Zurich University of Applied Sciences

Specialization Project 2

Institute of Mechanical Systems

Adaption of the SST turbulence model for optimization in ADflow

Author:
David Anderegg

Supervisor:
Prof. Marcello Righi
Dr. Anil Yildirim (Michigan)

July 31, 2023

Zurich University
of Applied Sciences

School of
Engineering

IMES Institute of
Mechanical Systems

Abstract

abstract

Contents

1. Introduction

High fidelity optimization using gradient based approaches have become more and more popular. This may be explained through the fact that they do not suffer from the *curse of dimensionality*. This allows them to have a much higher number of design variables (dvs) compared to alternatives such as genetic optimization approaches. The drawback is, of course, one must compute the gradients in an efficient manner. The *MACH-Aero* Framework offers a set of tools for gradient based high-fidelity optimization. Part of this Framework is a CFD solver called *ADflow* which has only one working turbulence model. This project aims to change that through the implementation of the k - ω *Shear Stress Transport (SST)* model.

1.1 ADflow

ADflow is an open-source multi-block[1] Computational Fluid Dynamics (CFD) solver. It solves the Reynolds Averaged Navier Stokes (RANS) equations to obtain the flow solution. It is developed and maintained at the *MDOLab* at the university of Michigan and was based on a CFD solver for turbo machinery called *sumb*. It has later been adopted for gradient based optimization by means of *Algorithmic Differentiation*[2] and the *adjoint method*. Most of ADflow is written in Fortran. But it is interfaced in Python. This means, the heavy lifting is done in a fast language, but the regular user has the benefits of an object-oriented high level interpreter language.

In optimization, a lot of simulations are necessary until an optimal design is found. It is also highly important to always get an objective value for each design, even if, or especially when, it is unphysical. Otherwise the optimizer does not know how bad the current design is. To cater those concerns, ADflow employs some highly efficient and robust NK[3] and ANK[4] solvers. Those can achieve machine-precision convergence, even for aircraft configurations at an angle of attack of 90°[10] [8] [18].

When the solver was still called sumb, various turbulence models such as Spalart-Allmaras, Spalart-Allmaras with Edwards Modification, k - ω Wilox, k - ω Wilox modified, k - τ, v2-f and Menter SST were implemented. The subsequent modification for optimizations changed the structure dramatically and only the SA model was carried over. But this means, a skeleton implementation of SST is available and only the structural changes need to be incorporated.

1.2 Goals

As stated above, the legacy code for SST is still available but does not really work anymore. The goal for this project is to get it working for simulation and optimization. The necessary sub-steps may be summarized as follows. Please note some in-depth knowledge is needed to understand them. But this will be explained in later sections of this report:

1. Get the current SST model running using a legacy DADI-method.
2. Modify it in such a way that it is automatically differentiate-able.
3. Actually differentiate it through *Automatic/Algorithmic Differentiation (AD)*.

[1] Overset meshes are also possible.
[2] Thats what AD stands for in ADflow.
[3] NK stands for the Newton–Krylov method.
[4] ANK is an approximated Newton–Krylov method.

4. Make sure the partial AD derivatives are correct.
5. Get the NK/ANK Solver working.
6. Test and verify the implementation
7. Get the adjoint Solver working.
8. Test and verify the total gradients.

1.3 Code contributions

Part of this project is a code contribution to ADflow and a setup of test cases, both can be found on GitHub under those links:

ADflow Pull Request	`https://github.com/mdolab/adflow/pull/301`[5]
Test cases	`https://github.com/DavidAnderegg/SST_rough_testcases`

1.4 Acknowledgment

Please note that part of the texts in this document have been refined using *chatGPT* [1].

[5] The commit at the time of writing was: `361bf3e9a2e1e4fee177becfdc94f1e48809fa2e`

2. Theoretical Fundamentals

2.1 Boundary layer

To introduce the concept of a boundary layer, envision a uniform flow moving in a single direction with a constant speed U_∞. Now, imagine placing a slender plate in this flow, aligning its long side with the direction of the flow. This configuration is commonly referred to as a *flat plate at zero incidence.* At the surface of the plate, the *no-slip condition* must be met, meaning the flow slows down until it comes to a complete stop at the surface. However, this deceleration doesn't occur in a linear manner; a significant portion of the flow maintains a uniform velocity. Only in the vicinity of the plate's surface does the flow experience a slowdown, primarily due to frictional forces. This specific area is termed the *boundary layer* or *frictional layer.* The thickness of the boundary layer, denoted as $\delta(x)$, is influenced by various factors, with its position from the leading edge being the most prominent one. In reality, there is no sharp demarcation between the uniform flow and the boundary layer. Therefore, it is often defined as the region where the flow reaches 99% of the velocity of the outer flow [14]. Refer to Figure 2.1 for a visual representation of this concept.

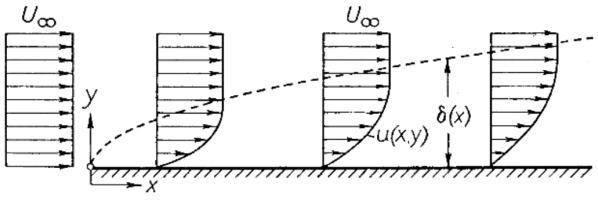

Figure 2.1: Laminar boundary layer of a flat plate at zero incidence [14].

Types of boundary layers

The flow within the boundary layer can exhibit two different characteristics: *laminar* or *turbulent.* In actuality, it starts as laminar at the leading edge, then undergoes a transition to turbulence over a specific distance, and eventually becomes fully turbulent thereafter [14], this report specifically focuses on a fully turbulent turbulence model, and as a result, the explanations of the laminar and transitional boundary layers are not further elaborated upon.

Frictional forces

As explained earlier, the flow in the boundary layer is slowed down until it becomes zero at the surface. This slowing down exerts a force in the flow direction on the surface. This force, normalized by its application area, is called *shear stress* (τ_w). To obtain a dimensionless coefficient, which may be easily compared, the shear stress is divided by the *dynamic pressure* [14]:

$$c_f = \frac{\tau_w(x)}{\frac{1}{2}\rho U_\infty^2} \tag{2.1}$$

Where ρ is the density of the fluid and c_f the skin friction coefficient.

2.1.1 Turbulent boundary layer

Upon closer examination of a turbulent boundary layer, one can discern two distinct regions. The upper region constitutes a *turbulent layer*, which is influenced indirectly by friction with the wall. In contrast, the lower region is noticeably thinner compared to the overall boundary layer. This lower region is referred to as the *viscous sublayer* or *viscous wall* layer" and is directly influenced by friction. Just like the boundary layer's broader context, there is no clear demarcation between these two regions; instead, a smooth transition can be observed.

To analyze the cross-section of the boundary layer effectively, it is helpful to introduce the concept of the dimensionless *wall distance*, denoted as y^+. In conjunction with this, we also consider the dimensionless velocity, u^+. Adopting this dimensionless system facilitates the comparison of various boundary layers from different flow conditions in a more straightforward manner. The expression for the dimensionless velocity, u^+, is as follows: [14]

$$u^+ = \frac{u}{u_\tau} \tag{2.2}$$

Whereas u is the flow velocity and u_τ the *friction velocity*. It is given by:

$$u_\tau = \sqrt{\frac{\tau_w}{\rho}} \tag{2.3}$$

As before, τ_w is the *shear stress* and ρ the *density*. The dimensionless *wall distance* y^+ is given by:

$$y^+ = \frac{y u_\tau}{\nu} \tag{2.4}$$

Whereas y is the distance to the wall and ν is the *kinematic viscosity* of the fluid.

Universal law of the wall

Theory which describes the velocity distribution of a turbulent boundary layer in fully developed flow[1] is know as the *universal law of the wall*. It defines the different regions as follows [14]:

Viscous sublayer ($y^+ < 5$) For the viscous sublayer, u^+ is given by:

$$u^+ = y^+ \tag{2.5}$$

Logarithmic overlap law ($y^+ > 30$) In the fully turbulent region at the top of the boundary layer, the turbulence stress dominates and the velocity profile varies very slowly with a logarithmic function:

$$u^+ = \frac{1}{\kappa} ln(y^+) + C^+ \tag{2.6}$$

The Karman constant κ is equal to 0.41 and C^+ equals to 5.0 for smooth walls.

Buffer layer ($5 < y^+ < 30$) The *buffer layer* is located between the viscous sublayer and the logarithmic area. It is a region where the flow transitions from one to the other. It can not be described with such an easy equation as for the other two regions.

If we plot the wall distance y^+ on a logarithmic scale and the velocity u^+ on a linear scale, the different regions are obvious. Take a look at figure 2.2.

[1] This means, the flow does not change with increasing x.

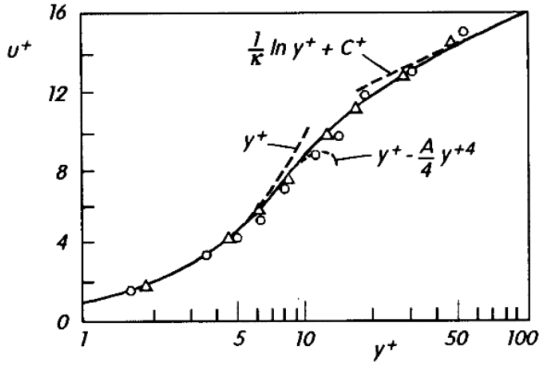

Figure 2.2: Cross-section of a fully developed turbulent boundary layer overlapped with measurements[14].

2.2 Reynold's Averaged Navier Stokes (RANS).

The *Navier-Stokes* equations establish the connection between the velocity (U), pressure (P), temperature (T), and density (ρ) of a moving fluid. These equations, represented as partial differential equations, are typically challenging to solve analytically. Consequently, numerical methods become necessary. The physical properties are functions of four variables: spatial coordinates (x, y, z), and time (t). To apply numerical methods effectively, these properties must be discretized. [12]

In flows with high Reynolds numbers, various eddies have significantly different length and time scales. Properly capturing the smallest eddies, essentially representing turbulence, would require an extremely fine mesh and time step, rendering it impractical. To address this challenge, simplifications are necessary: (1) considering a steady flow instead of unsteady, and (2) accounting for turbulence in a stochastic manner. By adopting these approaches, a single simulation (instead of one every x milliseconds) and a coarser grid become sufficient to achieve meaningful results.

Reynolds averaging

In 1895, Osborne Reynolds introduced a solution that would later be referred to as *Reynolds Averaged Navier-Stokes*. The fundamental concept involves dividing the velocity (along with other resolved physical properties) into two components: [9]

$$U_i = \bar{U}_i + u_i' \qquad P = \bar{P} + p' \tag{2.7}$$

The subscript i represents all three spatial coordinates (x, y, z). The mean velocity, denoted as $\bar{U}_i$, remains constant, while u_i' represents the fluctuating component caused by turbulence, which remains unresolved. The same notation applies to the pressure P. Substituting these expressions into the incompressible Navier-Stokes equations results in:

$$\frac{\partial \rho \bar{U}_i \bar{U}_j}{\partial x_j} = \frac{\partial \bar{P}}{\partial x_i} + \frac{\partial}{\partial x_j}\nu\left(\frac{\partial \bar{U}_i}{\partial x_j} + \frac{\partial \bar{U}_j}{\partial x_i}\right) - \frac{\partial}{\partial x_j}\rho u_i' u_j' \tag{2.8}$$

The *six* independent *Reynolds-stresses* are represented by $\rho u_i' u_j'$. It is important to note that the fluctuating component for pressure, p', cancels out and does not reappear. The Reynolds stresses can be denoted using tensor notation:

$$\rho \bar{u}_i \bar{u}_j = \rho \begin{pmatrix} u_1'^2 & u_1' u_2' & u_1' u_3' \\ u_2' u_1' & u_2'^2 & u_2' u_3' \\ u_3' u_1' & u_3' u_2' & u_3'^2 \end{pmatrix} \tag{2.9}$$

Equation 2.8 is complemented by the Reynolds-averaged mass-conservation equation:

$$\frac{\partial \rho \bar{U}_j}{\partial x_j} = 0 \tag{2.10}$$

Turbulence model

To determine the unknown Reynolds stresses, a *turbulence model* is employed. Among several approaches, the two most commonly used are *Eddy viscosity* models and *Reynolds stress transport* models. The SST model belongs to the former category, and therefore, it will be elaborated on in more detail.

Eddy viscosity models These kind of models depend on the *Boussinesq-assumption* which says that the effect of turbulence is similar to that of an increased viscosity. Thus, it introduces the *eddy viscosity* μ_t. After some equation mangling one may calculate the Reynolds stresses from the eddy viscosity as follows:

$$-\rho u_i' u_j' = \nu_t(\frac{\partial \bar{U}_i}{\partial x_j} + \frac{\partial \bar{U}_j}{\partial x_i}) - \frac{2}{3}\delta_{ij}\rho k \tag{2.11}$$

Where

$$\begin{aligned} \delta_{ij} = 0 \quad & \text{for} \quad i \neq j \\ 1 \quad & \text{for} \quad i = j \end{aligned}$$

and k is the *turbulent kinetic energy*. Thus the calculation of the six Reynolds-stresses has reduced to calculating ν_t and k. [9]

Turbulent kinetic energy is the mean *kinetic energy* per unit mass associated with eddies in turbulent flows. It is defined as half the sum of variances of the velocity components:

$$k = \frac{1}{2}\left(\overline{(u')^2} + \overline{(v')^2} + \overline{(w')^2}\right) \tag{2.12}$$

2.3 Turbulence models

The SST model is a mix between earlier models. It basically blends between the k - ϵ and k - ω turbulence models. As such, they are introduced first.

2.3.1 k - ϵ model

This model was first introduced in 1972 and solves two transport equations. [7] One for the turbulent kinetic energy (k) and one for the dissipation rate (ϵ) [2]. To apply the Boussinesq-assumption, k and ν_t are needed. The first one is solved for directly and the second one is calculated as follows:

$$\nu_t = C_\nu \frac{\rho k^2}{\epsilon} \tag{2.13}$$

Transport equations

The transport equations are as follows:

$$\frac{\partial(\rho k)}{\partial t} + \nabla \cdot (\rho U k) = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_k}\right) + \nabla k\right] + P - \rho\epsilon \tag{2.14}$$

$$\frac{\partial(\rho \epsilon)}{\partial t} + \nabla \cdot (\rho U \epsilon) = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_\epsilon}\right) + \nabla \epsilon\right] + C_1 \frac{\epsilon}{k} P - C_2 \rho \frac{\epsilon^2}{k} \tag{2.15}$$

Where P is the production due to mean velocity shear and buoyancy. The symbol ∇ represents the spatial derivatives $\partial/\partial x_i$ where i is either in x, y or z direction. The remaining constants have been derived from experiments and may change. Originally, they were defined as follows:

[2] The rate at which k is converted into thermal energy through viscosity

$$C_1 = 1.55 \qquad C_2 = 2.0 \qquad C_\nu = 0.09$$
$$\sigma_k = 1.0 \qquad \sigma_\epsilon = 1.3$$

The variant of this model is referred to as the *high-Reynolds-number* form. As its name implies, it performs poorly in predicting results near walls where the Reynolds number is low. This deficiency arises due to the blocking effects caused by the wall in the viscous sub-layer, which reduces dissipation.

Modifications for near wall flows

The original authors proposed an extension that is better suited for flows in the viscous sub-layer. It is called *low-Reynolds-number form*. The model is modified as follows:

$$\nu_t = f_\nu C_\nu \frac{\rho k^2}{\epsilon}, \qquad ... + C_1 \frac{\epsilon}{k} f_1 P - f_2 C_2 \rho \frac{\epsilon^2}{k} + .. \tag{2.16}$$

Where the modifications in red are damping functions that are defined as follows:

$$f_1 = 1 \tag{2.17}$$
$$f_2 = 1 - 0.3 exp(-Re_T^2) \tag{2.18}$$
$$f_\nu = exp\left(\frac{-3.4}{(1 + (Re_T/50))^2}\right) \tag{2.19}$$
$$\tag{2.20}$$

Where Re_T is the turbulent Reynolds number which is:

$$Re_T = \frac{\rho k^2}{\nu \epsilon} \tag{2.21}$$

The damping function f_ν tends to be 1 far from the wall because Re_T tends to be high there. Close to the wall, where Re_T is low, f_ν goes towards 0 and thus the effects of the eddy viscosity vanish. The original authors did not see an improvement for f_1 and therefore left it at 1. The function f_2 is applied to the dissipation of ϵ and consequently lowers it near the wall it.

Boundary conditions

The boundary conditions are [7]:

$$k_w = 0 \qquad \epsilon_w = 0$$
$$u_\infty \frac{dk_\infty}{dx} = -\epsilon_\infty \qquad u_\infty \frac{d\epsilon_\infty}{dx} = -C_2 f_2 \epsilon_\infty^2 / k_\infty$$

Where the subscript w stands for *wall* and ∞ for the *freestream condition*.

Weaknesses

This model works well for prediction flows outside of boundary layers. Although it has the capacity to predict the flow near the wall, inside the boundary layer, it does so poorly. If there are **adverse pressure gradients** and/or **shocks** present, the prediction inside the boundary layer is even worse. [15]

2.3.2 k - ω model

Because of the shortcomings mentioned in the section before, a better turbulence model was needed for aerodynamics and turbomachinery. This model tried to address this by modeling the specific turbulent dissipation rate ω instead of ϵ. They are related as follows:

$$\omega = \frac{\epsilon}{C_\nu k} \tag{2.22}$$

where $C_\nu = 0.09$. Plugging this into equation 2.13 leads to:

$$\nu_t = \frac{\rho k}{\omega} \tag{2.23}$$

Transport equations

The transport equations for k remains the same as in equation 2.14. But the transport equation for ϵ is replaced by one for ω. As the notation has changed slightly, the transport equation for k is also shown:

$$\frac{\partial(\rho k)}{\partial t} + \nabla \cdot (\rho U k) = \nabla \cdot [(\nu + \sigma_k \nu_t) + \nabla k] + P - \rho\epsilon \tag{2.24}$$

$$\frac{\partial(\rho \omega)}{\partial t} + \nabla \cdot (\rho U \omega) = \nabla \cdot [(\nu + \sigma_\omega \nu_t) + \nabla \omega] + \frac{\gamma}{\nu_t} P - \beta\rho\omega^2 \tag{2.25}$$

The constants for the Wilcox 1988 version[3] are as follows: [13]

$$\beta = 3/40 \qquad \gamma = 5/9$$
$$\sigma_k = 0.5 \qquad \sigma_\omega = 0.5$$

Boundary conditions

The boundary conditions for the wall are equal to the k - ϵ model. As there are many different versions, it is not quite clear what the correct values for the farfield would be. [13] suggest that the ones for the SST model should be used (see section XXX). [17]

$$k_w = 0 \qquad \epsilon_w = 0$$

Differences to k - ϵ

While both models share similarities, the k - ω model stands out by not requiring inaccurate damping functions, making it more suitable for adverse pressure gradients. Consequently, the k - ω model significantly outperforms the other model for flows inside boundary layers, leading to improved predictions in the fields of aerodynamics and turbomachinery.

Weaknesses

Unfortunately, this models solution highly depends on the freestream boundary conditions. Even small changes can lead to drastic different skin friction coefficients. This in turn may lead to different flow separation points. Unfortunately, it is not exactly clear where this dependence is coming from. [17]

2.3.3 k - ω SST model

To recap, there is a model that works well far away from walls (k - ϵ) but is not suited for flows near it. And there is a model that acts the other way around (k - ω). Additionally, both models are highly similar. Thus the Idea of the k - ω SST model is to blend both models in such a way that the ϵ formulation is used far away from walls and the ω formulation is used near walls. Figure 2.3 shows this concept.

[3] There a lot of different versions for this model

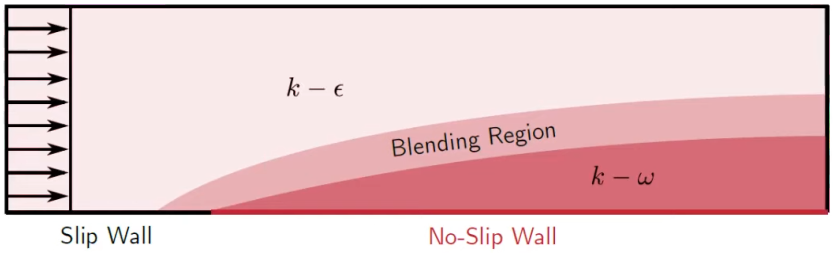

Figure 2.3: Flat plate a zero incidence with SST blending between k - ω and k - ϵ where appropriate [17].

Transport equations

It has already been established that the transport equation of k is the same for both models (see equation 2.24). When we take the transport equation for ϵ (eq. 2.25) and substitute ϵ with equation 2.22, we get:

$$\frac{\partial(\rho\omega)}{\partial t} + \nabla\cdot(\rho U\omega) = \nabla\cdot\left[\left(\nu + \frac{\nu_t}{\sigma_\omega}\right) + \nabla\omega\right] + \frac{\gamma}{\nu_t}P_k - \beta\rho\omega^2 + 2\frac{\rho\sigma_{\omega 2}}{\omega}\nabla k : \nabla\omega \tag{2.26}$$

When ignoring the additional part in red[4], then it is the same equation as was used to to transport ω. When we multiply this additional term with $(1-F_1)$, we can blend between the two models by varying F_1 from 0 (k - ϵ) to 1 (k - ω). Of course, both models have different constants and they must be blended as well:

$$\phi = F_1\phi_1 + (1-F1)\phi_2 \tag{2.27}$$

Where subscript 1 and 2 are the respective constants from each model. The constants for the standard Menter SST variant from 1994 are as follows: [13]

$$\gamma_1 = \frac{\beta_1}{C_\nu} - \frac{\sigma_{\omega 1}\kappa^2}{\sqrt{C_\nu}} \qquad \gamma_2 = \frac{\beta_2}{C_\nu} - \frac{\sigma_{\omega 2}\kappa^2}{\sqrt{C_\nu}}$$
$$\sigma_{k1} = 0.85 \qquad \sigma_{k2} = 1.0$$
$$\sigma_{\omega 1} = 0.5 \qquad \sigma_{\omega 2} = 0.856$$
$$\beta_1 = 0.075 \qquad \beta_2 = 0.0828$$
$$\kappa = 0.41$$

Blending function

The blending function is computed as follows:

$$F_1 = tanh(arg_1^4) \tag{2.28}$$

Where:

$$arg_1 = min\left[max\left(\frac{\sqrt{k}}{\beta^*\omega d}, \frac{500\nu}{d^2\omega}\right), \frac{4\rho\sigma_{\omega 2}k}{CD_{k\omega}d^2}\right] \tag{2.29}$$

Where d is the distance to the nearest wall and CD_{kw}:

$$CD_{kw} = max\left(2\rho\sigma_{\omega 2}\frac{1}{\omega}\nabla k\nabla\omega, 10^{-20}\right) \tag{2.30}$$

At this point, we have got the k - ω BST model.

[4] $\nabla k : \nabla\omega$ stands for: $\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j}$

Viscosity Limiter

It has been found through experiments that a viscosity limiter is beneficial. Thus, the computation of ν_t is modified:

$$\nu_t = \frac{a_1 \rho k}{max(a_1 \omega, \Omega F_2)} \tag{2.31}$$

Where $a_1 = 0.31$, Ω is the vorticity magnitude and F_2 is a second blending function. It is computed as follows:

$$F_2 = tanh(arg_2^2) \tag{2.32}$$

Where:

$$arg_2 = max\left(2\frac{\sqrt{k}}{C_\nu \omega d}, \frac{500\nu}{d^2, \omega}\right) \tag{2.33}$$

The closer we are to the wall, the bigger arg_2 is (as it depends on this distance). The higher arg_2 is, the bigger F_2 gets. And if F_2 gets bigger, the viscosity is limited more and more.

Conclusion

This model agrees better with experiments of mildly separated flows. This is mainly due to the viscosity limiter. Hence, it is best for external aerodynamics or simulations where separation is important. [16]

2.4 Gradient computation

When optimizing, we try to minimize the objective function $o(x)$ while making sure the constraints $c(x)$ are satisfied. The vector x stands for the *design variables (DVs)*. From an outside perspective, the objective $o(x)$ and constraints $c(x)$ are the same in the sense that we need to provide a function value an its gradients. The distinction only happens at the optimizer level. Thus, they are are called *Functions of Interest (FoI)* or $f(x) = \begin{bmatrix} o(x) & c(x) \end{bmatrix}^T$. Therefore, the gradients we are after are df/dx. To obtain it, we have a view different options available: [11]

- **Finite Differences (FD)** are the traditional approach. But they are inefficient as they scale with the number of DVs which can grow quite substantial in practical optimizations. Additionally, they are not that accurate as the lowest possible step is limited by machine precision.
- **Complex Step (CS)** is almost the same as FD. But the step is applied in the complex plane. This decouples it from the base-floating point number and thus allows an almost infinitely small step without running into machine precision problems. This means, CS is almost as accurate as it gets, but it is still inefficient.
- **Automatic/Algorithmic Differentiation (AD)** is a technique that is known widely as *back propagation* in machine learning. It basically uses the chain rule in an automated and systematic manner to compute the total derivatives from partial derivatives based on basic maths operations (e.g. Additions or multiplications). There exists a *forward* and *backwards* mode. The forward mode computes the derivative df/dx and scales with the number of design variables. Thus, it is as efficient as CS or FD. But it is as accurate as analytic derivatives are. The backwards mode computes the same derivative but scales with the number of FoIs. In practical optimizations, the number of FoIs is usually a lot lower than the number of DVs. Thus the backwards mode is more efficient and highly desirable.
- **Adjoint method** is, roughly spoken, an extension to AD. The problem with AD for the backwards mode (which we are interested the most) is, that you need to store additional memory for each math operation. This works well in machine learning because there are no iterative solvers. But this is not the case here and the memory required is just to much. The *adjoint method* bypasses this by replacing part, where an iterative solver is needed, with a linear system which needs to be

solved. So the trade is less memory footprint in exchange for more computing power. A trade that is well worth.

2.4.1 Adjoint method

The adjoint method[5] involves solving the following system (as it scales with the number of FoIs, a system per FoI needs to be solved.):

$$\psi^T = \frac{\partial f}{\partial u}\frac{\partial r}{\partial u}^{-1} \tag{2.34}$$

Where ψ^T are the adjoint vectors, u are the state variables the CFD solver solves for (e.g. pressure, velocity) and r are the residuals.

Once the adjoint vectors are obtained, the total derivative can be computed as follows:

$$\frac{df}{dx} = \frac{\partial f}{\partial x} - \psi^T \frac{\partial r}{\partial x} \tag{2.35}$$

To compute the partial derivatives, any method mentioned in the section above may be used, but is most reasonable to use AD. [11]

2.4.2 Direct mode

As said before, there also exists an 'opposite' mode to adjoint, it is called *direct mode*. First, we need to solve the direct vectors:

$$\phi = \frac{\partial r}{\partial x}\frac{\partial r}{\partial u}^{-1} \tag{2.36}$$

Then, we can use it to obtain the total derivatives:

$$\frac{df}{dx} = \frac{\partial f}{\partial x} - \phi\frac{\partial f}{\partial u} \tag{2.37}$$

This mode is less desirable as it depends on the number of DVs, but it comes in handy for verification of the adjoint method.

2.4.3 Verification

Accurate gradients are important as they accelerate the optimization or may even prevent it from converging at all. Thus, the partial and total derivatives need to be verified. As the total derivatives depend on the partial ones, it makes sense to verify them first:

1. **Compare forward derivatives against FD.** As FD is not as accurate, it is not enough, but it is easy to implement (no modifications needed to the code) and shows that we are in the right bulk part.
2. **Compare forward derivatives against CS.** This is absolutely needed to be sure. But it requires modification of the code and thus may introduce errors.
3. **Dot product test** Once the forward routines are verified, we can apply the dot product test (see below) to verify the backwards routines as well.

Once the partial derivatives are verified, we can try to solve the adjoint system. If this is done, the total derivatives may be verified (1.) against FD and then (2.) against CS.

[5] Analogous to AD, there exists also a forward mode called *direct method*

Dot product test

Sometimes, it is only possible to verify the forward mode against CS, but we are more interested in the other one. In such a situation, the *dot-product test* comes in handy as it provides a relation ship between both modes:

$$\psi_j^T \frac{\partial r}{\partial x_i} = \frac{\partial f_j}{\partial u} \phi i \tag{2.38}$$

Where i and j represent the column and row vectors respectively. When both sides of the equation match to a tolerance that is slightly higher than machine precision, we can be confident that the reverse (adjoint) method has been implemented correctly.[11]

2.5 Newtons method for residual equations

Usually, the RANS equations are solved using a "dumb" algorithm such as *jacobi iteration*. It is used to solve a linear system such as:

$$Au = b \tag{2.39}$$

Where u are the state variables (such as pressure and velocity). b are the sources, and A is the state matrix. As this is an iterative approach, the solutions gets better and better the longer the algorithm runs. This means, we need an criteria to stop. For this, we use the residual formulation:

$$r(u_n) = b - Au_n \tag{2.40}$$

Where n represents the value of the current iteration. Once the residual value reaches a threshold, we stop.

When looking at the partials that are needed for the adjoint method, we realize that we already compute the partials $\partial r / \partial u = r'(u)$. This comes in handy as it can be used in Newtons method for the solution of equation 2.39:

$$u_{n+1} = u_n - \Delta u_n \tag{2.41}$$

Where Δu_n is the net update or step. In the multivariate newtons method, it is obtained by solving the following linear system:

$$\frac{\partial r}{\partial u} \Delta u_n = -r(u_n) \tag{2.42}$$

Newtons method does converge faster than the before mentioned "dumb" approaches. But it is usually not implemented as the effort to obtain $r'(u)$ is not considered worth the payoff. But we are interested in gradient computation for optimizations and thus need the partials anyways. So it makes sense to use Newtons method "free of charge". This is exactly what ADflow does with its *Newton-Krylov (NK)* and *Approximate Newton-Krylov (ANK)* solvers. [11]

2.6 Grid Convergence

When discretizing a partial differential equation and solving it numerically, an error is introduced. It may be decreased through a finer mesh or a higher order method. To demonstrate that the method approaches the exact solution, finder and finer grids are used. This process is called a *grid refinement study* or *mesh convergence*. For a given grid, the grid spacing is:

$$h = N^{-1/d} \tag{2.43}$$

Where N is the number of cells and d is the dimension of the problem. For ADflow, the expected rate of convergence is $p = 2$. But in reality, this might not be the case. For three grids, the actual rate may be calculated as follows:

$$\hat{p} = ln(\frac{f_{L2} - f_{L1}}{f_{L1} - f_{L0}})/ln(r) \tag{2.44}$$

Where f is the function of interest (e.g. c_d) and the subscript tells the grid used. $L0$ is the finest grid and $L2$ the coarsest. The parameter r is the grid refinement ratio .[6]

3. Methods

3.1 Introductory thoughts

ADflow started its life in the early 2000s and was called *Standford University Multiblock (sumb)*. It was intended for turbomachinery but was extended for optimization later on. This extension required quite a substantial change to the solvers structure which rendered multiple features non working. The SST turbulence model is such a part that once worked but does not anymore.

ADflow is written in FORTRAN, but has a python wrapper. This means, the heavy lifting is done in a fast, compiled language, but the user has the benefits of an interpreted, object oriented programming environment. As explained in section 2.4, the adjoint methods need partial derivatives that are (in ADflow) obtained through means of automatic/algorithmic differentiation (AD). For this, a tool called *tapenade* [1] is used. It automatically differentiates FORTRAN source code.

Please note it was not possible to properly cite all the ADflow specific information given in this section. Most of it comes from talks with one of the supervisors and developer of ADflow *Dr. Anil Yildirim*. The remaining part comes from reading the source code.

3.1.1 Terms

The following lines explain some concepts that might not be known.

Flow variables Those are the variables that the solver solves for. The state variables of the turbulence model are excluded. Here, this means: *velocity (x, y, z)*, *density* and *energy*.

Turbulence variables The variables the solver solves for for the turbulence. For SST, this is k and ω.

Coupled turbulence When the turbulence variables are solved in a coupled manner, we have only one system of equations for all variables. When they are solved in a decoupled manner, we have two different systems: one for the *flow variables* and one for the *turbulence variables*. As everything is related to each other, solving a decoupled system slows down convergence, but it increases robustness.

3.1.2 Flow Solvers

Before diving deep into the changes that made SST run, we need to understand how ADflow solves the RANS equations and what it needs for that.

For that, ADflow has three different solvers available: *multigrid (MG)*, *Newton-Krylov (NK)* and *Approximate Newton-Krylov (ANK)*. It is possible to switch between the different solvers during a solution run. This allows to use each solver when it is most efficient. An example run might look like: initiate the simulation using multigrid, once a certain level of convergence is reached, engage the ANK solver and finally converge the last couple order of magnitudes using the NK Solver. [2]

Multigrid is the baseline solver that was implemented first. It uses either the *Runge-Kutta (RK)*, or the *Diagonalized Diagonally-Dominant Alternating Direction Implicit (D3ADI)* algorithm as a smoother.

[1] `http://www-tapenade.inria.fr:8080/tapenade/index.jsp`

[2] Please note that the ANK solver by itself is sufficient as a startup strategy and MG is not necessarily needed.

The flow and turbulence model is solved in a decoupled manner and using the *Diagonalized Alternating Direction Implicit (DADI)* method.

The Newton-Krylov solver solves the nonlinear system of governing equations by means of Newton's method (sec. 2.5). To solve the linear system at each step, the GMRES[3] algorithm is used. The turbulence variables are solved in a coupled manner and thus no other solvers are needed. This method is equivalent to using Euler's method with an infinite time step. It is most efficient when the solution is already at the final stages of convergence. If it is used in the early stages, it most likely stalls.

The Approximate Newton-Krylov is similar to the NK solver in that it also uses Newton's method. But its time step is adjustable. At the beginning of the run, it is quite low. As the solution accuracy increases, the timestep is increased as well. This increases robustness early on but also accelerates convergence later that would otherwise slow down drastically. The solver itself is subdivided into three different sub-solvers: *First order ANK (ANK)*, *Second Order ANK (SANK)* and *Coupled ANK (CANK)*. Please note, the combination of both *Coupled Second-Order ANK (CSANK)* is also possible.

In its base configuration **ANK** uses a first-order routine for the residual Jacobian where as the second order formulation **SANK** is as accurate as ADflows discretization is. The idea of this solver is to solve only as much as needed. In that context, it makes sense to use the first-order formulation early on when the solution is far from convergence. This not only saves computing power but also increases convergence because high frequency oscillations may not be captured. Once a user defined level of convergence is reached, the solver switches to an exact Jacobian formulation (SANK).

In these first two stages, the turbulence model is solved in a decoupled manner using either a second, turbulence specifi ANK solver (from now on called *ANK-turb*) or a legacy *Diagonalized Alternating Direction Implicit (DADI)* method. Splitting it up is beneficial early on as it increases robustness. Once again, when the residual norm reaches a user-defined level, the coupled mode is enganged. In that state, only one ANK solver remains which solves the flow and turbulence variables simultaneously. This helps to improve the convergence in the later stages. [2]

The ANK solver is made even more robust by introducing the concept of a *physicality check*. When looking at section 2.5, we see that we need to solve a linear system to obtain the next newton step. The solution of this system does not only depend on the accuracy of our residual computation but also on how accurate we solve the linear system. This means, a step we obtain might actually lead to a divergence of the solution. To prevent this, the ANK solver performs some checks to make sure the current step improves the solution. If this is not the case, it tries to apply only part of the step [4].

3.1.3 Adjoint Solver and total derivatives

When computing the total derivatives, it is important to realize that the adjoint solver is not the whole story. Take a look at figure 3.1. It shows the flow of data for a simple airfoil optimization example.

[3] This stands for *generalized minimal residual method.*
[4] This concept is known as backtracking.

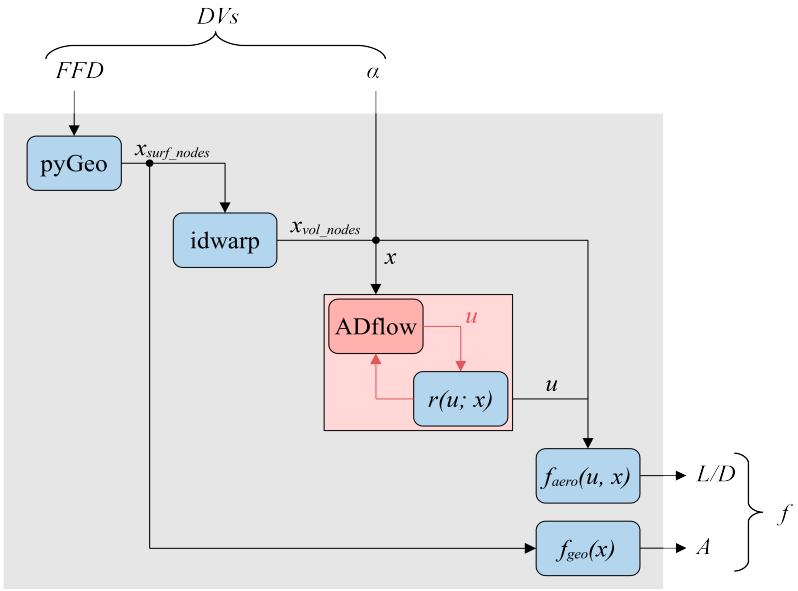

Figure 3.1: Gradient computation dataflow in the MACH-framework. [3]

At the top left, we have the design variables (DVs) and at the bottom right are the functions of interest (FoI). We have two types of design variables: *aerodynamic* (such as angle of attack) and *geometric ones*. The geometry is parameterized using the *Free Form Deformation (FFD)* approach. A similar picture emerges for the FoIs, the area (A) of the airfoil does only depend on the geometry, not the flow solution. But the lift to drag ratio (L/D) does depend on the aerodynamic solution.

Now, lets look at the flow of data through the CFD solver. First, the geometric DVs flow into a package called *pyGeo*. It maps the FFD variables to the surface mesh. This surface mesh is then used to warp the total volume mesh in *idwarp*. The deformed volume mesh is then feed into ADflow where the flow solution is computed. Here, the aerodynamic DVs start to play a role. After that, the flow variables are integrated as desired and the final function of interest (in this case L/D) is assembled. As said before, the area (A) does not depend on the flow solution and is directly computed using only the surface mesh.

If you reverse the flow of information and exchange "CFD solver" with "adjoint solver", you know how ADflow computes total derivatives for optimization. If you would like to learn more about it, take a look at [3] or [11].

3.1.4 Residual derivatives

Almost all solvers mentioned need the derivative of the residuals with respect to state variables: $r'(u) = \partial r/\partial u$. It is possible to compute it using AD or FD. If it is done through AD, it is called *automatic differentiated pre-conditioner (ADPC)*, otherwise its called *FDPC*. Counter-intuitively, the FD pre-conditioner is more efficient than the AD one. But the accuracy of FD is worse.

It is important to realize that this Jacobian is mostly zero. This is because a certain cell only depends on its neighboring cells (and not the whole domain). This realization allows to use the concept of coloring which is basically exploiting this non-dependence to compute most of the derivatives in parallel. [11]

3.1.5 Initial state of SST

When sumb was initially developed, multiple different turbulence models were implemented such as Spalart-Allmaras (SA) or SST. When the code was overhauled for optimization, only the SA model was carried over and differentiated. At that point SST would throw NaNs[5] and crash. Lately, this was fixed to a point where the DADI turbulence solver would work [6]. To summarize, before this project started, the code for SST was there and a solution could be obtained using ANK/SANK and the decoupled DADI

[5] Not a Number
[6] See pull request: `https://github.com/mdolab/adflow/pull/107`

turbulence solver. But nothing was differentiated which means, the adjoint, NK, CANK and ANK-turb solvers were not usable.

3.2 Needed changes

ADflow may be run in parallel where the computation is divided over multiple cpus and computers. It does this by splitting up the computational domain into blocks. These blocks may live on different cpus or computers. As those blocks do not live in isolation and do depended on each other, adjacent blocks need to exchange information. ADflow uses the *halo cell* approach for this. The idea is to have imaginary ghost (halo) cells around each block. Then they are filled with the values from the adjacent blocks. Figure 3.2 shows this idea. ADflow uses a second order discretizations for some terms and thus needs two layers of halo cells.

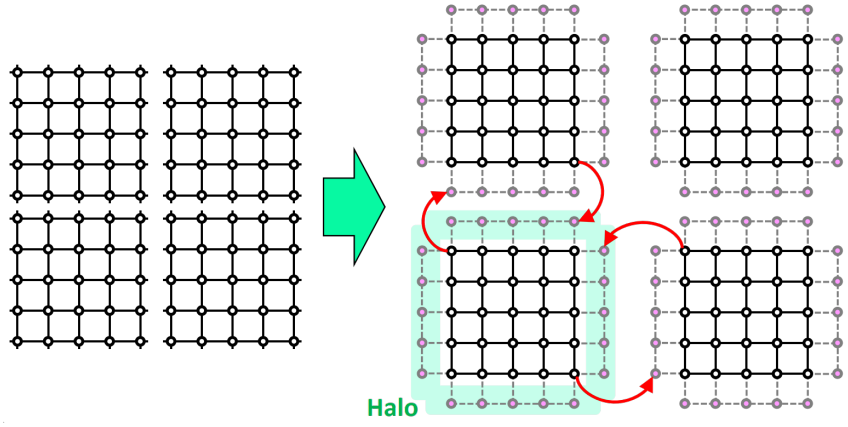

Figure 3.2: A block split in 4 (left) and its corresponding halo cells (right) [5].

3.2.1 Halo exchange and AD

If one does not care about AD, this halo exchange is straight forward. But we do, and as such, some things need to be considered. The exchanges is performed using the *Message Passing Interface (MPI)*. Unfortunately, tapenade can not handle MPI calls. To make this work, the differentiated code is divided into parts: The **math heavy** part is differentiated using tapenade and the **remaining part** is hand-differentiated. This is feasible because the hand-differentiated part consists mostly of calling the AD part and performing communication.

The turbulence model is considered a math heavy part that is automatically differentiated using tapenade. For SA, this was straight forward as there is no communication going on. But SST has one special case: The blending function $\mathbf{F_1}$ (equation 2.28). For reasons that are explained later, its value is required in halo cells. It is important to realize that halo cells are basically clones of other compute cells in other blocks. This has been exploited for F_1 by means of only computing it once and performing a halo exchange.

Exchanging values is absolutely detrimental and ADflow has routines and places where it is performed. But those routines are chosen carefully to not interfere with AD. The problem with F_1 is the fact, that it is not a regular flow or turbulence variable and thus, it has no such official routine for exchange. In the legacy implementation of SST, it was just exchanged in the model itself. But we cant do this anymore and thus the question is: Why was it done this way and how do we get rid of this intermediate communication?

The first part of the questions might be explained by fact that sumb was initially developed in the early 2000's. Back then, computing power was more expensive and computing the same F_1 value on different blocks was wasteful. The cost of communication was negligible in comparison. This did not really change, but computing power became a lot cheaper. So if it makes AD easier, it is a good trade to get rid of the communication.

3.3 Changes to wall distance

We settled on the idea of computing F_1 in the halo cells instead of communicating it across. To do this, we need to take a closer look at F_1. It depends on arg_1 (eq. 2.29) which also depends on CD_{kw} (eq. 2.30). This is a bit problematic as it requires the derivatives $\nabla k \nabla \omega$. These derivatives are discretized and thus even more halo cells are needed to compute the halo cells. Luckily, ADflow employs only a first-order discretization for the turbulence model. Thus, the second layer of halo cells is enough to compute CD_{kw} in the first halo layer.

On further examination, we realize that arg_1 also depends on the distance to the nearest wall. Unfortunately, in ADflow, the distance to the nearest wall is not assigned nor exchanged for halo cells . Therefor, this is the first thing that needs to be changed.

3.3.1 Wall distance computation

Before explaining the changes, one must briefly understand how the wall distance computation is done in ADflow. [7] It is important to realize that this is no easy task due to the block splitting. When computing the distance to the nearest wall in a cell on processor X, it is possible that the nearest wall lives on processor Y.

To make a long story short, ADflow first determines which surface cell includes the closets point for the current cell. This information is exchanged on initialization and assumed to not change[8]. After a mesh has been deformed, a function named `updateWallDistancesQuickly()` is called. It actually computes the distance to the earlier determined nearest surface cell.

3.3.2 Halo exchange

Now, we need to quickly talk about the halo exchange routine. ADflow has two options: `whalo1(...)` and `whalo2(...)`. The only difference between those functions is that the first one only exchanges the halos in the first layer and the second one exchanges all layers. For brevity, the arguments were omitted. But they are just boolean flags that determine what variables need to be exchanged (e.g. flow variables). We need the wall distance on both halo layers and thus the function `whalo2(...)` was extended.

This extension has been straight forward. Please note that some special care for the hand-differentiated part was needed. This is to make sure that the derivatives are aggregated correctly in the AD part of the code. Unfortunately (see section 4.4), there seems to be a bug present. Because of that, the author is not confident on how it is done best and does not elaborate further.

3.3.3 Bringing it together

The easy approach would be to simply call `whalo2(...)` at the end of `updateWallDistancesQuickly()`. This is not possible as this function is AD-ed and tapenade can not handle MPI-calls. So the next obvious choice is to call `whalo2(...)` manually every time `updateWallDistancesQuickly()` was executed. When initializing, the distance is computed before the communication part is initialized and thus crashes. The final solution has been to introduce a boolean flag: `exchangeWallDistanceHalos`. If it is true, the normal halo exchange-calls perform also the wall distance exchange. This works fine, but is a bit of a hack. The author believes it is possible to get rid of that flag once he understands the code better.

3.4 Algorithmic/Automatic Differentiation

Once the distance to the nearest wall was available in halo cells, the communication in F_1 has been removed. This paved the way to AD the whole SST. Before explaining those changes, one must know that ADflow employs three different versions of differentiated code: *forwards*, *backwards* and *backwards_fast*.

The first two modes are the regular AD approaches. The *backwards_fast* routine is a striped down version of the normal *backwards* routine which is done for performance reasons. Table 3.1 gives an overview on where what routine may be used. If multiple options are available, only one is enough.

[7] The interested reader may take a look at report [4] where it is explained in more detail.
[8] This is only partially true in the context of deforming meshes.

	forwards	backwards	backwards_fast
Assemble $\partial r/\partial u$	X		X
Compute total derivatives		X	
ADPC for ANK	X		
ADPC for NK	X		

Table 3.1: When what AD routines are used in ADflow.

It becomes clear that the forwards routine is the most important one. It allows to use almost all features except for total derivative computation. But once this is implemented, the reverse routines are not much more effort. The reverse_fast routines are great to speed up the computation but are not needed for a first running prototype.

3.4.1 Implementation

In order to ad the SST code, the following procedure has been followed:

1. Split the whole code in two parts: part (a) computes the residuals and part (b) solves SST in decoupled manner using DADI.
2. Part (b) stays as it is but needs to be called in the right spot.
3. Part (a) is split up even more: (I) compute kw_{cd}, (II) compute F_1, (III) compute the production terms, (IV) compute the source terms, (V) compute the advection term, (VI) compute the unsteady term, (VII) compute the viscous terms and finally (VIII) scale the residuals.
4. AD the sub-parts I through VIII using the forward routines.
5. Verify the forwards routines (see sec. 4.3).
6. AD the sub-parts I through VIII using the backwards routines.
7. Verify the backwards routines (see sec. 4.3).
8. AD the sub-parts I through VIII using the backwards_fast routines.
9. Verify the backwards_fast routines (see sec. 4.3).

When looking at the routine that computes F_1 (II), one has to take special care: To compute F_1, we need the distance to the nearest wall. This has been fixed for regular halo cells and works as intended, but this distance is not defined for halo cells that lie behind a boundary condition (like a wall). The legacy implementation solves this as follows:

1. allocate F_1 on all halo cells (including boundary conditions (BC))
2. compute F_1 on all cells (including BCs).
3. Loop over halos that lie behind a BC and simply copy the F_1 value from the nearest compute cell. This effectively is a *Neumann boundary condition*.

This procedure works fine for the regular residual evaluation. It also works for the forward AD mode. But when looking at the reverse mode, one has to revers procedure mentioned: first we apply the Neumann BC (3), and then (2) we overwrite F_1 with garbage that depends on an un-initialized wall distance. To prevent this, we must take special care to not loop over BC halos in step (2).

3.5 Verification

This section explains how SST and its derivatives were verified.

3.5.1 Testcases for robustness

To make sure, SST and the various solvers work under different circumstances, two testcases were set up. They are also used to show that the modifications made in this project did not change the legacy SST implementation that was available at the beginning.

NACA 0012

This first testcase is a NACA0012 airfoil under a low angle of attack and mach number. The mesh is a bit to coarse to obtain a physical solution. But this has the advantage that the solution is obtained faster. Which allows it to be used for debugging. Table 3.2 lists the flow conditions and table 3.3 lists the mesh parameters.

Parameter	Value
Angle of Attack	3°
Reynolds number	$5e6$
Mach number	0.3
Temperature	$288°K$

Table 3.2: Flow conditions for the NACA0012 testcase.

Parameter	Value
Chord length	1
Farfield distance	100
dimensional wall distance	$3e-6$
Growth ratio	~ 1.05
Points on airfoil surface	245
Points normal to flow direction	129
Number of compute cells	$31'232$

Table 3.3: Mesh parameters for the NACA0012 testcase.

RAE 2822

This setup uses the supercritical RAE 2822 airfoil. As such, it operates in the transsonic regime which is dominated by mach effects and shocks. Table 3.5 lists the mesh parameters and tab 3.4 the flow conditions. Similar to the NACA testcase, the number of compute cells is to low to obtain physical correct results.

Parameter	Value
Angle of Attack	2.92°
Reynolds number	$6.5e6$
Mach number	0.725
Temperature	$288°K$

Table 3.4: Flow conditions for RAE2822 testcase.

Parameter	Value
Chord length	1
Farfield distance	100
dimensional wall distance	$3e-6$
Growth ratio	~ 1.05
Points on airfoil surface	129
Points normal to flow direction	129
Number of compute cells	16′384

Table 3.5: Mesh parameters for RAE2822 testcase.

3.5.2 Testcases for accuracy

The following testcases were primarily setup to make sure SST is implemented correctly. Of course, they also provide a way to test the solvers under different circumstances.

Flatplate

The flatplate at zero incidence is a classic testcase for basic flow experiments and CFD solver verification. The setup and meshes were provided by a website called *Turbulence Modeling Resource (TMR)* which is maintained by NASA. Figure 3.3 shows the flow setup and table 3.6 list the number of cells per grid-level. All grids were obtained from that TMR website.

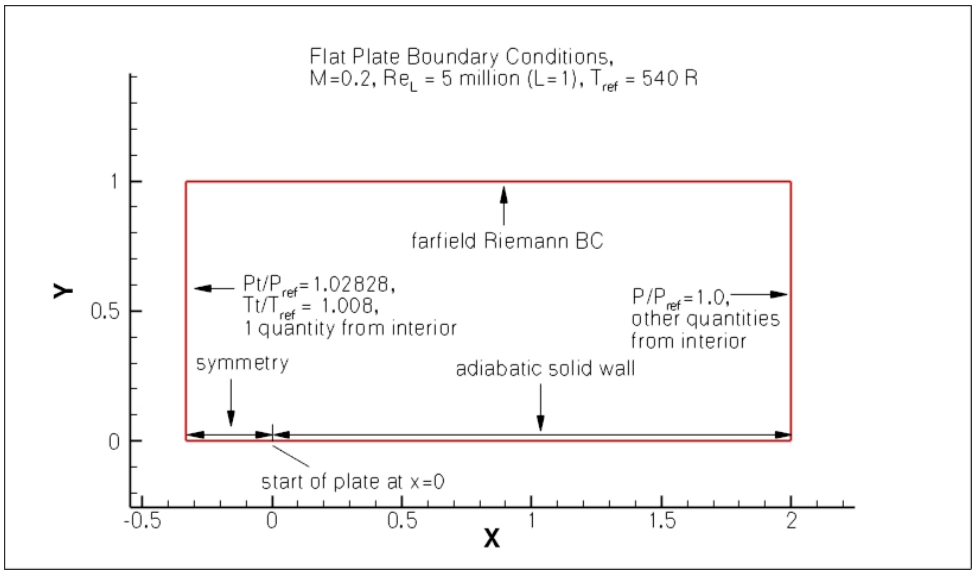

Figure 3.3: Boundary conditions and test case overview. [13]

Identifier	# of nodes	# of cells
L4	1’800	816
L3	6’860	3’264
L2	26’772	13’056
L1	105’764	52’224
L0	420’420	208’896

Table 3.6: Mesh sizes used for the flatplate testcases.

2D bump

This is also a classic testcase and as such is also provided by the TMR website. It is more involved and harder to simulate as it contains an adverse pressure gradient after the bump. Figure 3.4 shows the boundary conditions, figure 3.5 gives a close up of the bump and tab. 3.7 lists the mesh sizes.

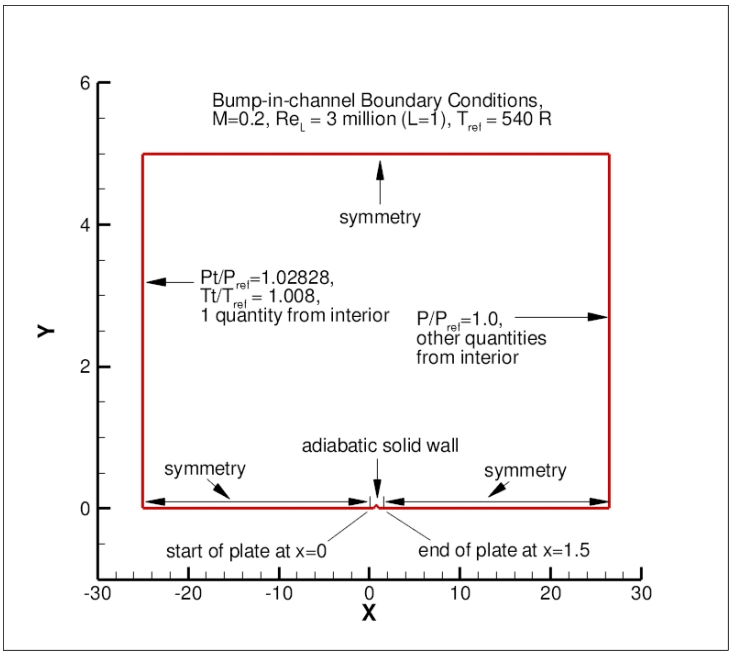

Figure 3.4: 2D bump test case overview. [13]

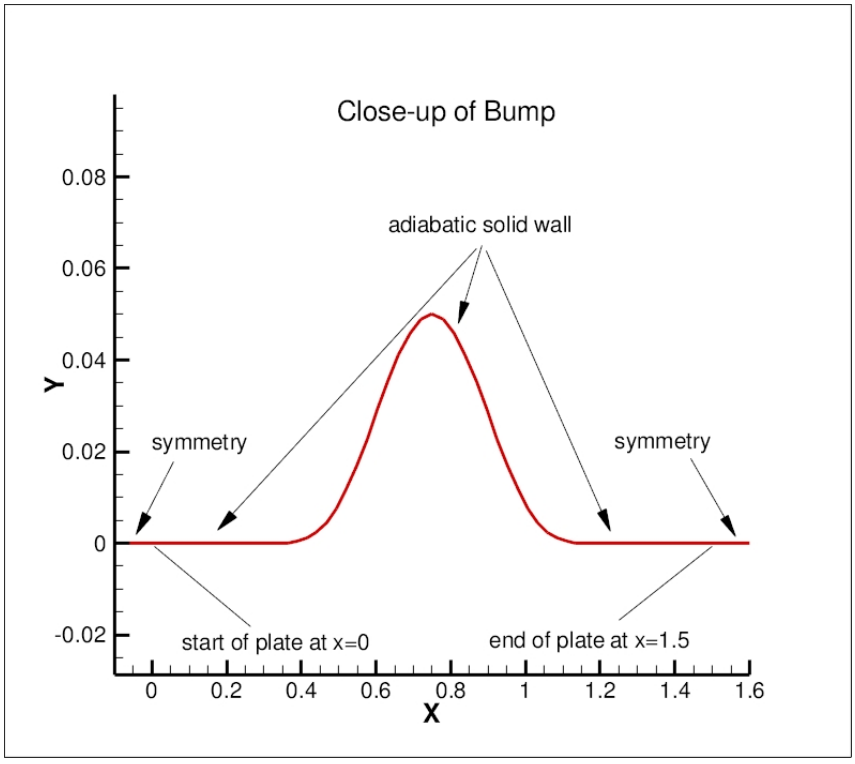

Figure 3.5: Close up of 2D bump. [13]

Identifier	# of nodes	# of cells
L4	7'462	3'520
L3	28'998	14'080
L2	114'310	56'320
L1	453'894	225'280
L0	1'808'902	901'120

Table 3.7: Mesh sizes used for the 2D bump testcase.

3.5.3 Testcases for derivatives

3D wing

This is the only 3D test employed. It makes sure the changes also work in the third dimension, but it is primarily intended for gradient verification. It is quite coarse and thus not physical. Figure 3.6 shows the surface mesh with its *Free Form Deformation (FFD)* points around. They are used as parametrisation for various geometric design variables. Table 3.9 lists the flow conditions and table 3.8 shows the design variables available.

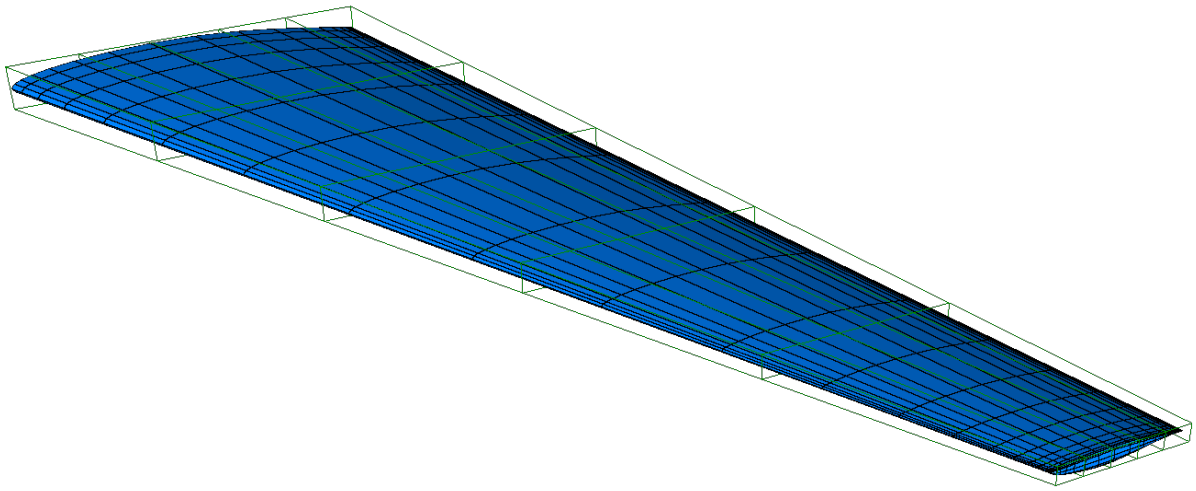

Figure 3.6: Surface mesh (blue) and FFD points (green) for the automated test setup. [4]

Name	Type	Comment
shape	geometric	Local pertubations of FFD points.
twist	geometric	Twisting of wing at each FFD-section.
span	geometric	Length of the wing.
alpha	aerodynamic	Angle of attack.
beta	aerodynamic	Slip angle.
mach	aerodynamic	Mach number.
P	aerodynamic	Pressure.
T	aerodynamic	Temperature.
xRef	aerodynamic	X location of reference point for moment calculation
yRef	aerodynamic	Y location of reference point for moment calculation
zRef	aerodynamic	Z location of reference point for moment calculation

Table 3.8: Design variables for the 3D wing testcase.

Parameter	Value
Specific gas constant	$287.87 J/(kgK)$
Pressure	$200 hPa$
Temperature	$220°K$
Alpha	1.8°
Mach	0.8
Mesh node count	$30'375$
Mesh cell count	$24'192$

Table 3.9: Flow conditions for the automated test setup.

3.5.4 Partial derivatives

To verify any derivative in this report, the before mentioned 3D wing testcase is used. ADflow computes the following partial derivatives:

$$\frac{\partial r}{\partial u} \qquad \frac{\partial f}{\partial u} \qquad \frac{\partial F}{\partial u} \tag{3.1}$$

$$\frac{\partial r}{\partial x_{geo}} \qquad \frac{\partial f}{\partial x_{geo}} \qquad \frac{\partial F}{\partial x_{geo}} \tag{3.2}$$

$$\frac{\partial r}{\partial x_{aero}} \qquad \frac{\partial f}{\partial x_{aero}} \qquad \frac{\partial F}{\partial x_{aero}} \tag{3.3}$$

$$\tag{3.4}$$

Where r are the residuals, u the state variables, f the functions of interest and F the forces on the nodes of the mesh. x are the geometric or aerodynamic design variables (see sec. 3.1.3). All those partial derivatives are obtained using either forward or backwards AD mode.

As explained in sec. 3.4, the *backwards_fast* mode is a subset and only provides:

$$\frac{\partial r}{\partial u} \tag{3.5}$$

Verification

To verify the various derivatives, the following procedure is followed:

1. Verify forward partials against finite differences.
2. Verify the forward partials against complex step.
3. Verify the backwards partials against the forward ones using the dot-product test.
4. Verify the backwards_fast partials against the regular backwards partial derivatives.

3.5.5 Total derivatives

Verifying the total derivatives is more straight forward using the 3D wing testcase. First, we obtain accurate total derivatives using complex step. Then we compute them using the adjoint approach. Please note that there are two ways to assemble the state residual matrix ($\partial r/\partial u$): either by using the forward AD or backwards_fast AD mode. Finally, we need to compare all derivatives and make sure they match to a reasonable tolerance.

3.5.6 Regression tests

ADflow uses regression tests to make sure new features do not break existing ones. Those test usually use the 3D wing testcase under the hood and have been extended for SST. Table 3.10 lists what tests have been extended and what they are used for.

Name	Purpose
`test_jacVecProdFWD.py`	Makes sure the partial derivatives agree to FD and CS. It basically performs step (1) and (2) from the previous section.
`test_functionals.py`	This tests a lot of different things. But we are mostly interested in the dot-product test that makes sure forward AD is consistent with backwards AD. It is basically step (3).
`test_jacVecProdBWDFast.py`	This test makes sure the backwards_fast AD routines agree with the backwards AD routines.
`test_adjoint.py`	Makes sure the total derivatives agree with derivatives obtained using complex step.

Table 3.10: Automated test setup.

4. Results

4.1 Solver Convergence

NACA0012

Figure 4.1 shows the solver convergence for the NACA0012 testcase. The baseline implementation (marked as *base*) and the final state when writing this report (marked with *modified*) is shown. Both states were run once with 1 and 6 cpus.

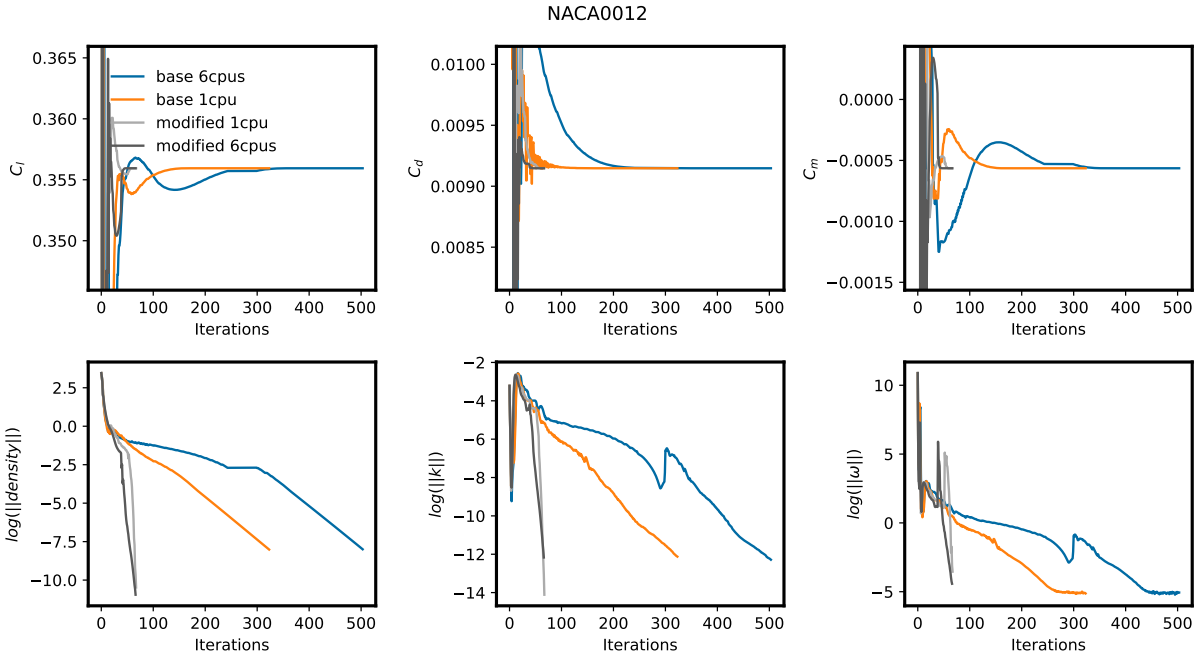

Figure 4.1: Convergence history for NACA0012 testcase.

Lets take a look at the baseline implementation first. It was obtained using the ANK solver and a decoupled DADI solver for the turbulence model. For the turbulence model, a total of 20 sub iterations were used. When looking at the function values (top row), it can be seen that 1 and 6 cpus reach the same value. But 6 cpus need more iterations. It is not quite clear what causes this, but it is a known phenomenon for low cpu counts and vanishes when this number is increased. Production runs usually require tens or even hundrets of cpus and thus this is not considered detrimental.

Now, lets look at the modified implementation. Here, SST was differentiated and the turbulence ANK and fully coupled CANK solvers are available. Annectdotal evidence suggest SST is highly non-linear. This is especially true for the inital stages of convergence. Due to this[1], the turbulence DADI solver is way more efficient early on. Thus, at the beginning, the regular ANK solver with decoupled DADI was used. But once a relative convergence of 1e-6 is reached, the second order coupled ANK (CSANK) is engaged. Once it gets traction, it exhibits almost Newton-like convergence. The number of cpus does not really affect the number of iterations needed. It is also obvious that the modified version approaches

[1] The author believes the ANK solver does some finite-differencing for some terms under the hood.

the same function values as the baseline implementation.

RAE2822

Figure 4.2 shows a similar convergence plot for the RAE2822 testcase. Once again, the baseline and modified version with each 1 and 6 cpus is plotted. It is important to note that this case is somewhat hard as it lies in the transsonic regimes where shocks appear. But at the same time, it is even coarser than the NACA case which makes it hard to resolve the shocks properly.

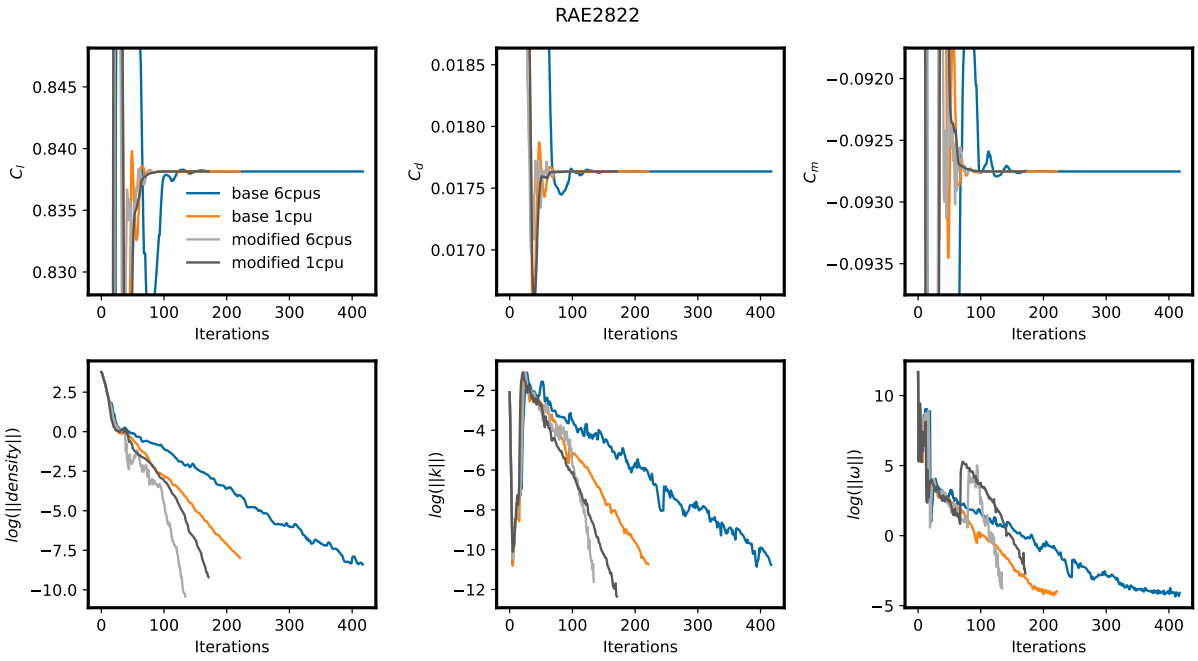

Figure 4.2: Convergence history for RAE2822 testcase.

First, lets glance at the baseline. This has also been obtained using the ANK solver for the flow variables and the DADI solver for the decoupled turbulence variables. A similar pattern to the NACA testcase appears: 1 cpu takes only half the iterations of what 6 cpus need. But, the converged values are the same.

When looking at the general line pattern and comparing it to the NACA testcase, it appears to be more 'wiggly'. The author believes this is due to SST being highly non-linear which is exaggerated by the appearances of transsonic shocks. This leads to a high sensitivity to the CFL number. The ANK solver uses a CFL-ramping algorithm that increases the CFL number steadily the closer to convergence it is. This ramping algorithm is not dumb and uses a smart formulation that also reduces the CFL number when the ANK solver has problems solving the linear system for the current step. The author believes herein lies the problem: the high non-linearity of SST and the fact that this ramping-algorithm is tuned for SA. This leads to a coupling where the ANK solver would increase the CFL number to much and start diverging. Once this is detected, the CFL number is lowered once again an convergence continues.

When looking at the modified version, a similar picture to NACA emerges. The strategy was the same, first use ANK with DADI and once a relative convergence of 1e-6 is reached, the CSANK solver is enganged which shows almost Newton-type convergence. Although the contrast is not as big. But it also has to be noted that that the before mentioned CFL dependence played a role here and some parameters had to be clipped to increase robustness at the cost of convergence.

4.2 Grid Convergence

4.2.1 Flatplate

Unfortunately, it was not possible to obtain a solution for the flat plate testcase on all grids. Thus this was not deemed worthy and is not shown here. The author is confident that a solution can be obtained but was unable to do so due to time constraints.

4.2.2 2D bump

Figure 4.3 shows the grid convergence for the 2d bump testcase compared to data from the *CFL3D* and *FUN3D* CFD solvers. At first glace, ADflow seems to be in the right bulk part but does not completely agree with the reference. This difference may be explained through slightly different model formulations and production terms. Usually, a formulation know as *vorticity* is used. The author tried to converge this testcase using it but was unable to do so. So the results shown where obtained using the *strain* turbulence production formulation. It is also not completely clear what version of SST was implemented in ADflow which might skew the results even more. The reference data was obtained using the *SSTm* formulation.

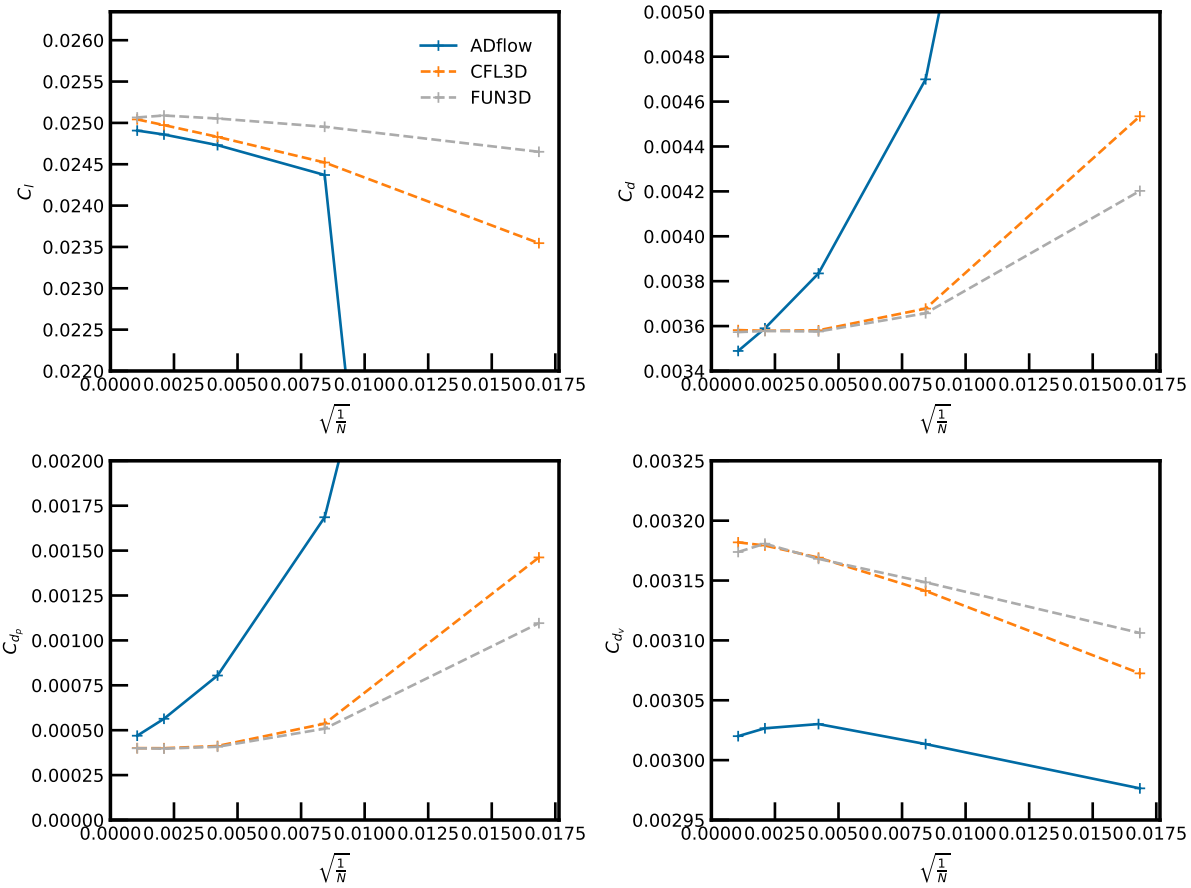

Figure 4.3: Grid convergence for 2D bump. Reference data is from [13].

But ignoring the discrepancies, one can observe that the values appear to smoothly approach a limit.

4.3 Partial derivatives

Forward mode

To verify the forward partial derivatives, the 3D test case is first converged to a relative tolerance of $1e-14$.[2] Once this is the case, the partials are compared against finite difference and complex step. Anecdotal evidence suggest SST is highly non-linear which is reflected in the fact that the FD partials are quite off compared to AD (The maximum observed difference was in the order of $1e2$). Table 4.1 lists the derivatives with the relative accuracy compared to CS (stepsize was $1e-40$).

Derivative	Relative tolerance
$\partial R/\partial u$	$\leq 1e-9$
$\partial f/\partial u$	$\leq 1e-9$
$\partial F/\partial u$	$\leq 1e-9$
$\partial R/\partial x_{geo}$	$\leq 1e-9$
$\partial f/\partial x_{geo}$	$\leq 1e-9$
$\partial F/\partial x_{geo}$	$\leq 1e-9$
$\partial R/\partial x_{aero}$	$\leq 1e-10$
$\partial f/\partial x_{aero}$	$\leq 1e-10$
$\partial F/\partial x_{aero}$	$\leq 1e-10$

Table 4.1: Relative accuracy of forward AD partials compared to CS.

This exact testcase is used in regression test that make sure new features do not break existing ones. Usually a tolerance of $1e-10$ is need to pass this test. But due to the non-linearity of SST, this had to be lifted to $1e-9$. Of course, this is only lifted for SST and the other turbulence models still need to pass $1e-10$.

Backwards mode

To verify the backwards partials, a dot-product test was performed. It is also part of the regression test-suite. Table 4.2 lists each test with the relative tolerance it achieved. The only test that did not pass was $u \rightarrow F$. As F are the nodal forces, those derivatives are only needed for aerostructural optimization. The reason for failing is probably the fact, that those routines buffer some values without recomputing. The changes to the wall distance probably require changes to those buffered routines as well. As this was not done, the test fails.

Dot product test	Relative tolerance
$u \rightarrow R$	$\leq 1e-10$
$u \rightarrow F$	$= 2e-6$
$x_{geo} \rightarrow R$	$\leq 1e-9$
$x_{geo} \rightarrow F$	$\leq 1e-10$
$(u, x_{geo}) \rightarrow (du, F)$	$\leq 1e-10$

Table 4.2: Relative accuracy of dot product test between forwards and backwards partial derivatives.

Reverse_fast mode

The regression test for the reverse_fast partials simply compares them to the forwards routines. This is possible because the fast routines are a subset of the normal ones. Table 4.3 lists the relative accuracy.

[2] In theory, the partials should be accurate regardless of the current convergence. But it is more valuable when one can show that they are accurate for a converged state as this is what we are after.

It obviously does not agree at all. Unfortunately, the author did not have enough time to properly debug this. But the fact that it yields a number and does not simply crash is already an achievement.

backwards vs backwards_fast	Relative tolerance
u	$= 5.7e4$

Table 4.3: Relative accuracy of backwards_fast routines compared to backwards.

4.4 Total derivatives

As described in sec. 3.5.5, the total derivatives were verified by comparing them to complex step. As described in sec. 3.4, ADflow may assemble the adjoint system using either the forwards partials or the reverse_fast partials. Table 4.4 lists the relative difference of various methods compared against complex step with a stepsize of $1e-200$.

Name	cmplx. ($h = 1e-40$)	adj. frwd. 1cpu	adj. frwd. 6 cpus	adj. rev. fast 6 cpus
alpha	-1.3e-09	-5.3e-06	-5.3e-06	-8.0e-03
mach	-6.1e-08	2.3e-05	2.3e-05	2.3e-01
span #0	-2.0e-09	4.0e-04	-9.0e-04	-1.0e-02
twist #0	9.3e-09	-1.0e-03	9.6e-05	-1.4e-02
shape #0	-5.3e-07	5.8e-02	5.6e-02	1.2e-00

Table 4.4: Relative difference of gradients compared to compplex step (stepsize = 1e-200). The function of interest is C_l. When the DVs are a vector (e.g span), only the first variable is listed.

First, lets look at the forward routines with 1 and 6 cpus. The first two variables are *aerodynamic*, which means that they do not control the geometry. It appears that they stay the same on 1 or 6 cpus. Their relative accuracy is approx 1e-5. This is relatively low, but may be explained through the high non-linearity of SST. It is also important to realize that this is the first guesses into the blue regarding options and tuning. One can probably expect a higher accuracy once more time is spend on the problem and the correct options are found.

When looking at the *geometric* derivatives (such as span, twist and shape), it becomes clear that they depend on the number of cpus. Also, they are less accurate than the geometric ones. This probably indicates a problem with the modification done to the wall distance exchange routines. Due to time constraints, it was not possible to fix it.

As already indicated with the partial reverse_fast routines, the total derivatives are also completely wrong. But once again, the fact that a number was obtainable is already an achievement. It is expected that this will be fixed once more time is spent on the problem.

4.5 Summary

The results show that a prototype state was achived. SST does converge using the coupled ANK solver and also gradients can be obtained. But of course, there are still some bugs present. Some other results were more anecdotal and could not be proven scientifically in this project. The following list summarizes the insights (anecdotal and proven) gained through the course of this project.

- SST appears to be implemented correctly.
- ANK does work using the AD preconditioner. But sometimes NaNs appear.
- ANK does not work using FD preconditioner (probably due high non-lineary of SST).
- Physicality check for ANK needs to be adjusted to SST.
- Prototype adjoint is running.

- Aerodynamic derivatives using forward routines are accurate.
- Geometric derivatives using forward routines are not accurate.
- There is probably a bug in the halo exchange of the wall distance in the backwards routines.
- Derivatives using reverse_fast routines are not accurate at all.
- SST is probably highly non-linear.

5. Conclusion

To conclude, this project has made significant strides in enhancing the SST turbulence model for optimization purposes within ADflow. The following key points highlight the project's achievements:

First, the distance to the nearest wall had to be extended for halo cells. This was necessary in order to get rid of some intermediate MPI communication in the F_1 blending function of SST. It was important to get rid of this communication as it prevents automatic differentiation.

The implementation of Automatic Differentiation (AD) for the SST turbulence model was a major milestone. This implementation facilitated the efficient computation of partial derivatives, which are essential for optimization tasks in ADflow. But also the regular solvers like *Approximate Netwon-Krylov (ANK)* and *Newton-Krylov (NK)* profit from this as they now have access to an AD-preconditioner.

It has been shown that the changes did not change the legacy implementation of SST. Using testcases from NASA's *Turbulence Modeling Resource (TMR)*, it has also been shown that the results obtained mostly agree to reference data. Although there is still some uncertainty because a different production term for the turbulence had to be used in order do achieve convergence. It was also not possible to obtain results for the *flatplate* testcase. The author is confident that a bit more time would adress this shortcomings.

It is even possible to obtain total gradients using the adjoint method when assembling the state residual matrix using forward routines. The aerodynamic design variables appear to be correct, although to relatively low tolerance. This my be explained through the high non-linearity of SST. It is also important to note that the geometric design variables seem to be wrong. Additionally, they appear to depend on the number of cpus used. This most likely indicates a problem with the extension of the wall distance to halo cells.

When trying to solve the adjoint system using the reverse_fast routines, on may get a number. But it is completely wrong. But the author still considers this an achievement because ADflow does not simply crash.

In order to maintain the quality and integrity of ADflow, regression test are used. This project did extended them for the SST turbulence model. But because the implementation of SST is only at a prototype stage, the tests are as well.

5.1 Comparison to Goals

When looking at the goals stated in the introduction (section 1.2), most of them have been partially achieved. The following lines take a look at the goals that were missed or only partially fulfilled.

4. Make sure the partial AD derivatives are correct.
    - This has almost been achieved. Only the backwards mode for the Forces is wrong. Also, the backwards_fast mode is not correct.
5. Get the NK/ANK Solver working.
    - Apart from the finite differences pre-conditioner, this has been achieved.
    - The physicality check probably also needs some adjustment for SST.
6. Test and verify the implementation
    - This has been partially achieved. It would be great if the flatplate testcase would have worked. Also the different production terms for SST need to be tested more.

7. Get the adjoint Solver working.
    - Has been achieved in the sense that it does not crash
8. Test and verify the total gradients.
    - This has been partially achieved. The aerodynamic design variables appear to be correct when using forward AD for the state residual matrix. But the aerodynamic variables are slightly off.
    - The gradients are completely off when using the backwards_fast routines.

To conclude, SST has now achieved a prototype state where it is almost ready for optimization. There are still some bugs present, but it is clear where they are and mostly by what they are caused. The only thing that prevented the author from fixing them was a lack of time. As such, one has to realize that getting SST fully working for optimization is a big task. Thus, the outcome of this project should be considered a success.

5.2 Outlook

This report stated openly and clearly where bugs are present and by what they are most likely caused. As fixing those bugs and getting SST ready for production is probably the most valuable target of a proceeding project, the steps that come next should be clear. But still, the following line lists the most prominent points in random order:

- Adapt the physicality check in ANK-turb for SST
- Get the finite difference preconditioner working in ANK for SST. Maybe only the step size needs to be changed.
- Make sure the partials are correctly summed up in the reverse routines that exchange the wall distance in halos.
- Take a closer look at what is going on with total derivatives obtained through complex step. The results have shown that there is a slight difference between a step size of $1e-40$ and $1e-200$. This should not be the case.
- Make sure all the gradients are correct when solving the adjoit system using the forward AD routines.
- Fix the partial backwards_fast derivatives.
- Make sure the gradients are correct when using the backwards_fast routines.
- Optimize an airfoil using SST.

List of Figures

List of Tables

Bibliography

[1] URL: https://chat.openai.com/.

[2] URL: https://mdolab-adflow.readthedocs-hosted.com/en/latest/solvers.html.

[3] David Anderegg. "Adjoint method for efficient gradient computation in optimization". In: (2023).

[4] David Anderegg. "Extension of the SA turbulence model for rough walls in ADflow". In: (2023).

[5] Danske Bank. *HPC Summer School Computational Fluid Dynamics Simulation and its Parallelization*. Processor Research Team, R-CCS Riken. July 3, 2018. URL: https://www.r-ccs.riken.jp/en/wp-content/uploads/sites/2/2020/12/KSano_CFD_20180703_0702_distributed.pdf (visited on 01/20/2023).

[6] *Grid refinement study*. URL: https://mdolab-mach-aero.readthedocs-hosted.com/en/latest/machAeroTutorials/aero_gridRefinementStudy.html.

[7] W.P Jones and B.E Launder. "The prediction of laminarization with a two-equation model of turbulence". In: *International Journal of Heat and Mass Transfer* 15.2 (1972), pp. 301–314. ISSN: 0017-9310. DOI: https://doi.org/10.1016/0017-9310(72)90076-2. URL: https://www.sciencedirect.com/science/article/pii/0017931072900762.

[8] Gaetan K. W. Kenway et al. "Effective Adjoint Approaches for Computational Fluid Dynamics". In: *Progress in Aerospace Sciences* 110 (Oct. 2019), p. 100542. DOI: 10.1016/j.paerosci.2019.05.002.

[9] M. Leschziner. *Statistical Turbulence Modelling for Fluid Dynamics, Demystified: An Introductory Text for Graduate Engineering Students*. Imperial College Press, 2015. ISBN: 9781783266616. URL: https://books.google.ch/books?id=ray8rQEACAAJ.

[10] Charles A. Mader et al. "ADflow—An open-source computational fluid dynamics solver for aerodynamic and multidisciplinary optimization". In: *Journal of Aerospace Information Systems* (2020). DOI: 10.2514/1.I010796.

[11] Joaquim R. R. A. Martins and Andrew Ning. *Engineering Design Optimization*. Cambridge University Press, Jan. 2022. ISBN: 9781108833417.

[12] *Navier-stokes equations*. URL: https://www.grc.nasa.gov/www/k-12/airplane/nseqs.html.

[13] Christopher Rumsey. *Spalart-allmaras model*. URL: https://turbmodels.larc.nasa.gov/spalart.html.

[14] H. Schlichting and K. Gersten. *Boundary-Layer Theory*. Springer, 2018. ISBN: 9783662570951.

[15] Aidan Wimshurst. *[CFD] the K - epsilon turbulence model*. June 2019. URL: https://www.youtube.com/watch?v=fOB91zQ7HJU&ab_channel=FluidMechanics101.

[16] Aidan Wimshurst. *[CFD] the K - omega SST turbulence model*. Mar. 2019. URL: https://www.youtube.com/watch?v=myv-ityFnS4.

[17] Aidan Wimshurst. *[CFD] the K-omega turbulence model*. June 2020. URL: https://www.youtube.com/watch?v=26QaCK6wDp8&ab_channel=FluidMechanics101.

[18] Anil Yildirim et al. In: *Journal of Computational Physics* (), p. 108741. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2019.06.018.