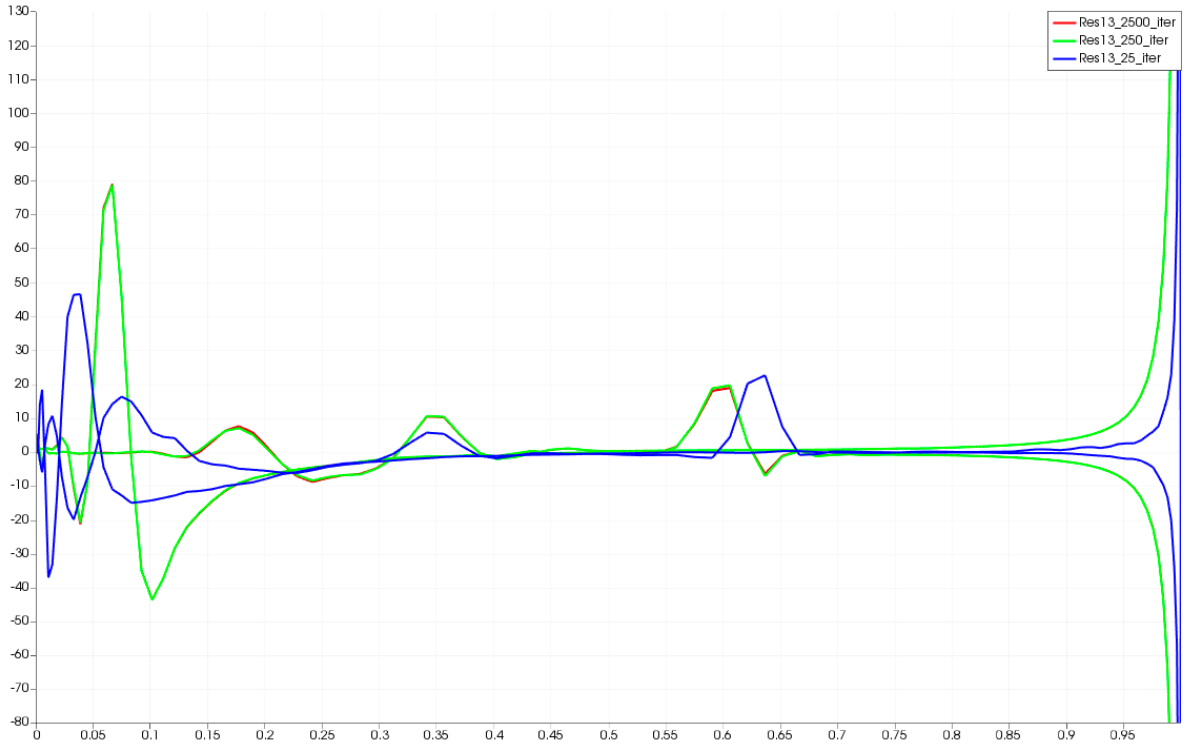

Figure 3.3: Surface sensitivity (y-axis) of transonic NACA case of the discrete adjoints for different iteration count with respect to chord position (x-axis)

Given the nature of the discrete adjoint, it was necessary to compare the accuracy of the adjoint with respect to the accuracy of the solution. This was performed by running the transonic NACA case at different convergence residual targets. This is shown in figure 3.3. All sensitivities are visually identical, with the lines superimposed on one another, even though the solve time to achieve the density residual of 10^{-13} took 219% longer to run than the 10^{-6} residual case.

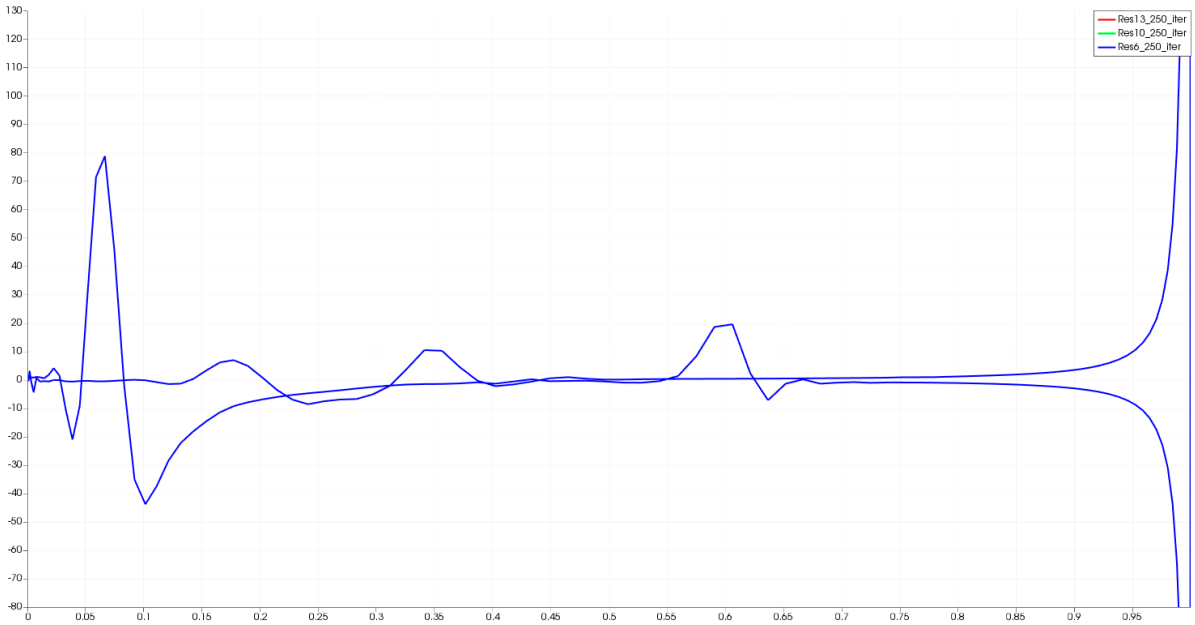

Figure 3.4: Surface sensitivity (y-axis) of transonic NACA case of the discrete adjoints for 250 iterations on cases with different states of convergence with respect to chord position (x-axis)

3.2 SU^2 optimisation techniques and limitations

SU^2's optimisation algorithms, written in mostly in the Python programming language, are designed to perform optimisation through the evaluation of sensitivity to a set of design variables. As per version 5.0, SU^2 can leverage any of three methods to perform an evaluation of the sensitivity, which may be by either finite difference, continuous adjoint or discrete adjoint. For optimisation of shapes that have few design variables, like that of an angle-of-attack optimisation, finite difference may be valuable and be sufficiently fast, but for high-resolution shape optimisation only adjoints are acceptable forms of shape optimisation. Within the SU^2 solver, flow-only optimisations can be easily done via the use of the `shape_optimization.py` Python script. When called, special flags can be defined, including the method of sensitivity analysis, as mentioned above, via the gradient flag `-g`, where the options are `CONTINUOUS_ADJOINT, DISCRETE_ADJOINT, FINDIFF`.

A study on the effectiveness of the continuous and discrete adjoint was performed, to investigate the computational cost of each and evaluate the asymptotic performance of each, where the minor discrepancies observed in figure 3.2 may come into play and affect the fully converged optimisation. For this optimisation, the transonic RAE2822 will be used, which is a Navier-Stokes solver with Spalart-Allmaras turbulence model.

Firstly, the two adjoints were compared based on their one-shot optimisation performance, which is the performance obtained in their first optimisation loop. Below, in figure 3.5, the volume flow, coloured by Mach, can be observed, as well as the surface flow, coloured by normal-direction sensitivity to the objective function, which is drag.

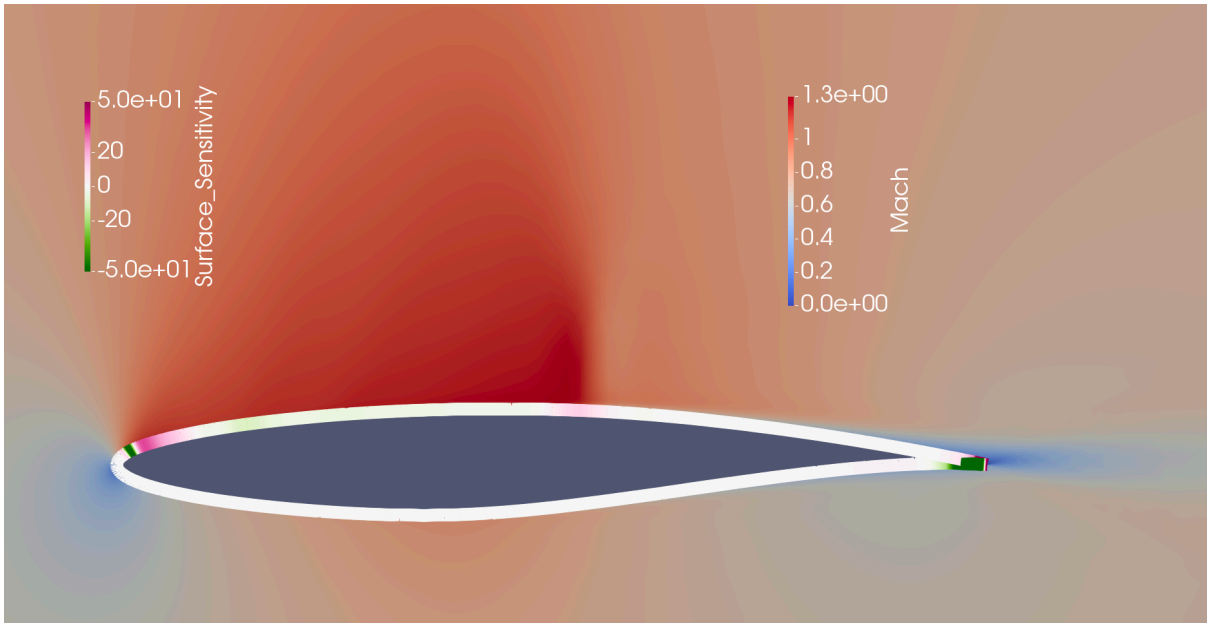

Figure 3.5 - transonic RAE2822 airfoil – field output for Mach and surface sensitivity to drag (surface sensitivity range constrained between -50 and 50)

In the same manner to figure 3.2, a plot of the surface sensitivity generated on the first iteration is compared, to inspect for any differences between the adjoints. This can be observed in figure 3.6, where the sensitivity of the discrete adjoint is orders of magnitude different from that of the continuous adjoint. For this reason the plot in figure 3.6 presents the discrete adjoint's sensitivity channel divided by a factor of 1000. This may be due an incorrect sensitivity measurement due to a failed use of the scaling factor by the discrete adjoint-based optimisation. For the case used, the optimisation had these configurations:

```
% Optimization objective function with optional scaling factor
OPT_OBJECTIVE= DRAG * 0.001

% Optimization constraint functions with scaling factors
OPT_CONSTRAINT= ( LIFT > 0.723 ) * 0.001
```

The visible difference between the scaled discrete adjoint and the continuous adjoint may be due to the configuration chosen on the optimisation loop. The adjoint case for both the discrete and continuous adjoint exhibited a convergence residual oscillating in the 10^{-4} region, so a maximum of 1000 iterations in the solver was used. Due to a current limitation with SU[2] in prescribing different convergence criteria, both the direct solver and adjoint solver share iteration count, hence the potential issue with the first-iteration comparison. Figure 3.7 shows the same data as figure 3.5 but with the surface coloured by the discrete adjoint output instead of the continuous adjoint. Interestingly, the continuous adjoint shows no sensitivity to the pressure-side of the airfoil while the discrete adjoint does.

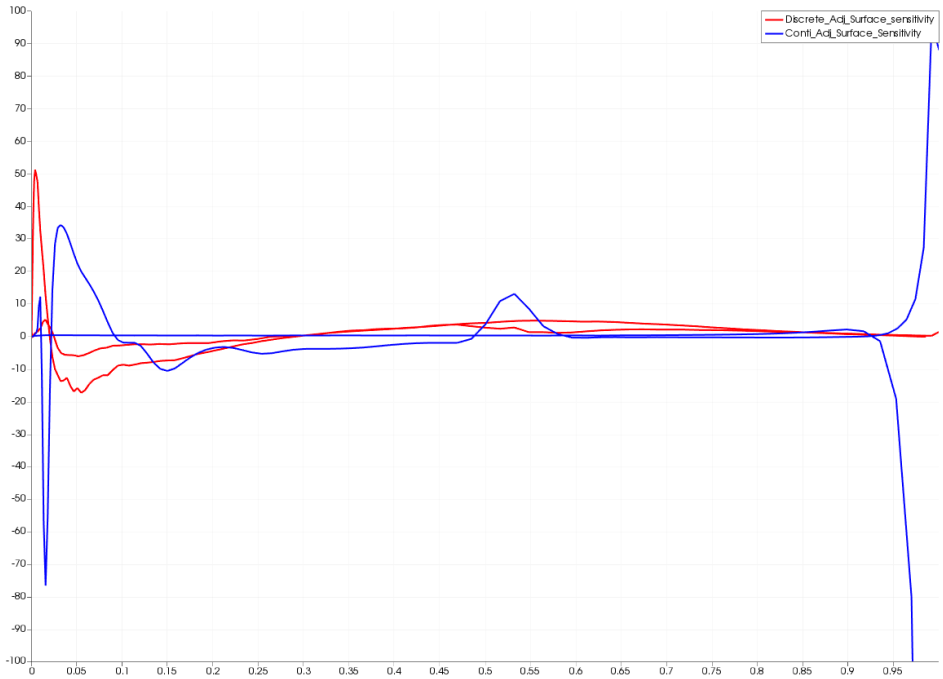

Figure 3.6: Surface sensitivity (y-axis) of transonic RAE2822 case of the discrete adjoints for different iteration count with respect to chord position (x-axis).

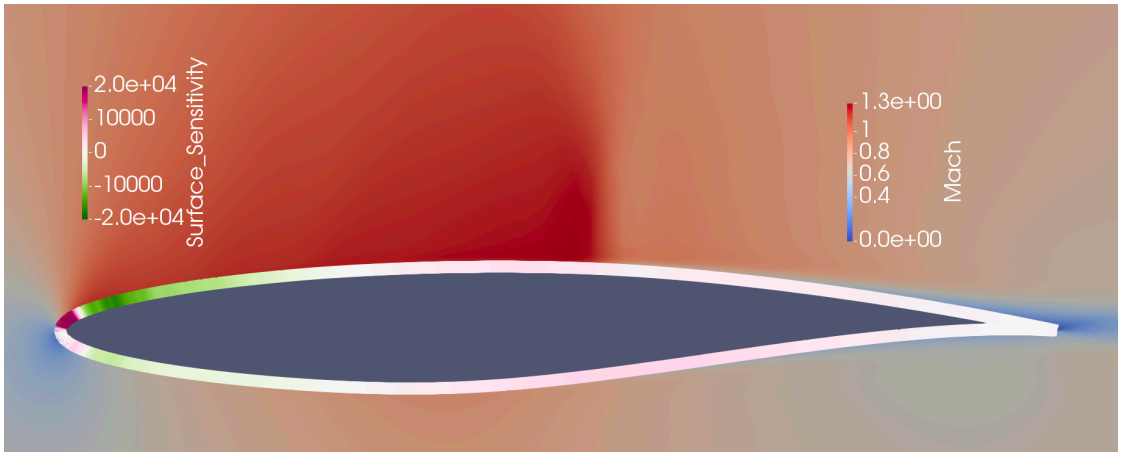

Figure 3.7: transonic RAE2822 airfoil – field output for Mach and surface sensitivity to drag (surface sensitivity range constrained between -2000 and 2000)

When looking at the one-shot optimisation, it is possible to appreciate the accuracy of the two adjoints. A one-shot optimisation is valuable as it represents an optimisation that can be performed within the same order of magnitude of the simulation's computational cost. Within the optimisation, only wall time (real time that passed during solve) was available, so both optimisations were run on 20 cores on a machine that did not exceed system resources and thus throttled the simulation. When observing figure 3.8, it can be appreciated that the discrete adjoint not only takes far longer to perform the zeroth iteration, but also perceives a substantial increase in drag. The plot is filtered by runs that meet the optimisation constraint of lift exceeding 0.723 only.

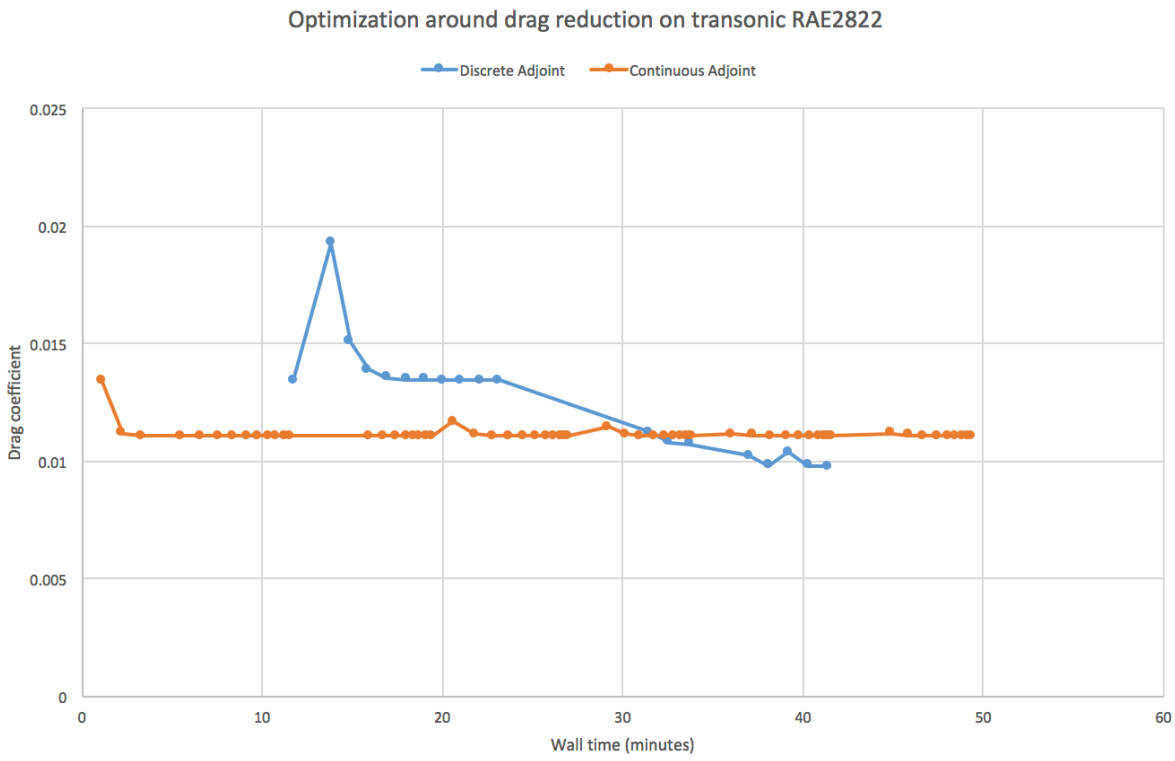

Figure 3.8: transonic RAE2822 airfoil optimisation of drag (y-axis) over wall time(x-axis)

It can also be noticed that the discrete adjoint optimizes the airfoil further, meeting a drag objective of 0.0097, or 12% lower than the continuous adjoint. The level of convergence for both optimized shapes was satisfactory and a discrepancy between the two methods was initially pointed to the nature of the discrete adjoint which optimizes the shape around the discretized mathematical problem rather than the continuous adjoint, but the resultant shapes, which can be observed in figure 3.9 and 3.10, reveal that the discrete adjoint had put far more pressure to the airfoil to reduce the thickness. It is important to note that no restriction was put on airfoil thickness, thus the optimisation performed by the discrete adjoint does not infringe optimisation restriction. Note that in figures 3.9 and 3.10 the surface sensitivity is still largely higher in the discrete adjoint than in the continuous adjoint.

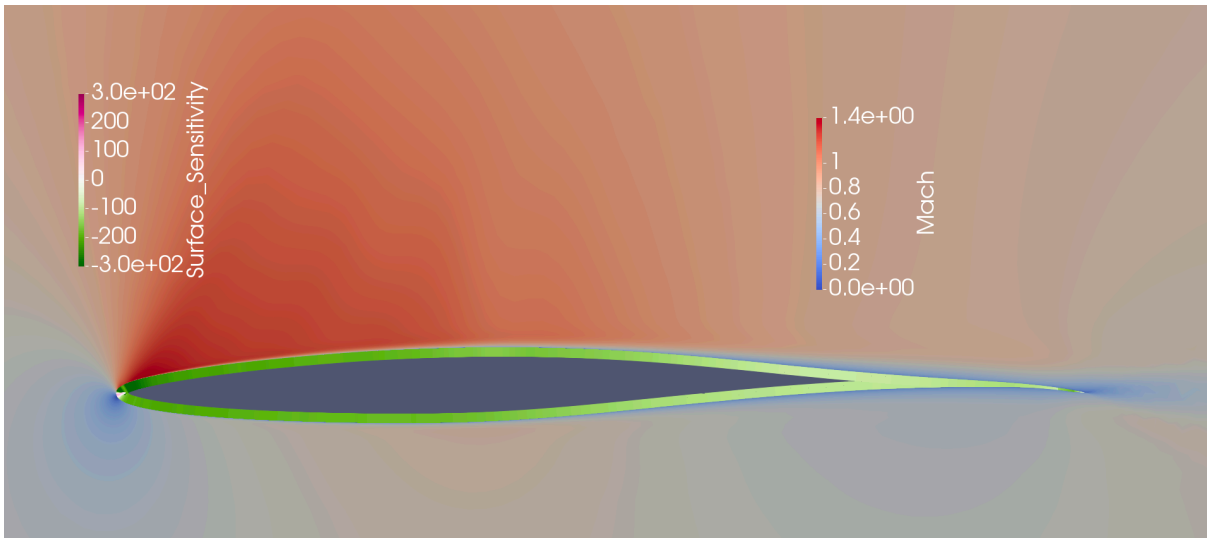

Figure 3.9: transonic RAE2822 airfoil (optimized by discrete adjoint) – field output for Mach and surface sensitivity to drag (surface sensitivity range {-300 300})

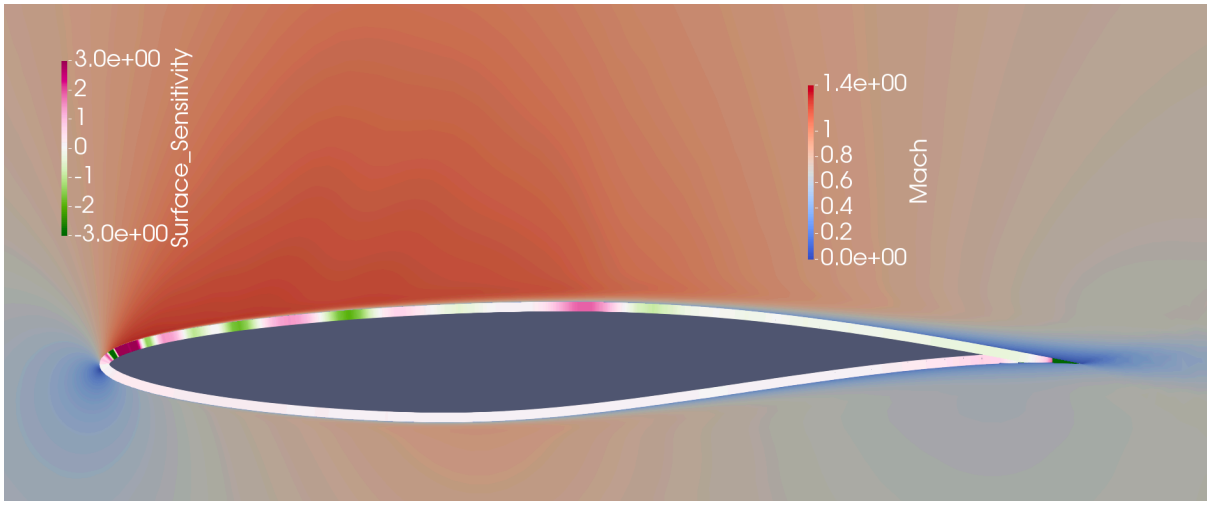

Figure 3.10: transonic RAE2822 airfoil (optimized by continuous adjoint) – field output for Mach and surface sensitivity to drag (surface sensitivity range {-3 3})

4. Structural solver and FSI optimisation

4.1 The structural solver in SU^2

The second solver that is necessary for fluid-structure interaction is the structural solver. SU^2 already has the capability to perform structural problems in two and three dimensions, as well as linear and nonlinear elastic solver, but is only restricted to isotropic materials. The solver can define different material properties (Young's modulus and Poisson's ratio) for different areas, but involves complex configuration of additional files, which was not performed in this project due to time limitations.

In order to utilize the structural solver, it was deemed necessary to evaluate its results on a toy problem and compare it to the analytic results. A simple two-dimensional beam was modelled and meshed in GMSH, and run in the SU^2 structural solver, using a 30m long beam, with 1m height and a Young's modulus of 3*10^9 Pa and Poisson's ratio of 0.2. A tensile and a cantilever beam cases were set up. Figure 4.1 shows the deformation of the beam due to a tensile load. The beam is clamped at the left end, and free at the right end. The tensile test is performed by applying a -10kPa pressure on the free end

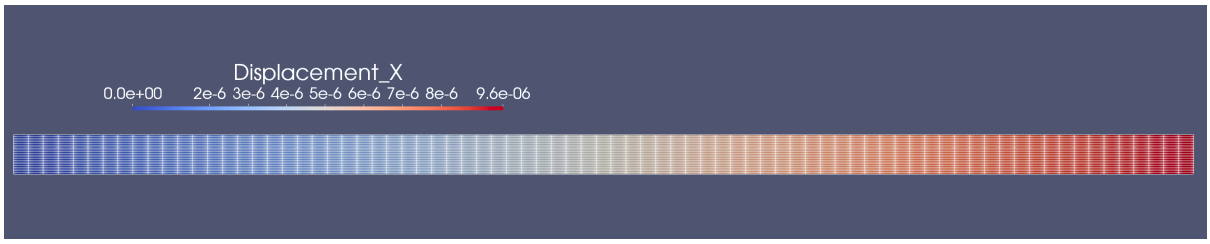

Figure 4.1: 2D Static beam tensile test. Mesh display in white and coloured by displacement in the tensile direction.

Because this is a planar strain, the equation for plane stress-strain must be used. For a one-dimensional strain, the equation for stress-strain is:

$$\sigma_x = \frac{1}{1-\nu^2} \cdot \varepsilon_x$$

For the material properties defined, the strain at the end should be 9.6*10^{-6}, which is equal to the result as shown in figure 4.1. The second load case, shown in figure 4.2, is that of an end-loaded cantilever beam. The beam was loaded on the edge with a load of 1kN. It can be observed from its deformed shape that it is not the expected, canonical solution. The configuration used for this case was that of a steady, linear elastic model. The model performs a single iteration due to the nature of the deterministic, direct method, so the solution would not need iterations to propagate the information through the numerical

domain. The solution shown in figure 4.2 instead shows what would appear as a time-dependent FEA at an early step where inertia is delaying the deformation.

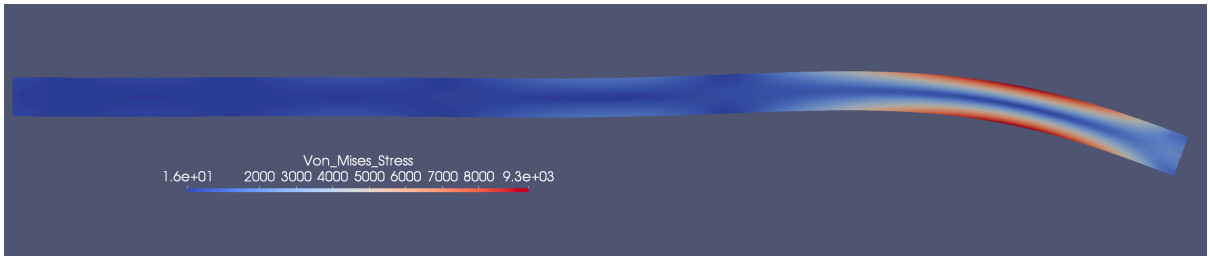

Figure 4.2: 2D Static linear-elastic beam bending test. Coloured by Von-Mises stress. Deformation scaled by a factor of 10'000

The issue encountered, shown in figure 4.2, was resolved by use of the NEO_HOOKEAN material model, which is a nonlinear FEA model. When run for 100 iterations, the solution, coloured by vertical deflection, can be observed in figure 4.3.

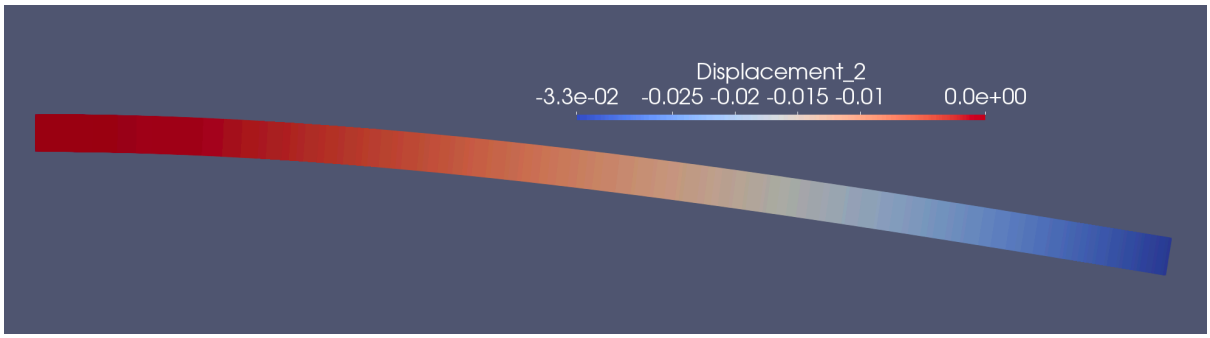

Figure 4.3: 2D Static nonlinear-elastic beam bending test. Coloured by vertical displacement. Deformation scaled by a factor of 100

The beam, in figure 4.3 now appears to have deformed expectedly, with a tip displacement of 32mm. This can be compared to the analytic solution based on the classical beam theory's equation for an end-loaded beam, which is:

$$w(x) = \frac{1}{1-\nu^2} \cdot \frac{Px^2(3L-x)}{6EI}$$

where I for a square beam is $\frac{1}{12}$, so the analytical solution for this beam, loaded 1kN at the end, is of a deflection of 34.5mm, which is 5.8% greater than the simulated deflection of 32.6mm. This may be due to discretisation errors or due to the nonlinear solver being used, which is in contrast with the classical beam theory's assumptions. Given the issues raised in the LINEAR_ELASTIC model, the structural solver used for the remainder of this paper will be based on the NEO_HOOKEAN solver configuration, as, to the author's knowledge, it appears there are still issues with the linear elastic model.

The discrete adjoint was performed on the structural model, to inspect the output of the adjoint. The beam on the fine mesh used on the cases shown in figure 4.1, 4.2 and 4.3 was diverging consistently, causing highly-varied sensitivities. Repeating the direct simulation for 3000 iterations to ensure full convergence of the direct solution, in the hope that the discrete adjoint's divergence was caused by a poorly converged solution, did not yield an improvement, except a reduced rate at which the discrete adjoint's residuals increased per-iteration. The case was thus remeshed with half of the resolution, which converged in the direct solver within 3-4 iterations. The discrete adjoint, similarly, converged within 10 iterations. The reduced-resolution mesh was validated again for the tensile case and the bending case, and deformed identically to the fine-mesh case, within 0.1% accuracy.

The output currently available from performing structural adjoints is limited. The discrete adjoint objective function used was a reference node objective. This objective is prescribed by defining the node to be tracked, and can also accept a target deflection. For simplicity, the top right corner of the beam was set as the tracked node, with the target deflection set to (0, 0). The objective function was configured in the case file as:

```
OBJECTIVE_FUNCTION = REFERENCE_NODE
REFERENCE_NODE = 2
REFERENCE_NODE_DISPLACEMENT = (0, 0)
REFERENCE_NODE_PENALTY = 1E3
```

The ParaView field output generated by running the discrete adjoint solver `SU2_CFD_AD` on the beam case, `beam_adj.vtk`, contains all zeros, but the other file, `Results_Reverse_Adjoint.txt`, the output, tabulated in table 4.1.

	Obj_Func	Sens_E_0	Sens_Nu_0
0	1.729216026285308e+01	-5.764037794169806e-09	3.189081368185833e+00

Table 4.1: Tabulated output of static nonlinear-elastic beam bending discrete adjoint, with top-right objective reference node set to zero deflection.

The output, shown in table 4.1, describes the sensitivity of the objective function to a change in the material input variables, which for the linear isotropic material prescribed, is the Young's modulus, E, and the Poisson Ratio, Nu. As expected, the objective function is equal to the modulus of the deformation vector of the tracked node, multiplied by the reference node penalty of 1000. As expected, for an objective that effectively prescribes a reduction in displacement, an increase in the stiffness value has a negative effect on the objective function (a reduction in objective function is an improvement). The Poisson's ratio instead has a positive relation to the objective function. This is expected, as a higher Poisson's ratio reduces the plane stiffness of the material due to plane strain conditions in two-dimensions, as shown in the fraction term equation 4.1.

In order to validate the result of the discrete adjoint of the beam bending case, a finite-difference analysis can be used to validate the discrete adjoint's results. This was performed for both the Poisson's ratio design variable as well as Young's modulus. Below, tabulated in table 4.2, are the results for the two variations.

Modification	Objective function	Change from baseline
Poisson's ratio -0.01	1.725847969105045e+01	-0.0336805718
Poisson's ratio +0.01	1.732222673515432e+01	0.0300664723
Young's modulus $+1{\cdot}10^{8}$	1.673435011513211e+01	-0.5578101477
Young's modulus $-1{\cdot}10^{8}$	1.788843995561198e+01	0.5962796928

Table 4.2: Tabulated output of static nonlinear-elastic beam bending finite difference, with top-right objective reference node set to zero deflection.

By central difference, the slope was evaluated for each input variable as 3.1873 and $5.77044{\cdot}10^{-9}$, which are consistent with the discrete adjoint results. A similar study was performed using a three-dimensional beam under tensile and beam bending cases, and direct results agreed with analytic solutions, as well as the discrete adjoint validated by a finite difference approach, as done for the two-dimensional case for table 4.2.

4.2 Fluid-structure interaction solver in SU^2

The fluid-structure solver in SU^2 is developed around the separate solving of the fluid equations and the structural equations, with an interface FSI solve that serves the purpose of transferring the variables of interest across the two domains. Flow solution properties of interest at the interface, such as shear rate and pressure, are transferred to the structural solver to be used as boundary load conditions. Similarly, once the structural solver has met convergence criteria, the node locations to the mesh-morphing toolset in SU^2. This can be observed in figures 4.5 and 4.6, where the deformation of the pillar structure has deformed the mesh significantly at its tip.

In the case of a dynamic simulation, non-zero velocities are defined due to the movement of the structure. This can be seen in figure 4.4, where the line integral convolution (LIC) shows localized velocity direction turns to a normal velocity at the wall, where a velocity is prescribed due to dynamic movement simulation from which the snapshot was taken.

Having performed both fluid-only and structure-only cases and their respective adjoints, an approach to performing discrete adjoint was made. The cases currently available based on fluid-structure interaction are two. The first one is an unsteady compressible flow solver with dynamic structural simulation. This case captures the vortex-induced vibrations due to flow around a square cylinder, and the galloping that occurs on a cantilevered beam hinged on the back face of the square cylinder. A snapshot of one of the time steps can be viewed in figure 4.4. An attempt to perform fluid structure interaction on this case was not possible as the discrete adjoint solver in SU^2 currently is not capable of performing adjoints on dynamically moving meshes like that presented in this case.

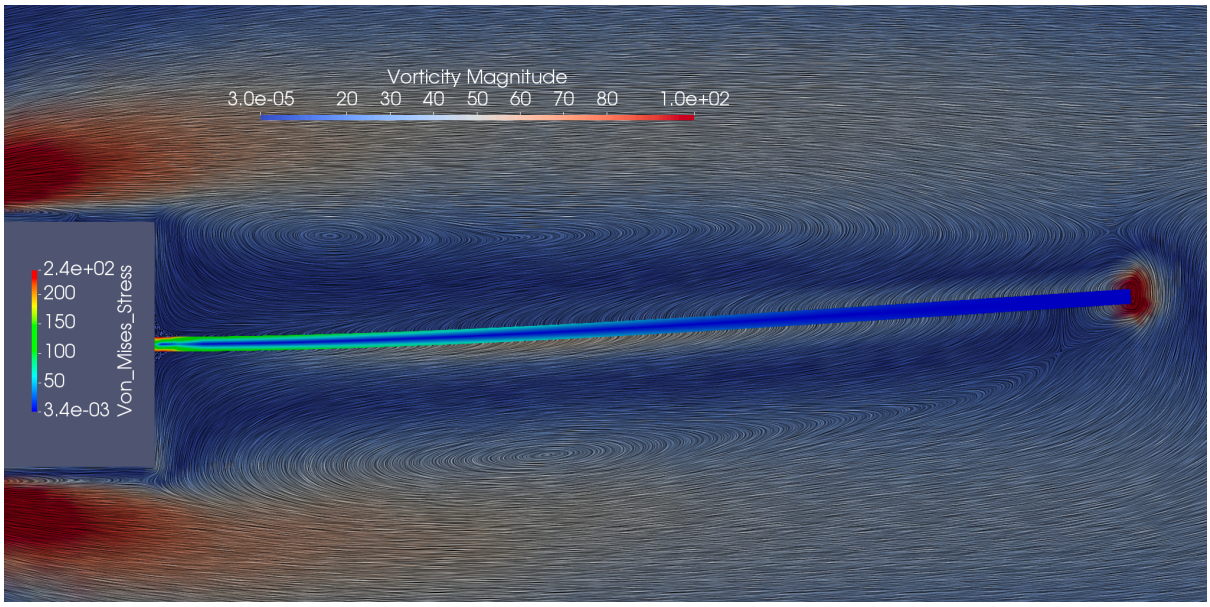

Figure 4.4: Buffeting beam behind square-cylinder fluid-structure interaction case. Flow volume coloured by vorticity, line integral convolution of velocity, and structure coloured by Von Mises stress

The second case available in SU^2 is a two-dimensional dam choking a channel flow. The dam is simulated as a structural problem and the pipe as a compressible Navier-Stokes flow. A close-up of the area of interest for this case can be observed in figure 4.5. It can be observed that, as per the current capabilities of the solver, the meshes' edge nodes match. A feature for the use of the radial basis function to allow the use of mismatching meshes is under development but was not used to minimize the potential causes of failure.

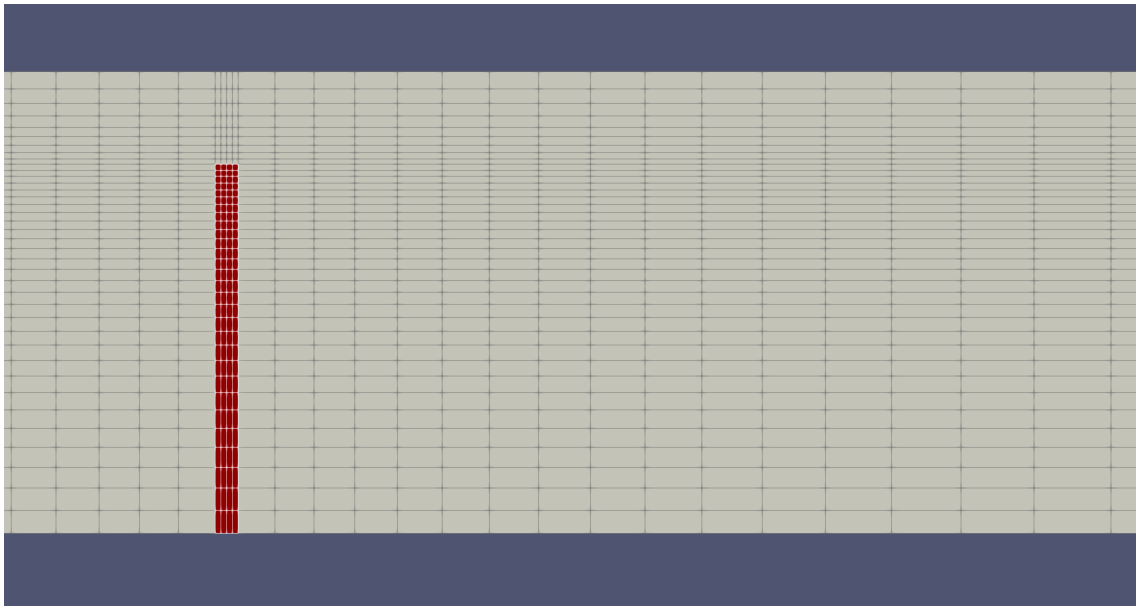

Figure 4.5: FSI choked pipe flow case. Fluid mesh colored by gray, structure mesh in red.

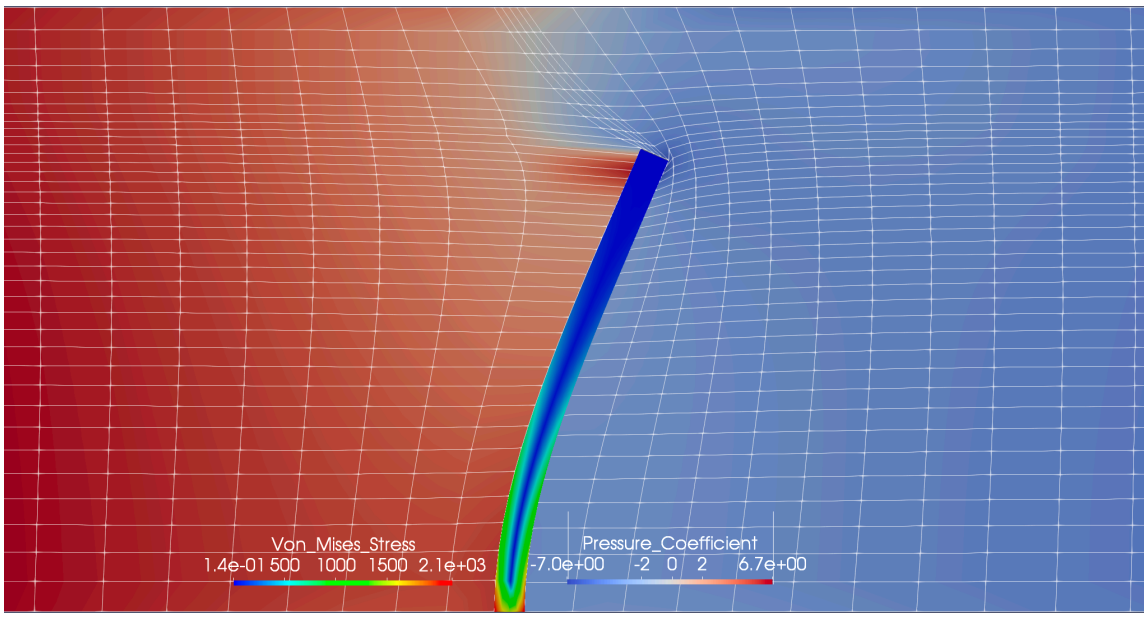

Figure 4.6: FSI choked pipe flow case at equilibrium on the FSI interface

As this case was set up as a dynamic simulation and is very atypical of a wing, which is the subject of interest of this piece of research, a more representative case was set up to capture the typical effects of an airfoil/wing. Given the need to avoid computational cost, a two-dimensional simulation was preferred. The typical airliner wing structure [21] was used by modelling the fluid-structure problem as a NACA0012 fluid flow, but with the structure only modelled from 20% chord to the trailing edge, under the assumption that the stiffness of the leading edge in conjunction with the leading-edge spar. The mesh representation of the model can be seen in figure 4.7, which is the overlay of the meshes shown in fig. 3.2 and 3.3.

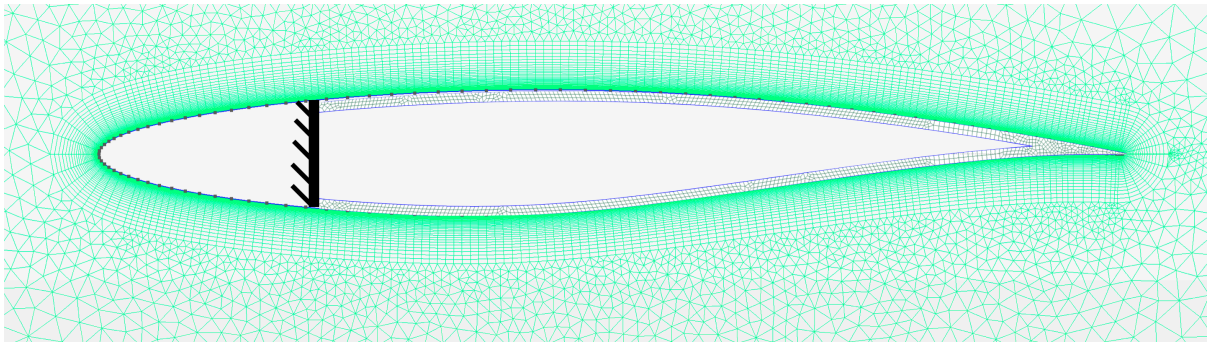

Figure 4.7: RAE2822 FSI problem set up. Black band represents clamped conditions applied to the structure

This case was meshed for both the RAE2822 and NACA0012 case, with a boundary layer mesh that appropriately captured the boundary layer for a full Navier Stokes simulation. This simulation was revealed to be too computationally expensive, with a single direct FSI simulation taking around a day on 20 cores. The solution of this case, a NACA0012 airfoil, is shown in figure 4.8. The flow was an incompressible zero angle-of-attack flow at 30m/s, with air-like fluid properties.

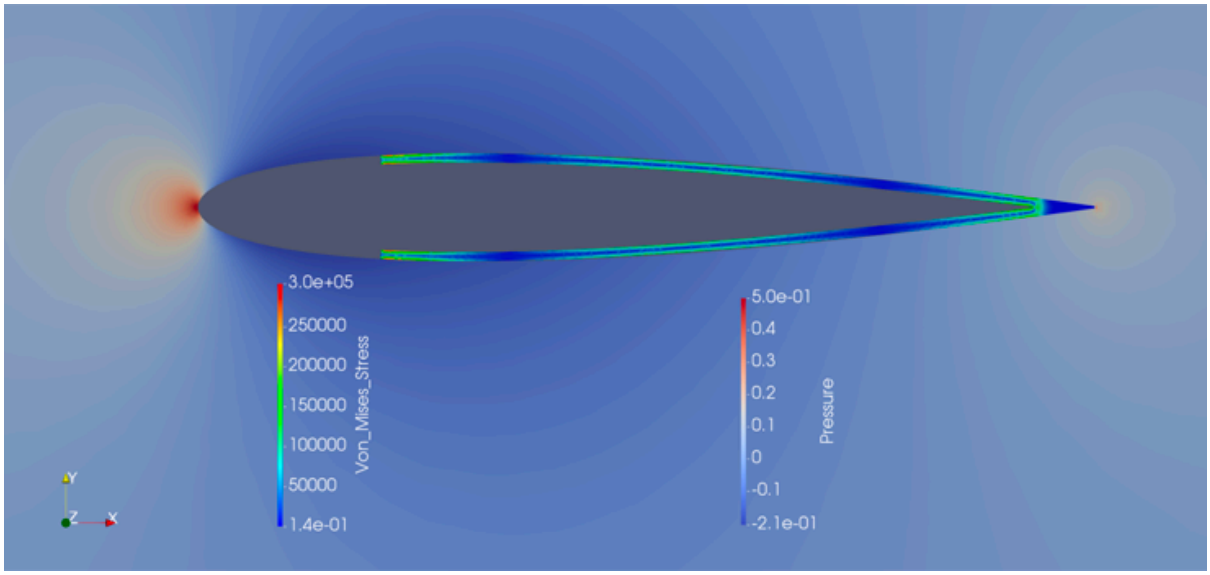

Figure 4.8: NACA0012 FSI case, Fluid coloured by pressure, structure by Von Mises stress

Due to the prohibitive computational cost of the Navier Stokes flow case described above, an additional, very simple Euler model was prepared for more rapid turnaround times. This case is a simple high-aspect ratio rhombus, placed at a small angle of attack. The rhombus was clamped at the suction side up until 50% of the chord. A deformation of the FSI case run can be observed in figure 4.9, as well as the mesh that was defined for the problem. The low mesh count for the structure and the fluid meant that the simulation had a far quicker turnaround time of 15 minutes on 12 cores to full convergence on the direct problem. The full solution of the FSI can be seen in figure 4.10.

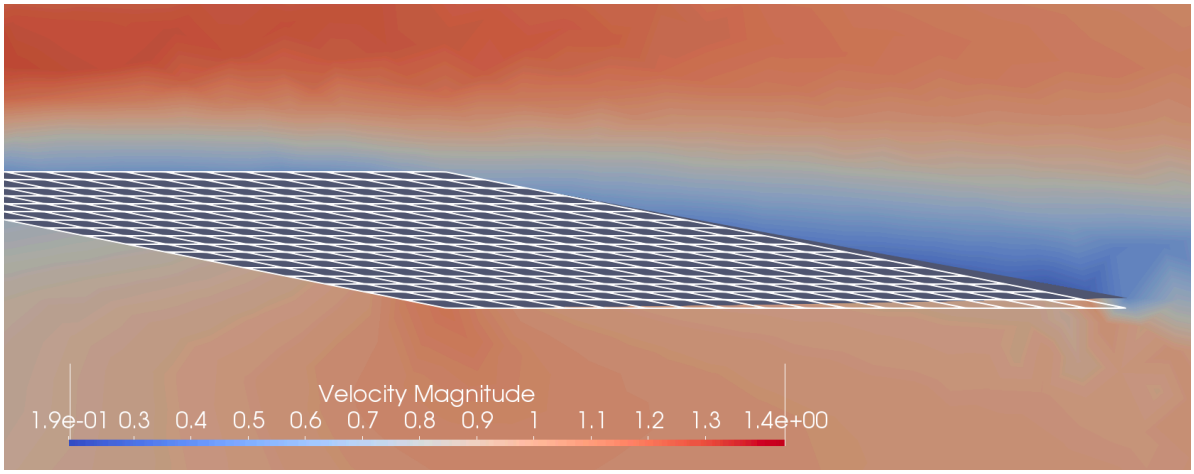

Figure 4.9: Rhombus FSI case, focused shot on structural mesh and trailing edge deformation.

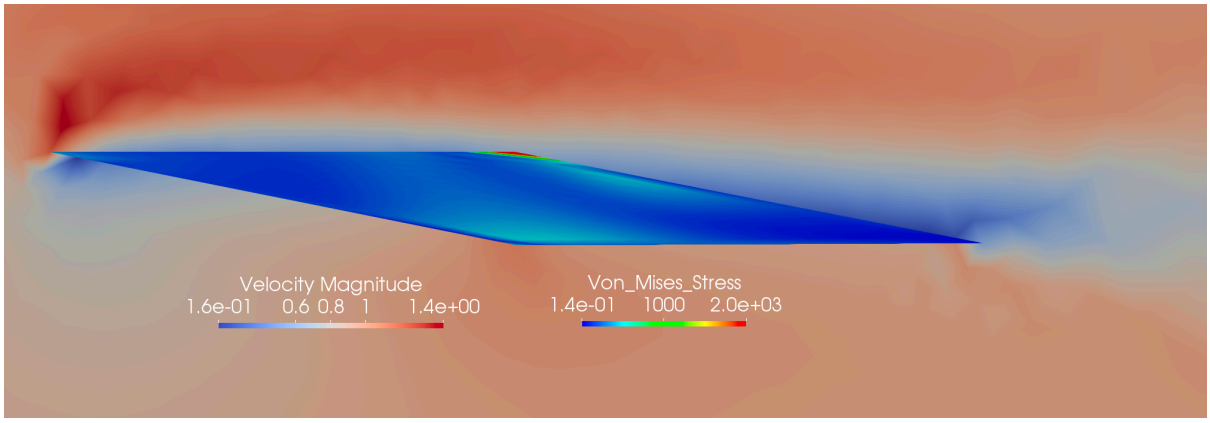

Figure 4.10: Rhombus FSI case, Fluid coloured by velocity, structure by Von Mises stress

Utilizing this case, several attempts were made at performing a discrete adjoint solve for the FSI case. Issues were encountered both when using an objective function focused on the fluid domain, like that of drag, and when using an objective function of reference node, like that used in the structural cases in figure 4.3. Upon inspection of SU^2's terminal window output, it appears as if the discrete adjoint appropriately performs the adjoint of the model whose objective function it relates to first, like a structural adjoint for the reference node one, but then immediately diverges on the first iteration of the FSI interface. This is shown in figure 4.11 and 4.12. A substantial number of attempts were made to successfully perform a discrete adjoint on the FSI case by using different cases, different material properties, convergence criteria, matrix solvers and mesh resolutions but to no avail.

```
------------------------------ Begin Solver -----------------------------

                    ****** BGS ITERATION 0 ******

Structural iteration: structural input -> structural output
Direct structural iteration to store computational graph.
Compute residuals to check the convergence of the direct problem.
Objective function: 2.27915.
UTOL-A: -1.295818e+01, RTOL-A: -8.578086e+00, ETOL-A: -2.270621e+01.
Objective function (Reference Node): 2.279147e+00

 Iter    BGSIter    Res[Ux_bar]     Res[Uy_bar]       Sens_E       Sens_Nu
       0          0       4.182973        4.963771   -7.0642e-05   -2.6359e+02

 !!! Error: SU2 has diverged. Now exiting... !!!
```

Figure 4.11: FSI case adjoint solver divergence when using Reference Node objective

```
  975          1      -2.415534      -2.786314
  980          1      -2.416325      -2.789605
  985          1      -2.417262      -2.793032
  990          1      -2.418410      -2.796624
  995          1      -2.419818      -2.800394

Geometry iteration: geometry input -> flow output
Direct flow iteration to store computational graph.
Compute residuals to check the convergence of the direct problem.
log10[RMS Density]: -1.277011e+01, Drag: 4.023789e-02, Lift: 4.752496e-01.
Objective function (Lift coefficient): 4.752496e-01

Structural cross term from geometry: structural input -> geometry output
Direct mesh deformation iteration to store computational graph.
Deform the grid using the converged solution of the direct problem.
CSysSolve::ConjugateGradient(): system solved by initial guess.
CSysSolve::ConjugateGradient(): system solved by initial guess.

Structural iteration: structural input -> structural output
Direct structural iteration to store computational graph.
Compute residuals to check the convergence of the direct problem.
UTOL-A: -1.295818e+01, RTOL-A: -8.578086e+00, ETOL-A: -2.270621e+01.

 !!! Error: SU2 has diverged. Now exiting... !!!
```

Figure 4.11: FSI case adjoint solver divergence when using Lift objective

5. Discussion

5.1 Results obtained

As per the mathematical formulation of the discrete adjoint, the accuracy and first-iteration residuals affected the discrete adjoint considerably. This was shown very clearly when the surface adjoint output of the NACA0012 case which, after full convergence of the direct and adjoint solution, was compared for the continuous and discrete adjoint solver. The solutions matched nearly exactly at all points on the airfoil except at the shock discontinuity. This is likely due to a poor residual due to the numerical scheme being used, but the resulting surface sensitivity was correct at the critical design area of the leading edge of the airfoil.

With respect to the fluid-only optimisation, while it was clear that the results for the two adjoints discussed above were identical, a study on the time-constrained optimisation performance of each adjoint method was performed. Due to an issue with the adjoint-flow convergence criteria, which cannot be defined differently from the direct solver and seemingly not respecting convergence criteria, the adjoint solution was only allowed 1000 iterations. The direct solver instead had useful convergence criteria based on residuals, but was also limited to 2000 iterations. As a consequence, as it was shown in figure 4.8, the discrete adjoint solver suffers due to its longer solve time which was measured to be over twice slower per-iteration than the direct solver. Additionally the discrete adjoint's first redesign is reached in at the same wall time as the 4^{th} continuous adjoint design iteration, and produces a design that worsens the objective function. This was attributed to the insufficiently converged direct solution causing a high adjoint on the discrete adjoint solve, which in turn did not satisfactorily converge to a good sensitivity calculation.

As the simulation progressed, where each iteration uses the previous iterations' solution as a restart file, the residuals of the direct solution improved. As such, the discrete adjoint gains accuracy and reduces the iteration count, modifying the shape beneficially to the objective function. It was observed in the results section that within the folder stack generated by the optimisation loop's trace, the adjoints from the discrete were of higher magnitude than the continuous adjoint. This was not fully understood, but the effects were of a more aggressive deformation of the airfoil, leading to a more inconsistent following of the constraint and improvement of the objective. The direct comparison of one method to another via the use of a single configuration of solver settings is, by the differing nature of the two adjoints, unbalanced, as the discrete adjoint is more sensitive to convergence parameters than others. The alternative of using different configurations for each may have caused other issues as the configuration could have been indefinitely optimised for each. The time dedicated to improving one adjoint's settings over another may have led to an unbalanced comparison or a result that was dependent on a specific case like that of a transonic airfoil.

The structural case, with the configurations used, showed a minor issue with the linear elastic solver due to misunderstood reasons, which were outside of the scope of this project. For this reason the nonlinear elastic model was used, even though it is slower to reach the solution and is unnecessary due to the small relative size of the deformations with respect to the size of the structure.

The results from the study on the adjoint of structural cases revealed that an adjoint can be performed on the structure through the use of the automatic differentiation of the discrete adjoint. The output, a sensitivity of a specified region to a change in material properties like Young's modulus and Poisson's ratio, is demonstrated correct via finite difference simulations. The use of this adjoint may be useful in the field of complex structures like that of a spaceframe, where one may be seeking which portion of the structure is most sensitive to an increase in Young's modulus. This can be interpreted as the equivalent of a thicker-walled tube. This would allow one to maintain the weight of a structure by reducing the material in low-sensitivity regions while adding to high-sensitivity regions to the objective function for a specific deformation objective function.

The FSI cases were run successfully from the lessons learned from the example cases and repeated refinement of the FSI step size was done to prevent divergence. The examples available within SU^2 aren't of particular interest to the assessment of airfoil-structures under FSI conditions, but the NACA0012 and rhombus cases were more representative. Under FSI conditions, both measured a reduction in lift due to the upwards deformation of the airfoil, particularly at the trailing edge.

The FSI adjoint case was unsuccessfully attempted via the use of different cases, using different mesh resolutions, structural solvers, fluid solvers and convergence criteria but with no success. It is complex to understand the dynamics behind this issue, as the automatic differentiation tools are reliant on a specific statement of the PDEs via the use of expression templates. The FSI case is, from an algorithmic point of view, far more complex than an iterative single matrix problem, but instead is a coupled model involving an alternation of two different matrix systems. A speculated reason for why the discrete adjoint could not be run is that the objective function, which defines the variable on which the adjoint is to be derived, is not being appropriately being applied to the full FSI solve cycle. Instead, the automatic differentiation is being performed on each of the two physics separately. This could lead to an attempt at formulating, for example, a density adjoint of the structure rather than the whole FSI iteration. This could happen due to the nature of the internal-external iteration loop causing the iteration not to be recorded correctly by the differentiation tool.

5.2 Problems encountered

This project, while simple at the outset, turned out to be composed of many, diverse tasks and new programs and mathematical concepts. These required time in order to grasp the underlying concepts or understand their functionality in the solver. This has occurred for the learning of use of the GMSH meshing tool, the discovery and refinement of its more advanced feature sets like boundary layer meshes and structured meshes, and the development of a script that can programmatically generate the mesh from primitive coordinate-list airfoils.

A complex problem that required significant investigation occurred in the early stages of the project where geometries created in GMSH would run for a number of iterations in SU^2 and then throw a memory deallocation error (figure 5.1), which often means that the C++ code is attempting to call a segment of memory that has been previously removed. Interestingly, if run a sufficient number of times, the solver was able to write the flow surface file before the memory corruption error bubbled to stop the program, which is shown in figure 5.2 as field output. This was believed to be an issue on the SU^2 side and extensive debugging using lldb, an advanced C++ debugger that can attach to running code and identify the sources of exceptions and errors, was used. This proved to be both very time consuming and difficult, due to the extensive code base of SU^2, but an attempt at running a case of a flow around a square beam from GMSH, proved successful. After further attempts it was discovered that the definition of a geometry in GMSH via the use of the `Spline{…}` command was the cause of these errors. Spline was chosen due to its superior snapping of the nodes to a curvature, rather than to discretized segments. Additionally, using a spline avoided the forcing of a mesh node at every airfoil input point, causing an inconsistent mesh resolution over the chord of the airfoil. The generation of the mesh had to be changed to use via lines, but due to the coarseness of the output, a re-sampling script was used by fitting a spline to the point and then sampling the spline for more fine and equally-spaced nodes.

```
  997   0.039744    -6.205093     -5.994813       1.173238       0.024401
  998   0.039757    -6.204987     -5.997428       1.173248       0.024409
  999   0.039743    -6.204821     -6.000227       1.173258       0.024417

------------------------- File Output Summary -------------------------
Writing comma-separated values (CSV) surface files.
Merging connectivities in the Master node.
Merging coordinates in the Master node.
Merging solution in the Master node.
Writing SU2 native restart file.
Writing Paraview ASCII volume solution file.
Writing Paraview ASCII surface solution file.
Writing the forces breakdown file.
-------------------------------------------------------------------------

History file, closed.

------------------------- Solver Postprocessing -------------------------
SU2_CFD(26422,0x7fffb7a453c0) malloc: *** error for object 0x7fd4d5912810: incorrect checksum for freed object
```

Figure 5.1: SU^2 memory corruption error snippet due to curvilinear geometry

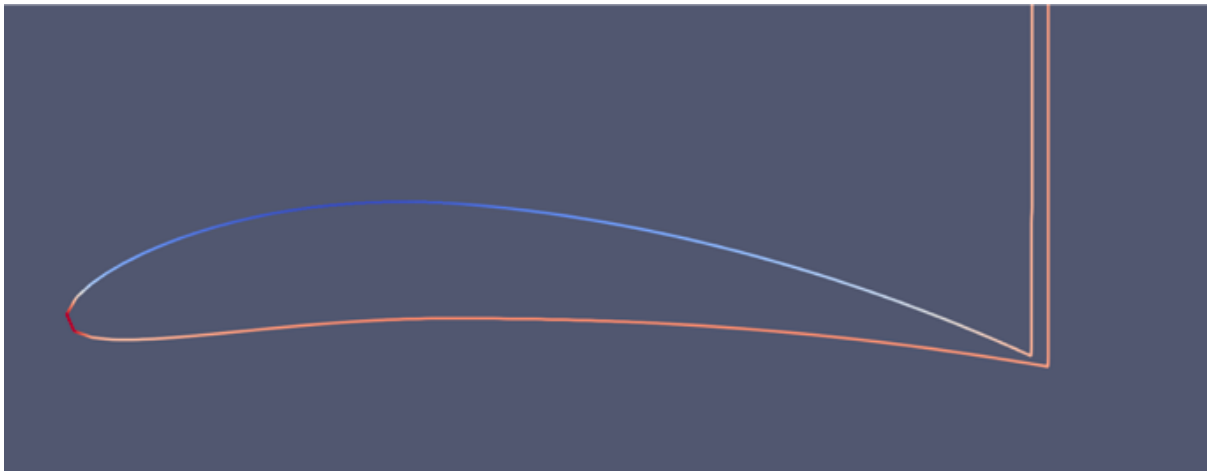

Figure 5.2: Surface flow of inverted NACA8412 airfoil colored by pressure. Evident issue occurring at the trailing edge. Different runs yielded different deformations.

While SU^2's codebase is publicly available and there are a few pages of tutorials on how to install it and run the example cases provided, there is little documentation on the code structure. The examples provided are fully configured without detailed explanation and come with solution restart files to help the solver reach the solution quickly and divergence-free. These examples are useful starting points for further work but are often very intricate and have a significant number of solver settings that vary from one another. While it can be appreciated that these are necessary to best fit with each PDE equations and cases, a significant number of times it was necessary to scan the code base on the acronym used in the configuration in order to find the full name of the routine used and from there perform research on its algorithms and uses. Alternatively, when editing example cases to perform a different simulation using the same geometry, issues or very difficult convergence were encountered as a consequence of using incorrect solver settings which were more prone to divergence with the physics changes made, such as from compressible to incompressible. Additionally, if one were to set up a case from scratch, there is no guidance on the process, but instead what seemed to be the best path was to scan the example cases for the configurations used and combine them in the attempt to allow it to solve correctly.

The feature sets currently under development in SU^2 are developed in so-called "branches" which are Git entities where the code is "forked", or deep-copied, and follows its own code changes until complete. As per common coding practice, when these features are complete, all the changes made in the branch are merged with what is commonly named "develop" branch, which is where new features are to be merged and tested together to ensure there are no clashes or issues arising from the various features. When the 'develop' branch is considered stable, it gets copied onto the "master" branch. Throughout this project it was necessary to repeatedly change the branch in order to understand which one supported which features, like FSI ramp loading in "feature_DE_AD" or checking if the discrete adjoint was failing due to being on the wrong branch by switching to "feature_disc_adj".

The SU^2 adjoint solver convergence criteria are difficult to control and most times, the only option that would work was by limiting iteration count, rather than other, residual-based methods. While the computational resources allocated to students in this project were more than sufficient, with public computers equipped with quad-core modern CPU, a shared server with 22 cores, called 'Spitfire', and an HPC cluster, it became quickly problematic to manage storage resources. User storage was limited to 8GB on the home directory, but was unregulated on Spitfire. Here many simulations were truncated halfway through solve due to an out-of-disk-space error due to many users running simulations contemporarily. The CX1 cluster could have been used but the added time required to learn how to operate on a scheduled distributed computer, and the impossibility of monitoring the simulation output meant that CX1 was not suitable for the task of running debugging, or real-time visualization of the output from the solver code.

Due to the tight limitations on storage, all pre-processing, post-processing and storing of solution was performed on a personal portable computer, accessing the computer resources through the VPN. During a short period of time working remotely, the VPN server failed which prevented running of the simulations during this period when working remotely, but was solved with the help of the University's IT department within a few weeks.

6. Project Conclusion

6.1 Reflections

This project was focused from the onset on the assessment of the discrete adjoint capabilities of SU^2 when applied to complex simulations like that of FSI. The successful realisation of this would be able to remove many of the limitations imposed on fluid-only optimisations. For example, an FSI optimisation may be based on a multi-objective around two different flow conditions, and may develop a structure that deforms favourably based on the different pressure forces of the flow conditions. Unfortunately, this was not successful, due to circumstances that were too complex to be investigated within the limited time available.

In order to perform the investigation into FSI adjoints, a case had to be selected, but due to the irrelevance of the cases provided within the context of airfoil optimisation, an airfoil case had to be generated as part of the project. A three-dimensional wing was available as part of the work of previous students at Imperial College London, but its computational cost was prohibitive, as the direct solution would have consumed up to a week of solve time. As such, time had to be allocated to get accustomed with GMSH, its scripting language, SU^2's mesh requirements for naming of physical surfaces. Additionally, due to the limited documentation available, which strictly explains how to run the sample cases, learning the SU^2's solver environment and configurations was difficult and tedious, involving tracing through the source code to understand the meaning of acronyms and discovering additional options not mapped in the global `config_template.cfg`.

Additional work was necessary to understand the structural solver and the FSI solver, as well as gain an understanding of the mathematics and application within SU^2 of the discrete adjoint. Time was also dedicated to learning and configuration of the shape optimisation routine and its various configurations. The inability to dedicate sufficient time to each of these tasks, and impossibility to skip any of them meant that no sufficient learning could be done on either work on the discrete adjoint or the FSI part of the SU^2 code to understand the issue and potentially correct it.

Overall, this project shone a light on the current state of SU^2's multiphysics capabilities, particularly to the extended applications of the solver to new problems involving FSI. Investigations into isotropic wing static and dynamic instabilities can accurately be performed within SU^2 as an addition to its list of other aviation-focused features and thermal solver.

The discrete adjoint implementation in SU^2 was benchmark and showed to be capable of matching the performance of the continuous adjoint and, potentially, exceeding it when the direct solution presents very low residuals. The shape optimisation can successfully utilise

the discrete adjoint as a replacement to the continuous adjoint as well as seemingly reaching a lower objective function, but other reasons for this were debated earlier in the report.

6.2 Future work

Further development of the structural solver would expand the information available from the adjoint. The addition of the mesh-node positional sensitivity would permit the addition of optimisation of a shape based on the morphing of its surface structure rather than by modification of its material properties. Further investigation is also necessary into the issue noted concerning the linear elastic solver for the structural beam in bending.

The FSI solver needs to complete development and merge with the ‘develop’ branch in order for its feature sets to be appropriately integrated into the configuration file, as well as the routine for merging the fluid and structural mesh needs to be made available in SU^2 natively.

An in-depth investigation into the memory corruption error that affected geometries defined with splines instead of point is deemed important as individuals who mesh on GMSH are currently forced to choose between a discretized, imperfect mesh or a complex routine of upsampling of points based on the required resolution of each airfoil, and performing a similar task in three dimensions is likely prohibitively time-consuming.

7. Bibliography

[1] M. Drela, “Development of the D8 Transport Configuration,” MIT, 2011.

[2] S. S. Desai, “Relative roles of computational fluid dynamics and wind tunnel testing in the development of aircraft,” 2003.

[3] J.-M. Bosquet, “NEW MEASUREMENT TECHNIQUES IN THE ONERA LARGE WIND TUNNELS,” in 25th International congress of the aeronautical sciences, 2006.

[4] J. Kompenhans, “Application of particle image velocimetry for the investigation of high speed flow fields,” CIMNE, Barcelona, 2002.

[5] K. L. e. al, “Aerodynamic Shape Optimization via Global Extremum Seeking,” IEEE TRANSACTIONS ON CONTROL SYSTEMS TECHNOLOGY, 2015.

[6] D. S. e. al, “Aerodynamic Shape Optimization of Supersonic Wings by Adaptive Range Multiobjective Genetic Algorithms,” Tohoku University, Department of Aeronautics and Space Engineering, Sendai, 2001.

[7] R. Baldock, “Structural optimisation in building design practice: case-studies in topology optimisation of bracing systems,” Cambridge school of Engineering, Cambridge, 2007.

[8] D. Sudholt, “Computational Complexity of Evolutionary Algorithms, Hybridizations, and Swarm Intelligence,” Dortmund, 2008.

[9] F. Montomoli, Uncertainty Quantification in Computational Fluid Dynamics and Aircraft Engines, 2015.

[10] T. H. D. F. S. P. M. Schwedes, Mesh Dependence in PDE-Constrained Optimisation, Springer, 2017.

[11] S. Weickgenannt, “Parametric-adjoint Optimization using STAR-CCM+ and CAESES,” in STAR Global Conference, 2016.

[12] U. Nilsson, “Description of adjointShapeOptimizationFoam and how to implement new objective functions,” Chalmers University of Technology, 2014.

[13] T. A. Albring, “Development of a Consistent Discrete Adjoint Solver in an Evolving Aerodynamic Design Framework,” in AIAA AVIATION Forum, 2015.

[14] Z. Lyu, “Aerodynamic Shape Optimization Investigations of the Common Research Model Wing Benchmark,” University of Michigan, 2014.

[15] G. G. e. al, “Stable loosely-coupled-type algorithm for fluid-structure interaction in blood flow,” University of Houston, Houston, 2008.

[16] H. G. M. e. al, “Partitioned strong coupling algorithms for fluid–structure interaction,” Elsevier Science Ltd, 2003.

[17] T. D. E. e. al, “SU2: An Open-Source Suite for Multiphysics Simulation and Design Read More: https://arc.aiaa.org/doi/10.2514/1.J053813,” SU2, Stanford, 2015.

[18] C. G. e. al, “Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities,” International Journal for Numerical Methods in Engineering, 2009.

[19] P. M. M. M. F. Cook, “Aerofoil RAE 2822 - Pressure Distributions, and Boundary Layer and Wake Measurements,” AGARD Report, 1979.

[20] “NACA 0012 AIRFOIL,” Airfoiltools, 2017. [Online]. Available: http://airfoiltools.com/airfoil/details?airfoil=n0012-il. [Accessed 17 07 2017].

[21] G. M. A. e. al, “Design and Stress Analysis of a General Aviation Aircraft Wing,” Texas A&M University, 2010.