Zurich University of Applied Sciences

Specialisation Project 1

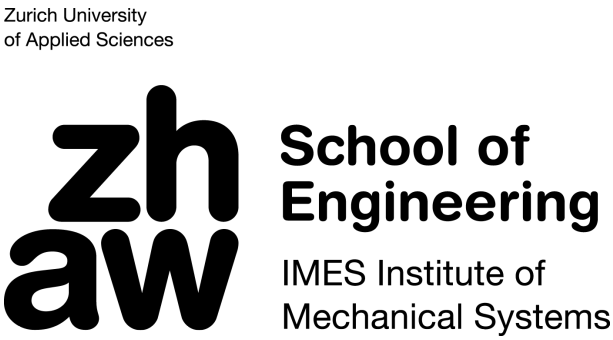
Institute of Mechanical Systems

Extension of the SA turbulence model for rough walls in ADflow

Author:
David Anderegg

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

January 31, 2023



Abstract

As the available computer power increases, Reynolds Averaged Navier Stokes (RANS) based optimizations come more and more in reach. When performing such high fidelity optimization, it is necessary to properly represent all flow conditions. If one fails to do so, the optimizer might exploit effects that do not exist in reality.

This work extends the Spalart Allmaras (SA) turbulence model in the open source CFD solver ADflow for rough walls with a modification originally proposed by Boeing. ADflow is specialized in optimizations and thus uses the *adjoint method* to compute the gradients needed in an efficient manner. To make the rough modification available for optimizations, the changes have been differentiated using Automatic Differentiation (AD).

For verification, the implementation is compared against theory, the open source CFD solver SU2 and experimental data of a *flat plate a zero incidence* for various surface roughnesses. The modified gradients are verified using the complex step method.

The results show that the implementation under-predicts the effect of roughness. But the predicted shape of the effect seems correct. The computed gradients only match to a relative tolerance of 1e−7 compared to complex step. A relative tolerance of less than 1e−8 would be desirable with the methods employed.

Contents

1. Introduction

When performing optimizations, the optimizer tries to maximize the objective as much as possible. This means, one operates in the limit of the modeling process used. Imagine an airfoil optimization where the goal is to increase the maximum coefficient of lift c_l. In this example, only one design variable is available: the angle of attack α. To simulate the airfoils performance, a Reynolds Averaged Navier Stokes (RANS) approach is implemented. During the optimization, the optimizer increases α right until the point where the flow starts to separate as this is the point where the highest C_l lies for an airfoil. But this means, it is absolutely necessary to accurately predict the point of separation. When this is not the case, the optimizer exploits effects that do not exist in reality and thus the whole optimization can not achieve its full potential. In the worst case, the results obtained might not even be usable.

ADflow is an open source RANS solver that also computes the gradients of the objective (and constraints) with respect to the design variables in an efficient manner called the *adjoint method*. In real life aerodynamic applications, as airplanes or wind turbines, surface contamination plays an important role. The primary effect of such contamination is a roughening of the surface. Thus it is important to be able to simulate and optimize taking the effect of rough surfaces into account.

1.1 Goals

The goal of this work is to be able to simulate and optimize under the effects of rough surfaces. ADflow's primary turbulence model is known as *Spalart Allmaras* (SA). It is a one-equation model that was proposed in 1992. It is widely used for external aerodynamic applications. In 2002, a modification for rough surfaces was published. This project implements this modification. Thus the projects aims may be listed as follows:

- Modify the existing SA turbulence model for rough walls
- Test and verify the implementation.
- Use Algorithmic/Automatic Differentiation (AD) to differentiate the newly added code. This is needed for the adjoint method.
- Test and verify the modified gradients.

1.2 ADflow

Before heading straight in, a few words about ADflow and its history are provided. It is an open-source RANS solver that is developed and maintained at *MDOLab* at the university of Michigan. It is based on a structured, multiblock solver called *sumb* and has been adapted for gradient based optimization by means of *Algorithmic Differentiation*[1] and the *adjoint method*.

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[2] and ANK[3] solvers. Those can achieve machine-precision convergence, even for aircraft configurations at an angle of attack of 90°[12] [10] [18].

[1] Thats what AD stands for in ADflow.
[2] NK stands for the Newton–Krylov method.
[3] ANK is an approximated Newton–Krylov 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.

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/259`
Test cases	`https://github.com/DavidAnderegg/SA_rough_testcases`

2. Theoretical Fundamentals

2.1 Boundary layer

To introduce the concept of a boundary layer, one has to think of a uniform flow in one direction with a certain speed U_∞. Now place a thin plate into this flow where its long side aligns with the flow direction. This setup is know as *flat plate at zero incidence*. At the surface of the wall, the *no slip condition* must be satisfied. This means, the flow slows down until it reaches exactly zero at the surface. This slow down does not happen linearly and a large portion of the flow remains uniform. The flow is slowed down only near the surface of the plate due to friction forces. This area is called the *boundary layer* or *frictional layer*. It's thickness $\delta(x)$ depends on a lot of different factors, but most prominently on its position from the leading edge. In reality, there is no hard border between the uniform flow and the boundary layer. Thus it is often defined as where the flow reaches 99% of the velocity of the outer flow [16]. Figure 2.1 shows this concept.

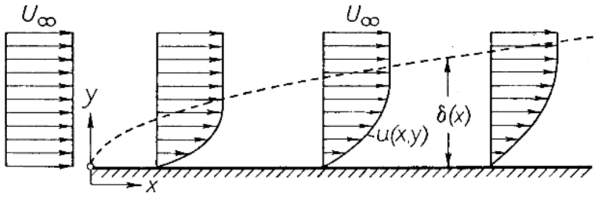


Figure 2.1: Laminar boundary layer of a flat plate at zero incidence [16].

Types of boundary layers

The flow inside the boundary layer might be *laminar* or *turbulent*. In reality, it is laminar at the leading edge, transitions to turbulence over a certain length and becomes fully turbulent afterwards [16]. As this report deals with a fully turbulent *turbulence model*, the first two types of boundary layer are not explained further.

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* [16]:

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

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

2.1.1 Turbulent boundary layer

When looking closer at a turbulent boundary layer, one can observe two different regions. At the top is a *turbulent layer* which is only indirectly affected by the friction with the wall. At the bottom is a layer that is really thin compared to the boundary layer. It is called *viscous sublayer* or *viscous wall layer* and

is directly affected by the friction. As is the case with the boundary layer itself, there is no hard border between those two regions. Instead one can observe a smooth transition [16].

To examine the cross section of the boundary layer, it make sens to introduce the concept of the dimensionless *wall distance* y^+. Hand in hand goes the dimensionless velocity u^+. Moving to this dimensionless system allows us to compare different boundary layers from different flow conditions more easily. The velocity u^+ is given by [16]:

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

Where as 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:

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

Where as 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 [16]:

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.

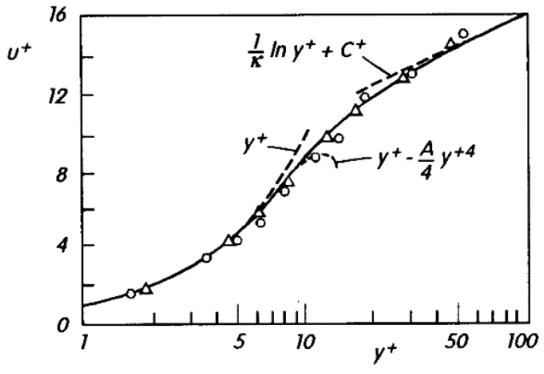


Figure 2.2: Cross section of a fully developed turbulent boundary layer overlayed with measurements[16].

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

2.1.2 The effect of roughness

Before the effects of roughness may be investigated, it makes sense to describe a standard roughness which is easier to compare. Take a look at figure 2.3. It is assumed the surface is tightly packed with balls of equal diameter. Because this is similar to sandpaper, this roughness is called *sand roughness* or *standard roughness*. The diameter of the balls is the *sand roughness height* k_s. It is a measure for the roughness of a surface [16].

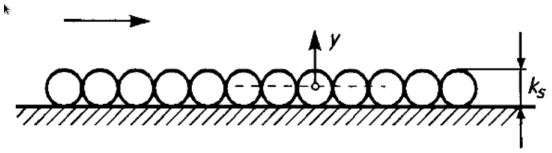


Figure 2.3: Definition of *sand roughness* and k_s [16].

Analog to u^+ and y^+ (see section 2.1.1), there exists a dimensionless k_s^+:

$$k_s^+ = \frac{k_s u_\tau}{\nu} \tag{2.7}$$

Equivalent sand roughness

Real surfaces are not made up of balls as assumed in figure 2.3, but instead have a random distribution of peaks and vales. It is possible to define a *equivalent sand roughness* k_{seq} which has the same effect on the boundary layer as the *standard roughness* assumed so far. The author has found some formula which allow the conversation of one to another, but they did not match with experimental data freely available and are thus deemed unreliable. According to [16], k_{seq} must be obtained experimentally. It lists some values for common materials and surface finishes[2].

The effect of roughness

In figure 2.4, the velocity distribution for different k_s^+ values is plotted. When looking at a certain y^+ value, the rougher the surface is, the slower the flow. Because the boundary layer is the region where the flow reaches the infinite velocity (U_∞), it must grow bigger. Thus the effect of roughness is simply a thickening of the boundary layer. This leads to an increase of the skin friction and ultimately drag. Additionally, the separation behavior might be changed.

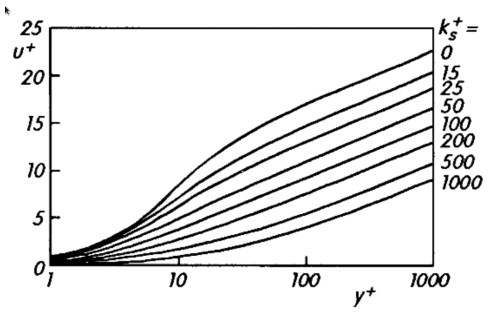


Figure 2.4: Velocity distribution for different surface roughness values [16].

Rough regimes

Depending on the height, roughness has different effects on the boundary layer. It may be classified in tree distinct regimes: *hydraulically smooth*, *transition region* and *fully rough*. They correspond approximately to the three different regions of the boundary layer. When the roughness is purely contained by the viscous sublayer, it does not have an effect at all. When it projects out of the viscous sublayer, the

[2] This may be found on page 532.

roughness effect start. If the roughness element projects right into the overlap layer, the viscosity effects vanish and the flow becomes fully rough [16].

When looking at equation 2.6, roughness simply changes the variable C^+. Thus it may be summarized as follows:

hydraulically smooth:	$0 \leq k_s^+ \leq 5$	$C^+ \approx 5.0$
transition region[3]:	$5 < k_s^+ < 70$	$C^+(k_s^+)$
fully rough:	$70 \leq k_s^+$	$C^+ \approx 8.0 - \frac{1}{\kappa} ln(k_s^+)$

Table 2.1: C^+ dependence on k_s^+ [16].

2.2 Reynold's Averaged Navier Stokes (RANS)

The *Navier-Stokes* equations describe how tho the *velocity* (U), *pressure* (P), *temperature* (T) and *density* (ρ) of a moving fluid are related. They are partial differential equations which almost allway can't be solved analytically. Thus numerical methods are needed. The physical quantities depend on four variables: *spatial coordinates* (x, y, z) and the *time* (t). As numerical methods are used, those quantities need to be discretized [13].

In high Reynold's number flows, the length and time-scales of different eddies differs drastically. To adequately resolve the smallest eddies, which are basically turbulence, such a fine mesh and timestep would be needed that it is not viable. To deal with it, simplifications are needed: (1) solving for a steady flow and (2) accounting for turbulence stochastically. This means only one simulation (instead of one every x millisecond) and a much coarser grid is sufficient.

Reynolds averaging

Osborne Reynolds proposed in 1895 a solution which would later be known as *Reynolds Averaged Navier-Stokes*. The basic idea is to separate the velocity (and all the other physical properties which are resolved) in two components [11]:

$$U_i = \bar{U}_i + u_i' \qquad P = \bar{P} + p' \tag{2.8}$$

Where the subscript i stands for all three spatial coordinates (x, y, z). $\bar{U}_i$ is the mean velocity, which does not change and u_i' is the fluctuating part due to turbulence which is not resolved. The same notation is used for P. Plugging this into the incompressible Navier-Stokes equations leads to:

$$\frac{\partial \rho \bar{U}_i \bar{U}_j}{\partial x_j} = \frac{\partial \bar{P}}{\partial x_i} + \frac{\partial}{\partial x_j} \nu \left(\frac{\partial \bar{U}_i}{\partial x_j} + \frac{\partial \bar{U}_j}{\partial x_i} \right) - \frac{\partial}{\partial x_j} \rho u_i' u_j' \tag{2.9}$$

Where $\rho u_i' u_j'$ are the *six* independent *Reynolds-stresses*. Please note the fluctuating part for pressure p' cancels out and does not appear again. The Reynolds-stresses may be denoted in tensor notation:

$$\rho \bar{u}_i \bar{u}_j = \rho \begin{pmatrix} u_1'^2 & u_1' u_2' & u_1' u_3' \\ u_2' u_1' & u_2'^2 & u_2' u_3' \\ u_3' u_1' & u_3' u_2' & u_3'^2 \end{pmatrix} \tag{2.10}$$

Equation 2.9 is complemented by the Reynolds-averaged mass-conservation equation:

$$\frac{\partial \rho \bar{U}_j}{\partial x_j} = 0 \tag{2.11}$$

Turbulence model

To calculate the unknown Reynolds-stresses, a *turbulence model* is used. There exist multiple approaches, but two are the most widely used: *Eddy viscosity models* and *Reynolds stress transport models*. As the *Spalart Allmaras* model is the former one, only this is explained in more detail.

[3] Please note C^+ is not further described in the transition region.

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:

$$-\rho u_i' u_j' = \nu_t \left(\frac{\partial \bar{U}_i}{\partial x_j} + \frac{\partial \bar{U}_j}{\partial x_i}\right) - \frac{2}{3}\delta_{ij}\rho k \tag{2.12}$$

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 [11].

2.2.1 Spalart Allmaras (SA) turbulence model

The *Spalart Allmaras* turbulence model has been introduced in 1992. It is a *one-equation* model which means it only computes the eddy viscosity μ_t. When looking at equation 2.12, the most right hand term is a correction which is neglected as the turbulent kinetic energy is not readily available. Therefore this model trades accuracy for speed [11] [15].

Analog to u^+ one can convert the eddy viscosity to wall scaling:

$$\nu_t^+ = \frac{\nu_t}{\nu} \tag{2.13}$$

In figure 2.5, its contour is plotted in the boundary layer of a flat plate a zero incidence. It is almost linear in the logarithmic region, but varies as $(y^+)^4$ in the viscous sublayer. Finite volume methods are only able to interpolate linearly in cells. This means a high number of cells would be needed to properly represent ν_t^+ in the viscous sublayer. The Idea of the Spalart Allmaras (SA) turbulence model is to replace ν_t^+ with a modified eddy viscosity $\tilde{\nu}^+$ which behaves linearly right to the surface. Additionally to reducing the number of cells, this also makes the numerical behavior more stable. As the viscous sublayer can not be neglected, the SA model adjusts $\tilde{\nu}^+$ in a post-processing step to reflect the real ν_t^+ behavior [9] [1].

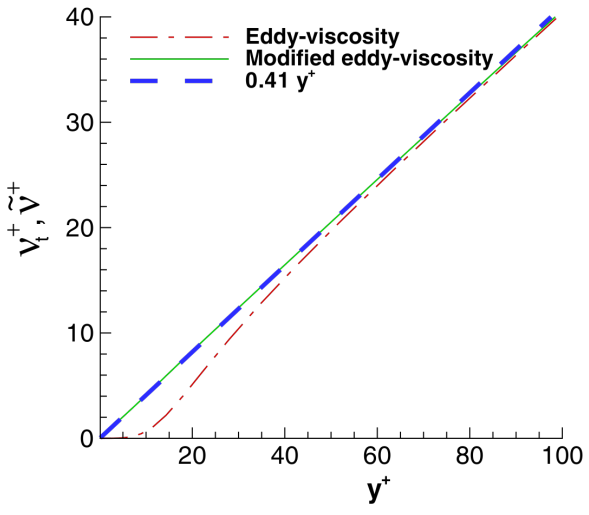


Figure 2.5: Eddy viscosity ν_t^+ and modified eddy viscosity $\tilde{\nu}^+$ used in the Spalart Allmaras model [9].

The transport equation is given by [15]:

$$\frac{D\tilde{\nu}}{Dt} = c_{b1}(1-f_{t2})\tilde{S}\tilde{\nu} - \left[c_{w1}f_w - \frac{c_{b1}}{\kappa^2}f_{t2}\right]\left(\frac{\tilde{\nu}}{d}\right)^2 + \frac{1}{\sigma}\left[\frac{\partial}{\partial x_j}\left((\nu+\tilde{\nu})\frac{\partial\tilde{\nu}}{\partial x_j}\right) + c_{b2}\frac{\partial\tilde{\nu}}{\partial x_i}\frac{\partial\tilde{\nu}}{\partial x_i}\right] \tag{2.14}$$

Then, the turbulent energy is computed as:

$$\nu_t = \rho\tilde{\nu}f_{v1} \tag{2.15}$$

where

$$f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3} \qquad \chi = \frac{\tilde{\nu}}{\nu} \qquad \tilde{S} = \Omega + \frac{\tilde{\nu}}{\kappa^2 d^2}f_{v2} \tag{2.16}$$

$$\Omega = \sqrt{2W_{ij}W_{ij}} \qquad f_{v2} = 1 - \frac{\chi}{1+\chi f_{v1}} \qquad f_w = g\left[\frac{1+c_{w3}^6}{g^6+c_{w3}^6}\right]^{1/6} \tag{2.17}$$

$$g = r + c_{w2}(r^6 - r) \qquad r = min\left[\frac{\tilde{\nu}}{\tilde{S}\kappa^2 d^2}\right] \qquad f_{t2} = c_{t3}exp(-c_{t4}\chi^2) \tag{2.18}$$

$$W_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{x_j} - \frac{\partial u_j}{\partial x_i}\right) \qquad d = \text{Distance to the nearest wall} \tag{2.19}$$

The boundary conditions are:

$$\tilde{\nu}_{wall} = 0 \qquad \tilde{\nu}_{farfield} = 3\nu_\infty : to : 5\nu_\infty \tag{2.20}$$

And the constants are:

$$c_{b1} = 0.1355 \qquad \sigma = 2/3 \qquad c_{b2} = 0.622 \qquad c_{w1} = \frac{c_{b1}}{\kappa^2} + \frac{1+c_{b2}}{\sigma}$$
$$c_{w2} = 0.3 \qquad c_{w3} = 2 \qquad c_{v1} = 7.1 \qquad c_{t3} = 1.2$$
$$c_{t4} = 0.5$$

Please note the distance to the nearest wall d is used as the turbulence length scale. It reduces the size of the eddies near the wall. This is a physical phenomenon which has been observed in reality [11]. But it can be a challenge for parallel CFD-codes as those distances must be communicated across different processors.

2.2.2 Modification of SA for rough walls

As can bee seen in figure 2.4, the main effect of roughness is a downward shift in the log region. According to [4], two different methods are proposed:

- Models where the eddy viscosity is zero at the wall. The effect of the roughness is modeled through a reduction of the damping function.
- Models where the eddy viscosity is not zero at wall. This can be interpreted as shifting the wall upwards into the roughness. According to [4], this approach works better for small roughness values.

There exists two different extension for the SA model which employ the strategy of a non zero eddy viscosity at the wall. One proposed by *Boeing* and the other by *ONERA*. The approach by ONERA is slightly more complicated and also needs the friction velocity. This work implemented the extension proposed by Boeing.

Changed terms

The biggest change is the distance d which accounts for the shift of a "virtual" wall upwards [4][15]:

$$d_{new} = d + 0.03k_s \tag{2.21}$$

Additionally, χ is modified:

$$\chi = \frac{\tilde{\nu}}{\nu} + c_{R1}\frac{k_s}{d_{new}} \tag{2.22}$$

Where

$$c_{R1} = 0.5$$

As the new definition of χ should not affect $\tilde{S}$, the function f_{v2} also needs to be modified:

$$f_{v2} = 1 - \frac{\tilde{\nu}}{\nu + \tilde{\nu} f_{v1}} \tag{2.23}$$

And finally, the boundary condition is replaced by:

$$\left(\frac{\partial \tilde{\nu}}{\partial n}\right)_{wall} = \frac{\tilde{\nu}_{wall}}{0.03 k_s} \tag{2.24}$$

Where ∂n is along the normal of the wall.

2.3 Adjoint method

For optimizations, the gradients of the objective (e.g c_d) with respect to the design variables are needed. ADflow computes them using the *adjoint method*. The author has written a report about it for his module called *complementary module 1*. Thus it will not be explained any further and the interested reader may take a look at it here [3].

2.4 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 an 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:

$$h = N^{-1/d} \tag{2.25}$$

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:

$$\hat{p} = ln(\frac{f_{L2} - f_{L1}}{f_{L1} - f_{L0}})/ln(r) \tag{2.26}$$

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 [7].

3. Methods

3.1 Implementation

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.

Before explaining how the modifications for SA-rough were implemented, a couple of words about the architecture of ADflow are needed.

Block architecture This solver only reads *structured grids*, this means all state and grid variables can be represented in a "three dimensional table". One can also think of a 3D array. This organization is called the *block architecture*. The index of the block variables are `i`, `j` and `k`. Each block has 6 faces. They are identified with `iMin`, `iMax`, `jMin`, etc.

Boundary Conditions ADflow saves the values of the boundary conditions in a similar way as the block variables. But they are 2d arrays where the indexes `i` and `j` are used. It is possible to directly relate a volume cell in the block to a surface cell at the boundary, but it is important to realise surface and volume cells are different entities.

Global Cell ID ADflow assigns a *global cell id* `gID` to each volume cell. The index is a continuously increasing integer that starts at **0**. It is applied in a systematic manner that will not be explained further.

Block splitting ADflow is capable of solving the governing equations by parallel means. To make this possible, the whole mesh is split in different blocks. Each processor then only loads its corresponding sub-block. It is important to realize that no single processor has directly access to the whole mesh[1].

3.1.1 General thoughts

This modification requires a little bit more RAM and cpu power. It could also obscure the standard implementation of the SA model. To cater those considerations, an on/off switch has been introduced in the python layer called `useRoughSA`. When it is `False`, all implemented changes are disabled and ADflow behaves exactly as it did before.

3.1.2 Changes to wall distance

The regular SA model needs the distance to the nearest wall. ADflow computes this distance in a preprocessing step and saves it in a block variable called `d2wall`. It is slightly more complicated as ADflow can handle warping meshes and thus this distance needs to be adjusted after each warp, but this will be explained in more detail later. As shown in equation 2.21, the distance to the nearest wall needs to be modified for the rough SA variant. There are two strategies possible:

1. Overwrite the current `d2wall` with the modified value. E.g. do the modification in a preprocessing step.

[1] Assuming more than one processor is used

2. Keep the current `d2wall` as it is and apply the modification later when the distance is actually needed.

Strategy (2) has been chosen as changing the `d2wall` might have unforeseen consequences and is just obscuring. But this means, a new block variable called `ks` is introduced. It will hold the roughness value of the nearest surface.

Calculating the distance to the nearest wall

To be able to assign the correct roughness value of the nearest wall, one must know which wall is nearest to the current volume cell. For the calculation of the wall distance, this information is already needed. Thus it was natural to adapt this function. To explain the changes, one must know how it works:

1. The function `buildClusterWalls` is called. It figures out which surface mesh belongs to the *no-slip wall* type and gathers all of it on each processor. This does not scale. But it is assumed the surface mesh is orders of magnitudes smaller than the volume mesh. Therefore this only becomes a problem when the size of the volume mesh approaches hundreds of millions of cells.

   Once it has this information, it builds up the whole surface mesh. This is not straight forward as it might be an overset[2] mesh where different meshes are overlapping. It must decide which cells to drop and which to keep.

   After that, it relates the surface mesh to the volume cells and returns the `gID`. At first glance, it might seem weird to return "volume cells" when the distance to a surface cell is required. But the grid points of both types are the same and for the walldistance computation the boundary conditions do not matter. This construct is called `clusterWalls`.

2. Once the `clusterWalls` are built, the function `determineWallAssociation` iterates through all the volume cells on the current processor and figures out which `gID` of the `clusterWalls` is nearest.

   Whit that information, it creates a "PETSc scatter"[3] object called `wallScatter`. This object is (in simplified terms) a two dimensional list which keeps track of which surface cell is nearest to which volume cell. As the surface cells have been replaced with volume cells before, this is basically a mapping of volume cells to volume cells.

   After that, the memory for `clusterWalls` is released.

3. A new block variable `xSurf` is introduced. It holds all the surface grid points, which are needed for the wall distance calculation for the volume cells of the current processor. It is the receiving end of the `wallScatter` object.

4. In the End, `updateWallDistancesQuickly` is called to actually compute the distance to the nearest wall based on the grid points stored in `xSurf`.

5. After the mesh has been warped, only `updateWallDistancesQuickly` is called. This means, it is assumed the nearest surface cell does not change, only its coordinates.

Please note the outlined steps above are in reality a bit more complicated and only the broad context is described.

Assigning the block variable `ks`

To fill the block variable `ks` with the roughness value of the nearest surface, the before described walldistance computation is hijacked as follows:

A Introduce a new block variable called `nearestWallCellInd`. It holds the `gID` of the nearest surface cell. Its value is assigned in step (2) where the `gID` of the nearest surface cell is determined.

[2] This is also know as Chimera patch.
[3] PETSc stands for Portable, Extensible Toolkit for Scientific Computation.

B Create a new subroutine called `updateWallRoughness`. A separate subroutine is needed as the roughness value on the boundary is only read from the CGNS[4] file once the walldistance has already been calculated. Thus it is not possible to do it when step (2) is done. ADflow has some helper functions that allow to overwrite the values of the boundary conditions which are saved in the CGNS file. Having a separate subroutine which is called later allows to also use those helper functions for the wall roughness.

Now, the inner workings of `updateWallRoughness` are explained in more detail:

A Each processor creates two lists: List (a) holds the roughness values of the surface cells on the current processor and list (b) holds the corresponding `gID`. To calculate the `gID`, the surface cell must be related to the volume cell.

B Then those two lists are gathered on all processors. This means, every processor has a list (α) of all surface roughness values and a list (β) of all corresponding `gID`. This does also not scale. But this constraint has been violated before (in `buildClusterWalls`) and thus the totally required RAM is not increased significantly.

C Now, each processor iterates through its volume cells and requests the `gID` from `nearestWallCellInd`. Then it searches list (β) until it finds the same `gID` and keeps the index `I` where it found it.

D Finally, it assigns the value of list (α) at index `I` to the block variable `ks`.

It is important to note that this strategy can handle different roughness values for the surface. But those values are not interpolated and are thus only accurate in the limiting sense of an infinitely fine grid. It can also lead to weird situations where one single volume cell is assigned a roughness value where as its surrounding cells are not. This happens because its center is somehow closest to a small corner of a rough surface cell. This also vanishes with increasing cell count.

3.1.3 SA source terms

As described in section 2.2.2, the terms χ (equation 2.22) and f_{v2} (equation 2.23) need to be modified. This has been straight forward. But it is important to not forget the calculation of d_{new} (equation 2.21) as this has been postponed in the previous section.

ADflow employs some Newton-type solvers that require the Jacobian of the resiudals. Thus it was necessary to also modify the derivative of $\partial f_{v2}/\partial\tilde{\nu}$ as follows:

$$\frac{\partial f_{v2}}{\partial\tilde{\nu}} = \frac{\tilde{\nu}^2 \frac{\partial f_{v1}}{\partial\tilde{\nu}} - \nu}{(\tilde{\nu} f_{v1} + \nu)^2} \tag{3.1}$$

As said before, those changes are only active when `useRoughSA` is `True`.

3.1.4 SA boundary conditions

ADflow employs the concept of *halo cells*. This is an idea to exchange the boundaries of the split blocks when running in parallel. Take a look at figure 3.1. On the left side, a block split in 4 is shown. Each sub-block lives on a different processor. On the right, one can see the artificial halo cells. After each iteration of the flow solver, the values of the halo cells are updated with their corresponding values of the other blocks (red arrows).

[4]CGNS stands for CFD General Notation System and is a format for meshes and their solutions. It is the primary format that ADflow interacts with.

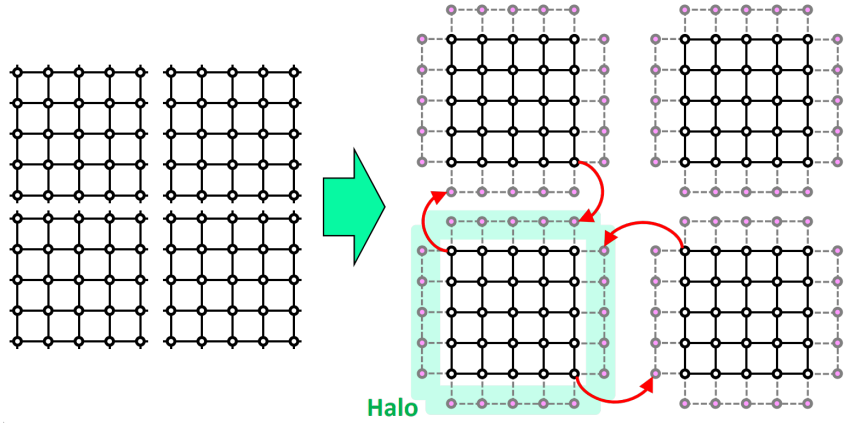


Figure 3.1: A block split in 4 (left) and its corresponding halo cells (right) [5].

ADflow is capable of deploying second order schemes and thus needs two layers of halo cells. But the idea remains the same. As they are not always active, ADflow takes care of interpolating them by itself.

Explicit boundary conditions

The concept of halo cells can also be used to prescribe the regular boundary conditions like a *no-slip wall*. According to equation 2.20, the regular SA model needs a boundary condition of $\tilde{\nu}_{wall} = 0$. This means, the first halo is simply updated with the negative value of the first cell:

$$\tilde{\nu}_h = -\tilde{\nu}_1 \qquad \rightarrow \qquad \tilde{\nu}_{wall} = \frac{\tilde{\nu}_1 + \tilde{\nu}_h}{2} = 0 \tag{3.2}$$

Where $\tilde{\nu}_1$ is the value of the first interior cell and $\tilde{\nu}_h$ is the halo cell.

For the rough version, the boundary conditions need to be modified according to equation 2.24:

$$\left(\frac{\partial \tilde{\nu}}{\partial n}\right) \equiv \frac{\tilde{\nu}_1 - \tilde{\nu}_h}{2d} = \frac{\tilde{\nu}_{wall}}{0.03k_s} \tag{3.3}$$

Replacing $\tilde{\nu}_{wall}$ with equation 3.2 and solving for $\tilde{\nu}_h$, one gets:

$$\tilde{\nu}_h = \tilde{\nu}_1 \frac{0.03k_s - d}{0.03k_s + d} \tag{3.4}$$

The underlying assumption is, that the first cell to the wall is not skewed and its center is normal to the wall. This is in general good practice and highly recommended for RANS meshes.

Implicit boundary conditions

ADflow uses a *Diagonal Dominant Alternate Direction Implicit* method (DDADI) to solve the turbulence equation. This requires a special treatment of the boundary conditions. Thus they also had to be modified. Because this is implementation specific, it will not be explained further.

3.1.5 Automatic Differentiation

As described in [3], ADflow uses Automatic/Algorithmic Differentiation to compute the partial derivatives which are needed in the *adjoint method* to compute the total derivatives of the functions of interest with respect to the design variables. The AD tool used is called *tapenade* [8]. It is based on JAVA and directly differentiates Fortran source code. One provides the source code of a Fortran routine, defines the dependent output variable and the independent input variable with respect to which the derivative is requested. Tapenade then returns Fortran source code that computes this derivative.

When the AD architecture for ADflow was set up, tapenade could not handle parallelization calls using MPI[5]. Thus the decision was made to split the whole AD in two parts: part (1) communicates data across different processors, i.e takes care of the parallelization; and part (2) does the actual computation. This allows tapenade to differentiate the math heavy part (2) and necessitates the developer to make sure the differentiated routines are called appropriately in part (1).

3.2 Verification

3.2.1 Roughness propagation

When assigning a roughness value to the surface, this value gets propagated to the volume cells as described in section 3.1.2. This has to be tested, especially when running on multiple processors. To get a chance to do this, ADflow has been modified to write the `ks` values to the solution grid (if requested).

Cube

The first test is a cube where each face is split into 9 parts. The center of each face has been prescribed a roughness value of $k_s = 1.0$. The rest of each face gets a value of $k_s = 0.1$. Figure 3.2 shows the surface with the corresponding roughness values. This test makes it easy to verify that the propagation is sound.

Overset cube

The highjacked subroutine for the wall distance calculation takes a slightly different path when overset meshes are used. To test it, the cube mesh from before was repurposed. It was extendend with a coarse cartesian background grid. ADflow uses the *implicit hole cutting scheme* to decide by itself how to interpolate between those grids. This was failing with the initial cube grid. To make it work, the grid had to be refined slightly.

Cuboid

Testing the conversion from the surface cell to the global cell ID (`gid`) is not possible for a symmetric case. Mixing up `i` and `j` would still yield the same result for the cube. To test this properly, a cuboid is introduced. One each face, a random cell is made rough ($k_s = 1.0$). As before, the rest of the face gets a roughness value of $k_s = 0.1$. This setup can be seen in figure 3.2. The basic idea for this test is as follows: When there is a `i` and `j` mixup, the calculation of the `gID` yields a number that does not exist. Thus the mixup results in failing code. This behavior has been observed as there was a mixup initially.

All volume meshes in this report were generated using pyHyp [17]. It reads a surface mesh and extrudes it into the third dimension. But it allways orients the internal block coordinates (`i`, `j` and `k`) in the same direction such that the `kMin` face is allways the wall. The correlation of the surface cell to the `gID` is different for each block face. To test all faces, 6 different meshes where created. Each mesh has the same coordinates, but the internal block orientation was changed such that each face is the wall once.

[5]MPI stands for Message Passing Interface and handles the communication across different processors and computers when performing parallel computations.

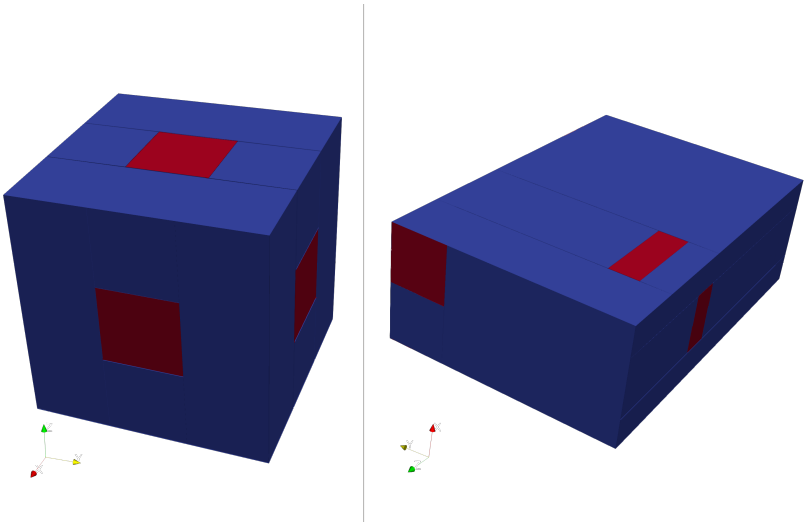


Figure 3.2: Cube and cuboid for k_s propagation test. Blue means $k_s = 0.1$ and red equals $k_s = 1.0$.

3.2.2 Flapt plate at zero incidence

It is necessary to compare the SA rough implementation against theory and experiments. As it is hard to get experimental data, the *flat plat a zero incidence* test case has been chosen for all comparisons. NASA maintains a website called *Turbulence Modeling Resource* (TMR) [14]. There, various formulations for different turbulence models may be found. Additionally it provides testcases, grids and validation data. The following grid-family with its corresponding boundary conditions was sourced from there (figure 3.3). Table 3.5 lists the different mesh sizes. Except for the grid convergence study, all simulations were done using the finest grid. The grid refinement ratios is $r = 2$ for these meshes.

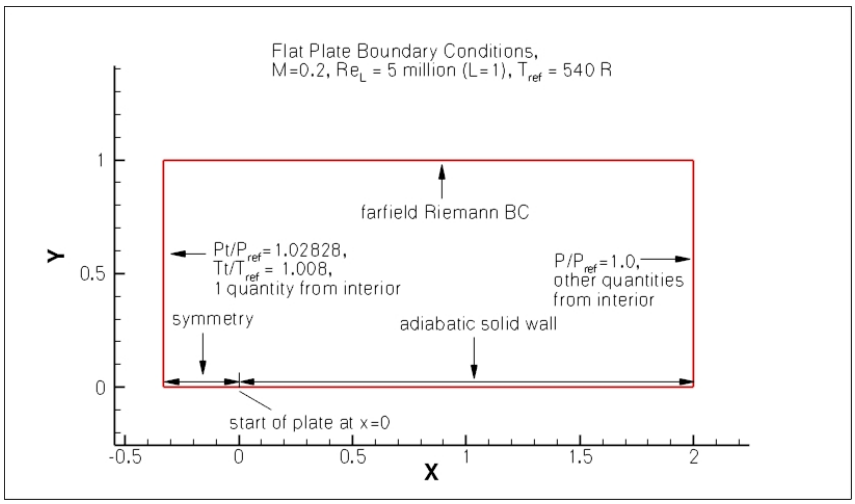


Figure 3.3: Boundary conditions and test case overview [14].

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 3.1: Mesh sizes used for testcases.

Clean

The NASA TMR website [14] provides numerical data on the skin friction coefficient (c_f) for a clean wall. This data is used to verify SA rough with a surface roughness of $k_s = 0$. The following table lists the flow conditions used.

Temperature	300	°K
Pressure	1013	hPa
Inlet velocity	78.544	m/s
Surface roughness	0	m

Table 3.2: Flow conditions for the clean case.

Blanchard

For his PhD thesis, Blanchard performed some experiments with flows over rough walls. The source is written in french and was not available. The data for the comparison was extracted from [4]. Only skin friction coefficients for one testcase could be obtained. [4] estimates k_s^+ to be around 150. The same meshes as in table 3.5 were used, but the flow conditions were adapted as follows:

Temperature	288	°K
Pressure	1013	hPa
Inlet velocity	45	m/s
Surface roughness 1	0	m
Surface roughness 2	0.001 095	m

Table 3.3: Flow conditions for the blanchard case.

Acharya et al.

Acharya et al. provides some experimental data for flows over rough surfaces [2]. Skin frictions coefficients could be obtained for two testcases. For the last case, a mean velocity profile was also available. [4] estimates k_s^+ to be around 25 for the first testcase and approximately 70 for the second. The same meshes as before were use, but the flow conditions were adapted as follows:

Temperature	288	°K
Pressure	1013	hPa
Inlet velocity	19	m/s
Surface roughness 1	0	m
Surface roughness 2	0.0005	m
Surface roughness 3	0.002 05	m

Table 3.4: Flow conditions for the Acharya case.

SU2

SU2 is an open source solver [6] that has allready implemented SA rough. It is used to compare the SA rough implementation of ADflow against data from a different solver. The same meshes were used with the following flow conditions:

Temperature	300	°K
Pressure	1013	hPa
Inlet velocity	78.544	m/s
Surface roughness 1	0	m
Surface roughness 2	0.0001	m
Surface roughness 3	0.001	m

Table 3.5: Flow conditions for the SU2 case.

3.3 Automated tests

ADflow uses automated tests to catch errors and breaking changes. They are run automatically when a new pull request has been created on its github page. Most of the tests are *regression tests*, but a couple of *unit tests* are also to be found.

The idea of *Regression tests* is to catch breaking changes. The are usually (in this context) a typical CFD case with a really coarse mesh. Once a feature is completed, a reference file is trained with this case. This reference file stores some identifiers of the cfd solution to a high precision. The test then runs the same CFD case and compares the result with the reference file.

The purpose of *unit tests* is to test a feature while it is developed. It usually tests a function with defined inputs and expects a defined output.

In ADflow, there is no clear cut between those two types. They are mostly regression, but also a bit unit tests.

3.3.1 Test setup

To test the SA rough changes, an allready existing test setup was modified. It is based on an wing using the transonic airfoil *RAE 2822*. The wing is swept and might resemble a simplified airliner wing. As ADflow is used in optimizations, the gradients for various design variables need to be tested. Figure 3.4 shows the coarse surface mesh and the FFD[6] points. Table 3.6 lists the design variables used. Table 3.7 lists the flow conditions and mesh size.

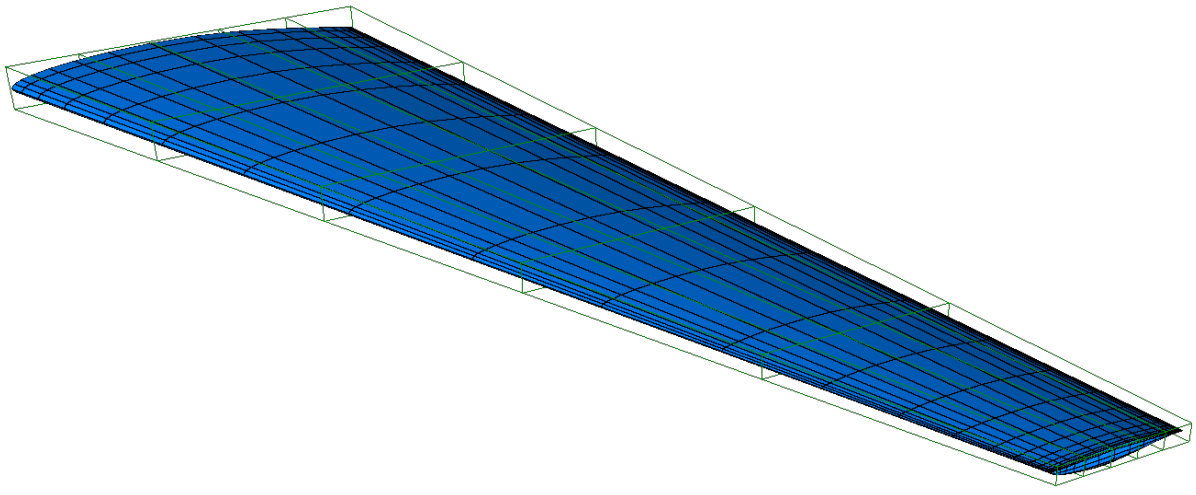


Figure 3.4: Surface mesh (blue) and FFD points (green) for the automated test setup.

[6]FFD stands for Free Form Deformation and is a parametrization technique.

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 etc.
yRef	aerodynamic	Y location of reference point for moment calculation etc.
zRef	aerodynamic	Z location of reference point for moment calculation etc.

Table 3.6: Design variables for automated test setup.

Specific gas constant	287.87	J / (kg K)
Pressure	200	hPa
Temperature	220	°K
Alpha	1.8	°
Beta	0	°
Mach	0.8	-
Mesh node count	30 375	-
Mesh cell count	24 192	-

Table 3.7: Flow conditions for the automated test setup.

3.3.2 Introduced tests

The following table 3.8 lists all newly introduces tests.

Test-Name	Comment
`TestAdjoint_5_Rough_SA_wing`	Surface of whole wing is rough ($k_s = 0.001$).
→ `rest_residuals`	Make sure ADflow converges to the same solution as reference file.
→ `rest_adjoint`	Make sure the adjoint vectors converge to the same solution as a reference file.
`TestCmplxStep_5_Rough_SA_wing`	Same test as before, but use complex step.
→ `cmplx_test_aero_dvs`	Make sure trained adjoint values from before are consistent with complex step (only aerodynamic design variables).
→ `cmplx_test_geom_dvs`	Make sure trained adjoint values from before are consistent with complex step (only geometric design variables).
`TestFunctionals_9_Rough_SA_wing`	Surface of whole wing is rough ($k_s = 0.001$).
→ `rest_residuals`	Make sure ADflow converges to the same solution as reference file.
→ `rest_restart_read`	Make sure restarts are possible.
→ `rest_functions`	Make sure function values (C_l, C_d, etc.) are the same as in reference file.

Test-Name	Comment
→ `test_forces_and_tractions`	Make sure tractions and forces are the same as in reference file.
→ `test_jac_vec_prod_fwd`	Make sure partial derivatives calculated using the forward mode are consistent with reference file.
→ `test_jac_vec_prod_bwd`	Make sure partial derivatives calculated using the backwards mode are consistent with reference file.
→ `test_dot_products`	Make sure the forward and backwards Algorithmic Differentiation is consistent using the dot-product test.
`TestFunctionals_10_Rough_SA_wing`	Same tests as in `TestFunctionals_9_Rough_SA_rans_tut_wing`, but the surface has a roughness of $k_s = 0$ (clean). The reference file is from a standard SA testcase. This should make sure SA rough with $k_s = 0$ behaves like the standard SA model.

Table 3.8: Newly introduced tests for SA rough.

4. Results

4.1 Roughness propagation

As described in section 3.2.1, three different test cases were setup to verify the propagation of the surface roughness values to the correct volume cells.

Cube

This test was designed to make sure the propagation is sound. Figure 4.1 shows the surface cube with the rough patch in the middle of each face. The volume mesh is sliced in two axis and the propagation of the roughness values can be seen.

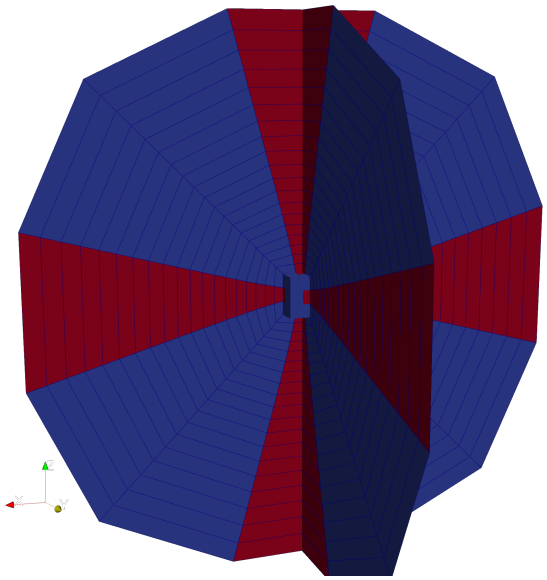


Figure 4.1: Cube with propagated roughness values. Red equals $k_s = 1.0$ and blue $k_s = 0.1$.

Overset cube

Similar to the previous test, it is used to show that the roughness values are propagated correctly for overset meshes. As described in section 3.2.1, it was necessary to refine the mesh a bit. This was needed as the *implicit hole cutting* would fail otherwise. In figure 4.2, the two overlapping meshes with the correctly propagated roughness values is shown.

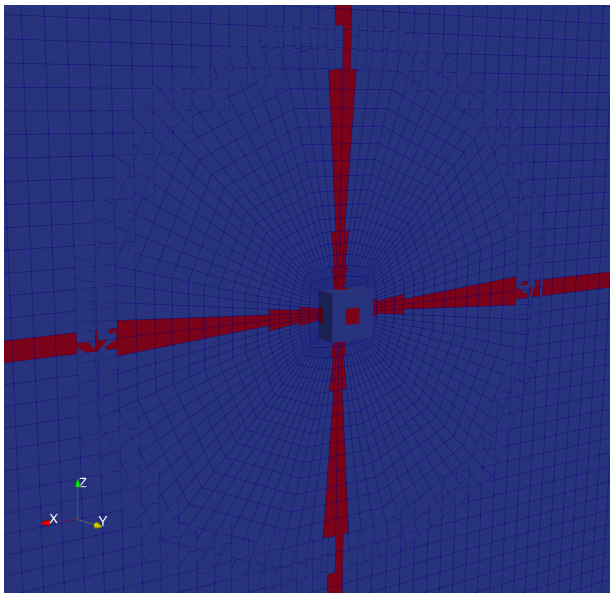


Figure 4.2: Overset cube with propagated roughness values. Red equals $k_s = 1.0$ and blue $k_s = 0.1$.

Cuboid

As described in section 3.2.1, this 6 test cases were designed to catch errors in the correlation of the surface cell to the global cell index (`gID`). As such, no figures have been generated. But the author is happy report that all tests have passed.

4.2 Flap plate at zero incidence

Grid Convergence

In figure 4.3, the grid convergence for various roughness values is shown. Additionally, grid convergence data for two different solvers from [14] is available for the skin friction coefficient. For ADflow, the expected convergence rate is 2. When computing the actual ratio using equation 2.26, one gets a ratio of approximately 1. This is less than expected and will be investigated further.

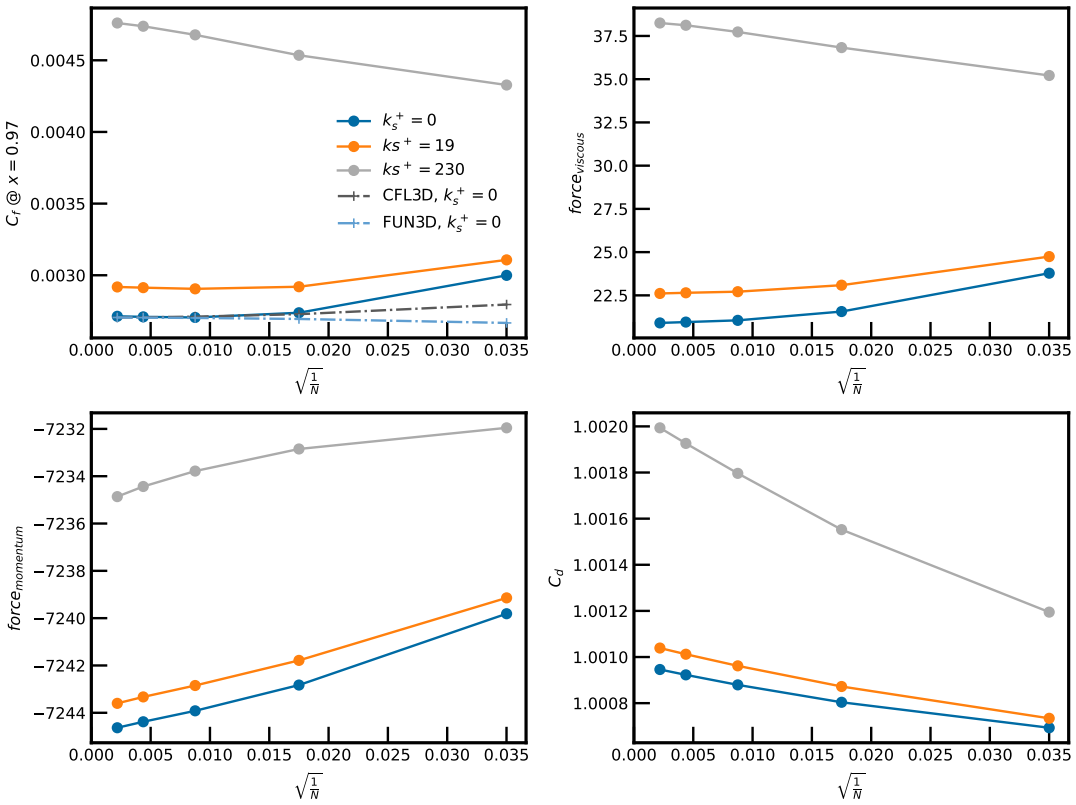


Figure 4.3: Grid convergence for various roughness values. The skin friction coefficient is overlayed with data from [14].

Clean

In figure 4.4, the skin friction coefficient over the flat plate for a roughness value of $k_s = 0$ (clean) is shown. It is compared against theory and data from the NASA TMR website [14]. When looking closely, ADflow and CFL3D agree well, but the theory is slightly off. The agreement of both solvers indicates that the SA implementation is done correctly (for a clean surface) in ADflow. The offset to theory probably means that the SA model is not completely accurate in predicting the skin friction coefficient of a flat plate. It is unclear what the dip of the skin friction at the beginning of the plate is. It may be explained as some kind of "numerical" transition. It can be observed in other plots as well, but only for ADflow.

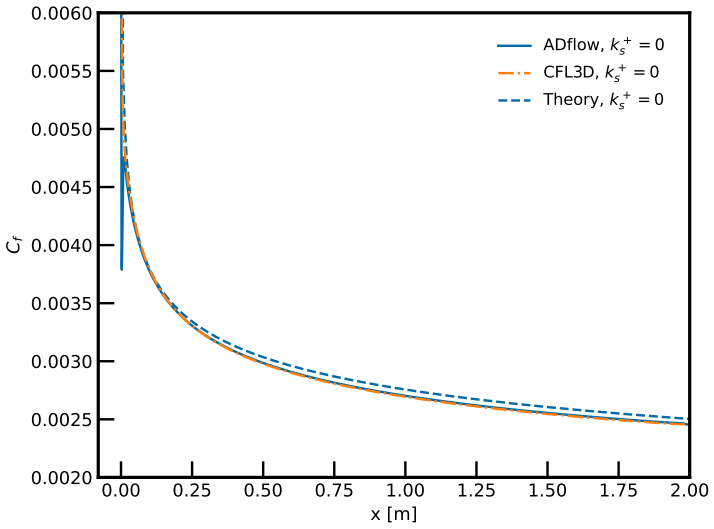


Figure 4.4: Skin friction coefficient for a roughness value of $k_s = 0$ compared against theory and data from [14].

In figure 4.5, the velocity profile at different positions on the flat plate is plotted. It is compared against theory and reference data from CFL3D. All curves agree well.

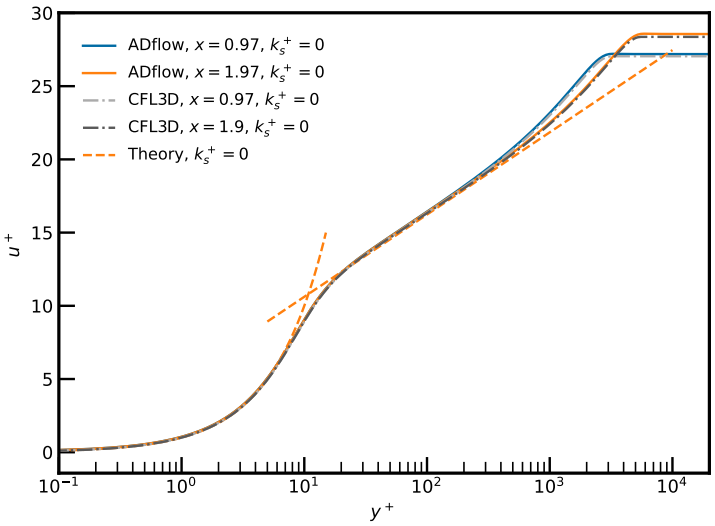


Figure 4.5: Velocity profile for a roughness value of $k_s = 0$ compared against theory and data from [14].

Blanchard

In figure 4.6, the skin friction coefficient for the clean case and a roughness value of $k_s^+ = 150$ is plotted. The clean case is compared against theory and the rough against experimental data from Blanchard's PhD thesis. But (as described in section 3.2.2), the data has been extracted from a different source [4]. The theory and clean case agree well, but the rough case does not. The "numerical" transition is more obvious for the clean case. It also exists for the rough case, but is seemingly inverted.

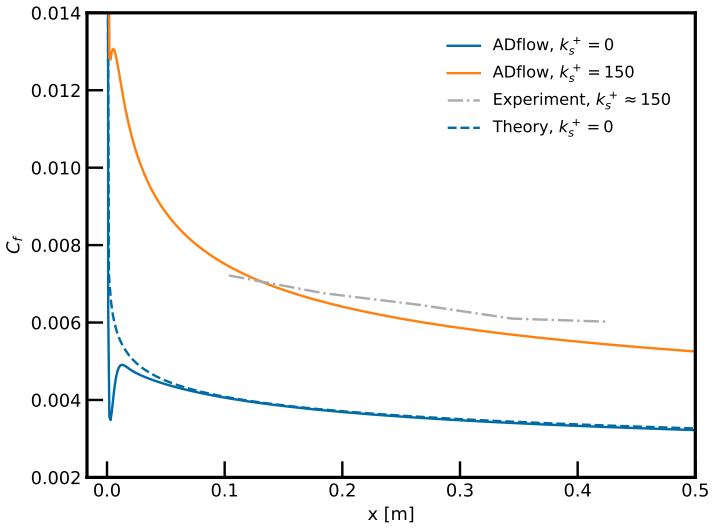


Figure 4.6: Skin friction coefficient for various roughness values compared against data from [4].

Acharya et al.

In figure 4.7, the skin friction coefficient for various roughness values are plotted against theory and two experiments from [2]. As before, the theory and clean case agree well. But the rough cases do not. First of all, the skin friction coefficient is under-predicted by ADflow. Coincidentally, the simulation for case two aligns with case one. When ignoring the offset, one can conclude that at least the predicted shape matches. The two curves do not agree at the beginning, but this may be explained through transitional effects as the SA model is fully turbulent. The "numerical" transition is again observable at the beginning of the plate.

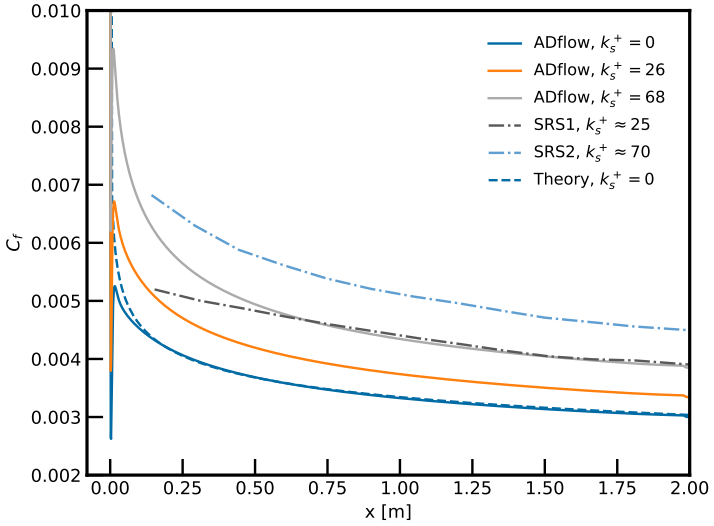


Figure 4.7: Skin friction coefficient for various roughness values compared against theory and experimental data from [2].

In figure 4.8, the velocity profile for various roughness values is plotted against theory and experimental data. Theory does agree well for the clean case, but there is a big offset for the rough case. ADflow values are not shifted enough. This indicates an underprediction of the roughness effects which is consistent with the underprediction of the skin friction coefficient. When looking at the experimental data, one can also observe a slight underprediction. But this is explainable through the use of mean values measured. This means, this curve also includes contributions from places where the boundary layer has not fully developed.

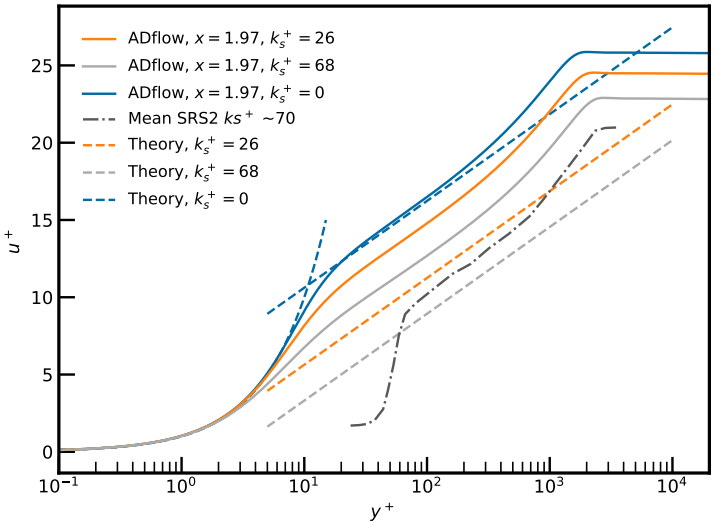


Figure 4.8: Velocity profile for various roughness values compared against theory and experimental data from [2].

SU2

To validate the implementation itself, the skin friction coefficient for various roughness values is plotted against data from SU2. For the clean case, both solvers agree well. But for the rough cases, one can observe the same offset as seen before. For ADflow, the "numerical" transition can be seen clearly at the beginning. For SU2, it is obscured, but the author reports that it does not exist.

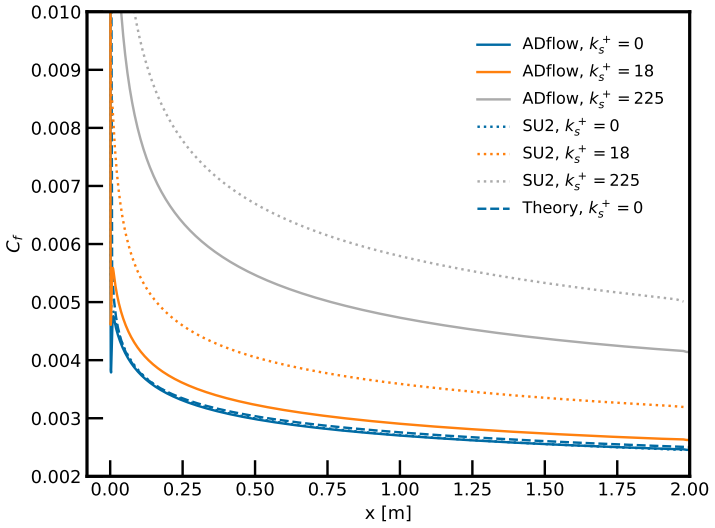


Figure 4.9: Skin friction coefficient for various roughness values compared against theory and SU2.

In figure, 4.10, the velocity profile for different roughness values is shown. For the clean case, both solvers agree well with theory. SU2 does also agree well for the rough cases although having a slight offset. For ADflow, the same shift compared to SU2 and theory is observable.

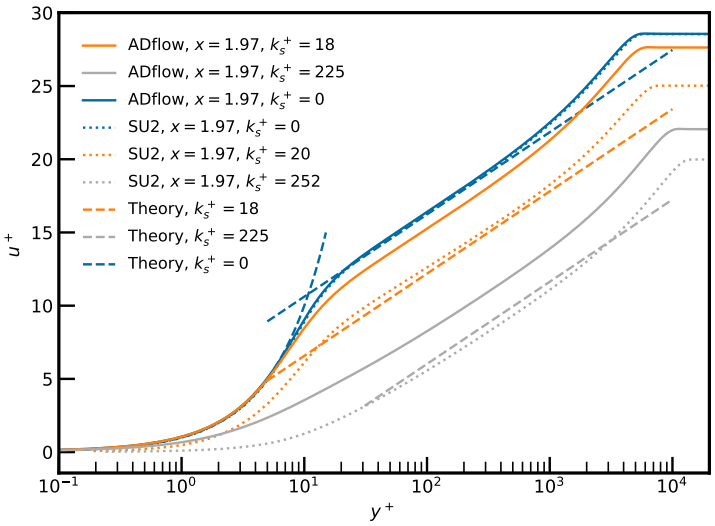


Figure 4.10: Velocity profile for various roughness values compared against theory and SU2.

4.2.1 Offset

As has been shown in the last section, the SA rough implementation in ADflow is somehow not correct. It is observable that the predicted shapes of the curves are correct, but they seem to be offset compared to experiments and SU2. At the time of writing it is not clear what is causing this and will be investigated further.

4.3 Automated tests

Reporting the results of the automated tests is easiest done in tabular form. Take a look at table 4.1.

Test-Name	Comment
`TestAdjoint_5_Rough_SA_wing`	
→ `rest_residuals`	Passes to an absolute and relative tolerance of 1e−10.
→ `rest_adjoint`	Passes to an absolute and relative tolerance of 1e−10.
`TestCmplxStep_5_Rough_SA_wing`	
→ `cmplx_test_aero_dvs`	Passes to an absolute tolerance of 5e−10 and a relative tolerance of 1e−8.
→ `cmplx_test_geom_dvs`	Fails with an absolute and relative tolerance of 5e−9 as follows:

Functional	design var.	abs. tol.	rel. tol.
C_l	span	1.64e−8	3.77e−7
C_d	span	9.39e−9	6.44e−6
C_{mz}	span	2.62e−8	4.10e−7
lift	span	6.67e−3	3.77e−7
drag	span	4.42e−4	4.73e−7
C_l	twist	7.21e−9	2.92e−7
C_l	shape	2.29e−7	3.45e−4

Test-Name	Comment
	Please note some design variables are neglected as it does not improve the overall picture.
`TestFunctionals_9_Rough_SA_wing`	
→ `rest_residuals`	Passes to an absolute and relative tolerance of 1e−10.
→ `rest_restart_read`	Passes to an absolute and relative tolerance of 1e−10.
→ `rest_functions`	Passes to an absolute and relative tolerance of 1e−9.
→ `test_forces_and_tractions`	Passes to an absolute and relative tolerance of 1e−10.
→ `test_jac_vec_prod_fwd`	Passes to an absolute and relative tolerance of 5e−9.
→ `test_jac_vec_prod_bwd`	Passes to an absolute and relative tolerance of 1e−10.
→ `test_dot_products`	Passes to an absolute and relative tolerance of 1e−10.
`TestFunctionals_10_Rough_SA_rans_tut_wing`	All tests pass. The tolerances are the same as for `TestFunctionals_9_Rough_SA_wing` listed above.

Table 4.1: Results from automated tests.

All tests passing except for the geometric design variables in `TestCmplxStep_5_Rough_SA_wing` indicates that there might be something wrong with the geometric derivatives. It is unclear what is causing this and will be investigated further. Although it is important to point out the size of the relative error of 1e−7 is already quite small. For comparison, the *finite differences* approach would only yield such accurate gradients when finding the perfect step length, which is not to be expected. Nevertheless, a relative tolerance of less than 1e−8 should be achievable with the methods employed.

5. Conclusion

During the course of this project, a modification to the Spalart Allmaras (SA) turbulence model for rough walls has been implemented. The changes have been proposed by Boeing and published by [4]. The SA model is a one-equation model that solves for the eddy viscosity. The distance to the wall is used for the turbulence length scale. The modification for rough walls simply changes this distance, meaning it shifts a virtual wall upwards. This increases the height of the boundary layer which is the primary effect of roughness.

To modify the wall distance, the roughness value of the nearest surface cell is needed. This might sound easy on first sight. But one has to keep in mind that ADflow is a parallel code. This means the whole mesh is split across different processors. Each processors has only access to mesh cells he owns. Thus the roughness value of the nearest wall for the current cell might live on a different processor and must be communicated across. Figuring out this communication was challenging and error prone. To make sure it was implemented correctly, testcases with cubes and cuboids were set up. Those tests would either fail or look wrong when the communication was not implemented correctly.

To validate the modifications itself, test cases with rough walls were setup. Those cases compare ADflow to experimental data, theory and the open source Solver SU2. They could show that the rough modification with a surface roughness of 0 would still be the same as the standard SA model. Unfortunately, they also made it clear that current implementation has a bug where the skin friction coefficient and the velocity profile are wrongly shifted for rough surfaces. This means, the predicted shape looks right, but the predicted values are not. It is unclear what is causing this. It will be investigated in the future.

The before mentioned test cases also showed that the expected grid convergence rate of 2 was not achieved. The real value lies more around 1. This will also be investigated further.

Lastly, automated tests were setup to make sure future changes do not break existing features. This concept is know as regression tests. In ADflow, the automated tests are also mixed with unit tests which prove a new feature is working. In this context, the gradients of the modified SA model was compared against complex step. Unfortunately, they do not completely agree and are off by a relative tolerance of 1e−7. It is unclear what was causing this and will be investigated further.

To conclude, a solid base was created. Although some bugs are still present that need further work. Luckily, they do not affect an optimization in the following sense: (1) The lower than expected grid convergence does not directly affect an optimization. (2) The offset of the gradients is so small that an optimization would still converge, maybe a bit slower. (3) The offset of the roughness effects can be compensated with a higher than expected surface roughness value. This is possible because the predicted shape of the roughness effects match. And lastly, in optimization, it is more important to take effects of roughness into account than accurately predicting those. After all, it is extremely difficult to define a valid roughness value for e.g a soiled wind turbine blades after 20 years of service.

List of Figures

List of Tables

Bibliography

[1] Fluid Mechanics 101. *[CFD] the spalart-allmaras turbulence model*. Feb. 2020. URL: https://www.youtube.com/watch?v=Xivc0EIGFQw.

[2] M. Acharya, J. Bornstein, and M. P. Escudier. "Turbulent boundary layers on rough surfaces". In: *Experiments in Fluids* 4.1 (Jan. 1986), pp. 33–47. ISSN: 1432-1114. DOI: 10.1007/BF00316784. URL: https://doi.org/10.1007/BF00316784.

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

[4] B. Aupoix and P.R. Spalart. "Extensions of the Spalart–Allmaras turbulence model to account for wall roughness". In: *International Journal of Heat and Fluid Flow* 24.4 (2003). Selected Papers from the Fifth International Conference on Engineering Turbulence Modelling and Measurements, pp. 454–462. ISSN: 0142-727X. DOI: https://doi.org/10.1016/S0142-727X(03)00043-2. URL: https://www.sciencedirect.com/science/article/pii/S0142727X03000432.

[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] Thomas D. Economon et al. "SU2: An Open-Source Suite for Multiphysics Simulation and Design". In: *AIAA Journal* 54.3 (2016), pp. 828–846. DOI: 10.2514/1.J053813. eprint: https://doi.org/10.2514/1.J053813. URL: https://doi.org/10.2514/1.J053813.

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

[8] Laurent Hascoët and Valérie Pascual. "The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification". In: *ACM Trans. Math. Softw.* 39 (May 2013), 20:1–20:43. DOI: 10.1145/2450153.2450158.

[9] Georgi Kalitzin et al. "Near-wall behavior of RANS turbulence models and implications for wall functions". In: *Journal of Computational Physics* 204.1 (2005), pp. 265–291. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2004.10.018. URL: https://www.sciencedirect.com/science/article/pii/S0021999104004164.

[10] 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.

[11] 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.

[12] 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.

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

[14] Christopher Rumsey. *2d zero pressure gradient flat plate verification - Intro page*. URL: https://turbmodels.larc.nasa.gov/flatplate.html.

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

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

[17] Ney Secco et al. “Efficient Mesh Generation and Deformation for Aerodynamic Shape Optimization”. In: *AIAA Journal* (2021). DOI: 10.2514/1.J059491.

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