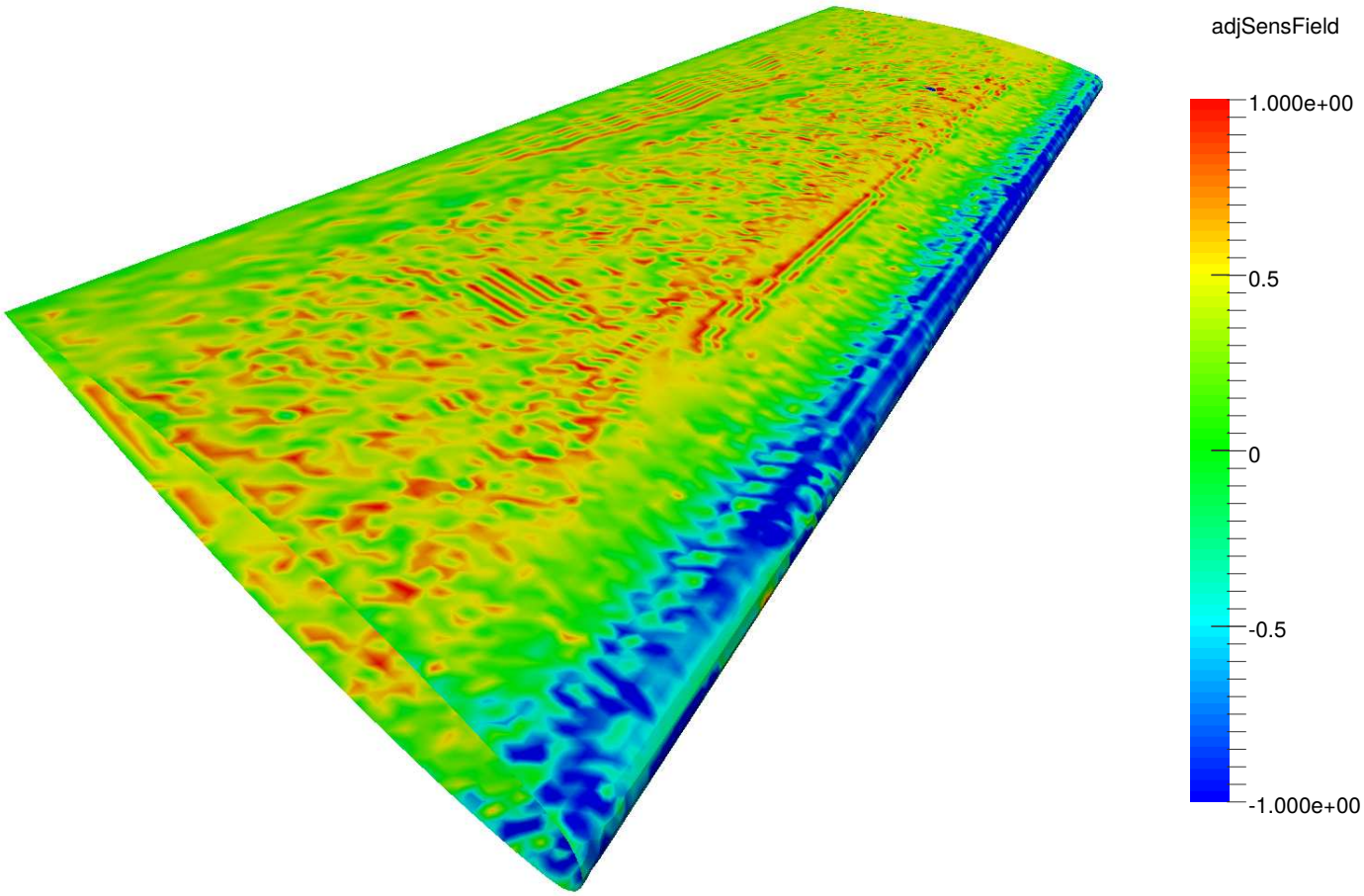


MASTER THESIS

Discrete Adjoint Approach to the Spalart-Allmaras Turbulence Model for Incompressible Flow in OpenFOAM

Dennis Kasper

Chair of Structural Analysis
Prof. Dr.-Ing. K.-U. Bletzinger
Technische Universität München

Chair of Structural Analysis
Technische Universität München

Discrete Adjoint Approach to the Spalart-Allmaras Turbulence Model for Incompressible Flow in OpenFOAM

Dennis Kasper
Mat.-Nr. 03653160

30. September 2016

Master Thesis in Computational Mechanics

Reza Najian M.Sc.
Univ.-Prof. Dr.-Ing. Kai-Uwe Bletzinger

Abstract

A discrete adjoint solver for the Spalart-Allmaras turbulence model equation is implemented in OpenFOAM v3.0.1 to compute shape sensitivities in context of aerodynamic shape optimization. The focus lies on incompressible and steady flow for high Reynolds numbers, governed by the Reynolds-Averaged Navier-Stokes equation completed with the Spalart-Allmaras turbulence model. The discrete adjoint equation is derived using the method of Lagrange multiplier. The Spalart-Allmaras turbulence model equation and its derivative w.r.t. the Spalart-Allmaras variable is presented. The implementation of the adjoint solver is documented in detail. Three examples are presented in order to verify the discrete adjoint solver against a total finite difference approach, a channel flow, a flow over the NACA 2412 airfoil, and a flow over the ONERA M6 wing. Furthermore, a discussion on the implicit Laplace operator for different OpenFOAM versions and a guideline to implement a new turbulence model in OpenFOAM v3.01 is provided.

Acknowledgments

First of all, I would like to thank my supervisor Reza Najian M.Sc., for giving me the opportunity to write this thesis, for his guidance and suggestions on this work as well as proofreading.

I also would like to thank Univ.-Prof. Dr.-Ing. Kai-Uwe Bletzinger for the evaluation of this thesis.

Last but not least I would like to thank my mother for giving me the possibility to study.

Contents

Nomenclature

Abbreviations

AoA	Angle of Attack
BCs	Boundary Conditions
CFD	Computationl Fluid Dynamics
DVs	Design Variables
Eq.	Equation
Fig.	Figure
FVM	Finite Volume Method
LHS	Left Hand Side
Lst.	Listing
PDE	Partial Differential Equation
RANS	Reynolds-Averaged Navier-Stokes equations
Re	Reynolds number
RHS	Right Hand Side
SIMPLE	Semi-Implicit Method for Pressure Linked Equations
Tbl.	Table
w.r.t.	with respect to

Greek letters

α	Angle of Attack
β	Turbulent viscosity ratio
$\boldsymbol{\Omega}$	Mean rotation-rate tensor
$\boldsymbol{\psi}_{\mathrm{u}}$	Vector of discrete RANS adjoint variables
$\boldsymbol{\psi}_{\tilde{\nu}}$	Vector of discrete Spalart-Allmaras adjoint variables
$\boldsymbol{\tau}$	Specific Reynolds stress tensor
$\tilde{\boldsymbol{\nu}}$	Vector of discrete Spalart-Allmaras state variables
χ	Spalart-Allmaras function
$\Delta\boldsymbol{x}$	Perturbation Vector
δ_{ij}	Kronecker delta
κ	Spalart-Allmaras constant
ν	Kinematic molecular viscosity
ν_{t}	Kinematic turbulent or eddy viscosity
Ω	Magnitude of the mean rotation-rate tensor
Ω_{ij}	ij-component of the mean rotation-rate tensor $\boldsymbol{\Omega}$
σ	Spalart-Allmaras constant
τ_{ij}	ij-th component of the specific Reynolds stress tensor $\boldsymbol{\tau}$
$\tilde{\nu}$	Spalart-Allmaras working variable
$\tilde{\nu}_{\mathrm{eff}}$	Effective kinematic turbulent or eddy viscosity

Latin letters

$\boldsymbol{g}$	Vector of inequality contrains

$\boldsymbol{h}$	Vector of equality contrains
$\boldsymbol{n}$	Unit normal vector
$\boldsymbol{R}_{\mathrm{u}}$	Vector of discrete RANS residuals
$\boldsymbol{R}_{\tilde{\nu}}$	Vector of discrete Spalart-Allmaras residuals
$\boldsymbol{S}$	Mean strain-rate tensor
$\boldsymbol{s}$	Instantaneous strain-rate tensor
$\boldsymbol{U}$	Averaged or mean velocity vector
$\boldsymbol{x}$	Position vector
$\tilde{P}$	Modified pressure
$\tilde{S}$	Modified vorticity
c_{b1}	Spalart-Allmaras constant
c_{b2}	Spalart-Allmaras constant
c_{s}	Spalart-Allmaras constant
c_{v1}	Spalart-Allmaras constant
c_{w1}	Spalart-Allmaras constant
c_{w2}	Spalart-Allmaras constant
c_{w3}	Spalart-Allmaras constant
d	Distance to the closest wall
f	Objective function
f_{t1}	Spalart-Allmaras function
f_{t2}	Spalart-Allmaras function
f_{v1}	Spalart-Allmaras function
f_{v2}	Spalart-Allmaras function
f_{w}	Spalart-Allmaras function
g	Spalart-Allmaras function
g_j	j-th inequality constrain
h_k	k-th equality constrain
k	Kinetic energy
L	Lagrange function
P	Averaged or mean pressure
p	Instantaneous pressure
r	Spalart-Allmaras function
S_{ij}	ij-component of the mean strain-rate tensor $\boldsymbol{S}(\boldsymbol{x}, t)$
s_{ij}	ij-component of the instantaneous strain-rate tensor $\boldsymbol{s}(\boldsymbol{x}, t)$
t	Time
u_i'	Fluctuating velocity component in i-direction
U_i	Averaged or mean velocity component in i-direction
u_i	Instantaneous velocity component in i-direction
x_i	i-th component of position vector $\boldsymbol{x}$

Operators

$\overline{(\bullet)}$	Time-Average operator
$\partial(\bullet)/\partial\boldsymbol{n}$	Normal derivative $\nabla(\bullet)\cdot\boldsymbol{n}$
$\partial(\bullet)/\partial t$	Partial derivative w.r.t. time
$\partial(\bullet)/\partial x_i$	Partial derivative w.r.t. the i-coordinate direction
$\max((\bullet),(\bullet))$	Maximum function
$\min((\bullet),(\bullet))$	Minimum function

1 Motivation

Fluid dynamic optimal design problems are ubiquitous in the aerospace, automotive and marine industry. The developed methodologies range from bionic algorithms over gradient based search to surrogate modeling. In the case of gradient based optimization, the so called adjoint method has long been identified as the method of choice for the computation of shape sensitivities. It allows for computing the complete gradient with an effort of only two solver calls, i.e. primal and adjoint, almost independent of the search space dimension. The success of this method is manifested by the large number of publications on adjoint based optimization in fluid dynamics and also structure mechanics [Oth08].

1.1 Discrete and Continuous Adjoint Approach

The adjoint equations can be derived through either the continuous, or the discrete adjoint approach. In the continuous approach, the adjoint equations are derived as PDEs and then discretized. In the discrete approach, the discrete adjoint equations results from the discretized PDEs [ZPGO09]. The two alternative ways to produce the discrete adjoint equations are illustrated in Fig. 1.1. Both methods have been developed over the past years, one can find a

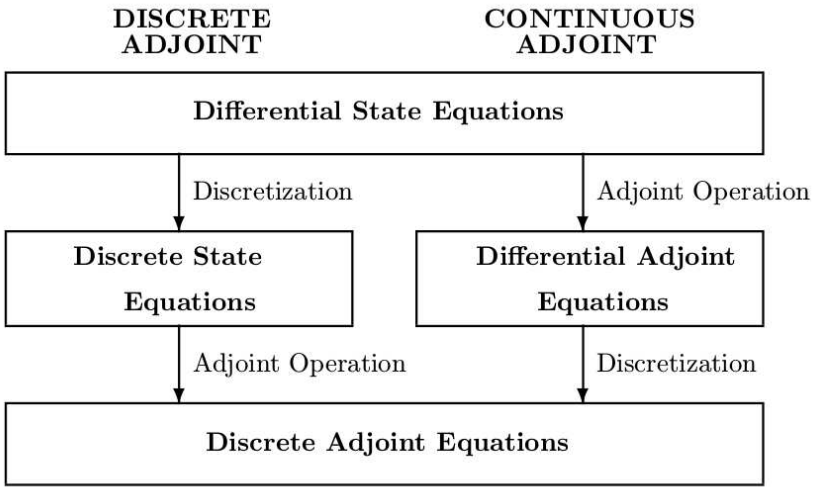


Figure 1.1: Discrete vs Continuous Adjoint approach to derive the adjoint equations [TJ08].

summary of the archived work as well as a discussion on there advantages and disadvantages in e.g. [BOCPZ12] or [Oth08].

In this thesis the main focus lies on turbulent, incompressible and steady flow governed by the Reynolds-Averaged Navier-Stokes equations closed with the Spalart-Allmaras turbulence model equation. In turbulent flows, the mean flow equations are coupled with the turbulent model one and both must be considered as state equations. The adjoint approach should take into account variation in the mean flow and the turbulence variable. However, the literature survey reveals that the latter is often neglected. This approximation is known as frozen turbulence assumption. Fig. 1.2 shows the sensitivities for a total pressure loss in a turbulent S-shape

duct flow. For details of the problem consult [ZPGO09]. The sensitivities are plotted for the

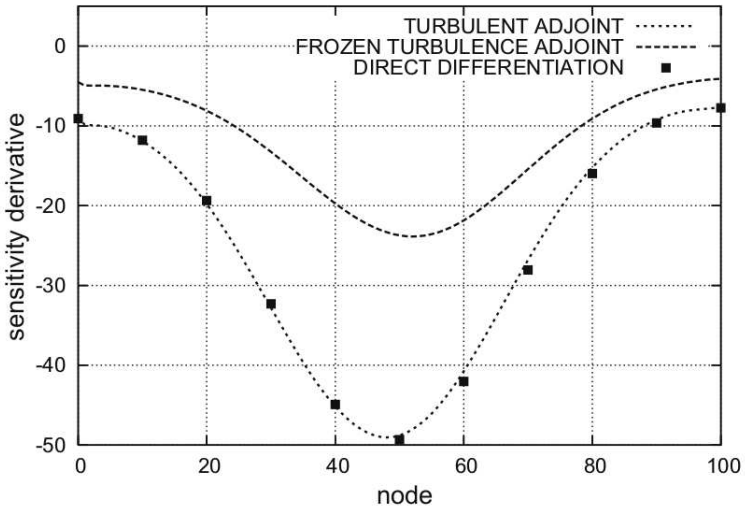


Figure 1.2: Sensitivity derivative computed with (FROZEN TURBULENT ADJOINT) and without (TURBULENT ADJOINT) frozen turbulence assumption and comparison to the reference solution (DIRECT DIFFERENTIATION) [ZPGO09].

turbulent adjoint, the frozen turbulent adjoint and for the direct differentiation approach. The direct differentiation appraoch serves as a reference solution. One can observe, that with the frozen turbulent assumption the sensitivities are underestimated by a factor of approximately two. Therefore, it is obvious that one should consider the variation of the turbulent model equation in oder to compute correct sensitivities.

The scope of this thesis is to set up a discrete adjoint approach to the Spalart-Allmaras turbulence model equation and verify it against a total finite difference approach. OpenFOAM (for "Open source Field Operation And Manipulation") is a C++ toolbox for the development of customized numerical solvers, and pre-/post-processing utilities for the solution of continuum mechanics problems, including computational fluid dynamics (CFD). The code is released as free and open source software under the GNU General Public License.[1] OpenFOAM has been chosen due to its ability to implement PDEs in a relatively easy way, it allows access to the source code and due to its popularity in research and industry.

1.2 Outline of this Thesis

In chapter 2 the governing fluid flow equations for turbulent, incompressible and steady flow are derived. The RANS equations are closed with the Spalart-Allmaras turbulence model.

In chapter 3 the shape sensitivity equations are derived. The method of Lagrange multiplier is applied.

In chapter 4 the Spalart-Allmaras turbulence model is presented. The Spalart-Allmaras adjoint equation is derived.

In chapter 5 the CFD solution is verified.

In chapter 6 the implementation of the adjoint solver in OpenFOAM v3.0.1 is presented.

In chapter 7 the verification of the adjoint solver with the help of three examples is presented.

In chapter 8 the discrepancy of the Laplace operator is discussed.

[1] `https://en.wikipedia.org/wiki/OpenFOAM` (21.09.2016)

In chapter 9 a guideline is presented on how to implement a turbulence model in OpenFOAM v3.0.1

In chapter 10 the results are summarized.

In chapter 11 an outlook on further topics is given.

2 Reynolds-Averaged Navier-Stokes Equations and Turbulence Model

In the following, the governing flow equations as well as the turbulence model used in this thesis are presented. The intention is to provide the necessary equations and comment on important details rather than deriving them in a rigorous mathematical manner. For a detailed derivation, one can consult e.g. [Wil06].

In order to reduce the complexity of the problem, the following assumptions are made. The flow is considered to be incompressible. Physical properties are assumed to be constant, i.e. the density ρ and the kinematic molecular viscosity ν are constant. For a Newtonian fluid, the equations for conservation of mass and linear momentum in the three dimensional Cartesian reference frame are

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{2.1}$$

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(2\nu s_{ij}\right) \tag{2.2}$$

for $i = \{1, 2, 3\}$, $\forall \boldsymbol{x} \in \mathbb{R}^3$ and $\forall t \in \mathbb{R}_+$. $u_i(\boldsymbol{x}, t)$ and x_i are the i-th components of the vector of instantaneous velocity $\boldsymbol{u}(\boldsymbol{x}, t)$ and position $\boldsymbol{x}$, $p(\boldsymbol{x}, t)$ is the instantaneous pressure, t the time and $s_{ij}(\boldsymbol{x}, t) = \frac{1}{2}\left(\partial u_i/\partial x_j + \partial u_j/\partial x_i\right)$ the ij-th component of the strain-rate tensor $\boldsymbol{s}(\boldsymbol{x}, t)$. If not otherwise stated, Einstein summation convention is applied.

Considering stationary turbulent flow and using Reynolds decomposition, i.e. $u_i(\boldsymbol{x}, t) = U_i(\boldsymbol{x}) + u_i'(\boldsymbol{x}, t)$ and $p(\boldsymbol{x}, t) = P(\boldsymbol{x}) + p'(\boldsymbol{x}, t)$, the process of time averaging Eq. (2.1) and (2.2) results in the RANS equations

$$\frac{\partial U_i}{\partial x_i} = 0 \tag{2.3}$$

$$U_j \frac{\partial U_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial P}{\partial x_i} + \frac{\partial}{\partial x_j}\left(2\nu S_{ij} - \overline{u_i' u_j'}\right) \tag{2.4}$$

where $U_i(\boldsymbol{x})$ is the i-th mean velocity component of the mean velocity vector $\boldsymbol{U}(\boldsymbol{x})$, $P(\boldsymbol{x})$ the mean pressure, $S_{ij}(\boldsymbol{x}) = \frac{1}{2}\left(\partial U_i/\partial x_j + \partial U_j/\partial x_i\right)$ the ij-th component of the mean strain-rate tensor $\boldsymbol{S}(\boldsymbol{x})$ and the quantity $\tau_{ij}(\boldsymbol{x}, t) \coloneqq -\overline{u_i' u_j'}$ the ij-th component of the specific symmetric Reynolds stress tensor denoted by $\boldsymbol{\tau}(\boldsymbol{x}, t)$. Due to the Reynolds stresses, six new unknowns are introduced into the system but no additional equations. In total there are ten unknowns and four equations, therefore the system is not yet closed. This is commonly known as the closure problem in turbulence modeling.

It is possible to derive transport equations for each Reynolds stress component, however, by doing so, we introduce even more unknown relations. In order to close the system, the Reynolds stress tensor needs to be modeled. One common approach is to use the hypothesis by Boussinesq, how states that the Reynolds stress tensor is related to mean strain-rate tensor by

$$-\overline{u_i' u_j'} = 2\nu_\mathrm{t} S_{ij} - \frac{2}{3} k \delta_{ij} \tag{2.5}$$

where ν_t is the turbulent or eddy viscosity assumed as an isotropic scalar field quantity, $k =$

$\frac{1}{2}\overline{u_i' u_i'} = \frac{1}{2}(\overline{u_1'^2} + \overline{u_2'^2} + \overline{u_3'^2})$ is the specific turbulent kinetic energy and δ_{ij} the Kronecker delta.[2] Note, that the trace of the Reynolds stress tensor is proportional to the specific kinetic energy, i.e.

$$\tau_{ii} = -\overline{u_i' u_i'} = -2k \tag{2.6}$$

The problem of finding six unknown Reynold stress components, is reduced to find only one scalar field quantity ν_t. There are numerous turbulence models with different levels of complexity and research is still going on since turbulent flow is highly demanded in engineering. For this work, the Spalart-Allmaras turbulence model is chosen which introduces one more transport equation to the RANS equations and therefore closes the system.

Combining Eq. (2.4) and (2.5), one can write

$$U_j \frac{\partial U_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \tilde{P}}{\partial x_i} + \frac{\partial}{\partial x_j}\left(2\left(\nu + \nu_\mathrm{t}\right) S_{ij}\right) \tag{2.7}$$

where the pressure is modified by the turbulent kinetic energy, i.e. $\tilde{P} = P + \rho\frac{2}{3}k$. However, for the Spalart-Allmaras turbulence model, the turbulent kinetic energy is not calculated and therefore the last term in Eq. (2.5) is ignored when estimating the Reynolds stresses resulting in

$$U_j \frac{\partial U_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial P}{\partial x_i} + \frac{\partial}{\partial x_j}\left(2\left(\nu + \nu_\mathrm{t}\right) S_{ij}\right) \tag{2.8}$$

Finally, the set of governing equations for mass, momentum and turbulent viscosity are

$$\frac{\partial U_i}{\partial x_i} = 0 \tag{2.9}$$

$$U_j \frac{\partial U_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial P}{\partial x_i} + \frac{\partial}{\partial x_j}\left(2\left(\nu + \nu_\mathrm{t}\right) S_{ij}\right) \tag{2.10}$$

$$\begin{aligned} U_j \frac{\partial \tilde{\nu}}{\partial x_j} = c_{\mathrm{b}1}\left(1 - f_{\mathrm{t}2}\right)\tilde{S}\tilde{\nu} + \frac{1}{\sigma}\left\{\frac{\partial}{\partial x_j}\left[\left(\nu + \tilde{\nu}\right)\frac{\partial \tilde{\nu}}{\partial x_j}\right] + c_{\mathrm{b}2}\frac{\partial \tilde{\nu}}{\partial x_i}\frac{\partial \tilde{\nu}}{\partial x_i}\right\} \\ - \left(c_{\mathrm{w}1} f_\mathrm{w} - \frac{c_{\mathrm{b}1}}{\kappa^2} f_{\mathrm{t}2}\right)\left(\frac{\tilde{\nu}}{d}\right)^2 + f_{\mathrm{t}1}\Delta U^2 \end{aligned} \tag{2.11}$$

where Eq. (2.11) is the transport equation for the Spalart-Allmaras variable $\tilde{\nu}$ related to the turbulent viscosity by $\nu_\mathrm{t} = f_{\mathrm{v}1}\tilde{\nu}$. The last term on the RHS in Eq. (2.11) is the trip term, used in moderate Reynolds number flows to capture the transition point between laminar and turbulent flow. However, for fully turbulent flow this term is neglected. The Spalart-Allmaras turbulence model is described in more detail in chapter 4.

[2] $\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$

3 Discrete Adjoint based Shape Sensitivity Analysis

This chapter presents the main equations used in this thesis. The intention is to derive the discrete adjoint based sensitivity equations rather then presenting a discussion on optimization in general. The focus lies on node based shape sensitivities. In the following the numerator layout matrix notation is used.[3]

3.1 General Shape Optimization Problem

Shape optimization problems can be stated in the most general form as

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^n, \quad n \in \mathbb{N}_+ \\ \text{s.t.} \quad & g_j(\boldsymbol{x}) \leq 0, \quad j = 1, \ldots, p \\ & h_k(\boldsymbol{x}) = 0, \quad k = 1, \ldots, q \end{aligned} \tag{3.1}$$

where f, g_j and h_k are the objective function, the inequality and the equality constrains, respectively. They are functions of the n optimization variables x_i, $i = 1, \ldots, n$ which are arranged in the vector of DVs $\boldsymbol{x} \in \mathbb{R}^n$. In shape optimization context these are the coordinates of some position vectors. Arranging the constraints in vectors $\boldsymbol{g}$ and $\boldsymbol{h}$, Eq. (3.1) can be written in its most compact form

$$\min_{\boldsymbol{x}} \{ f(\boldsymbol{x}) \,|\, \boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{0}, \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}, \boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^p, \boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^q, \boldsymbol{x} \in \mathbb{R}^n, p, q, n \in \mathbb{N}_+ \} \tag{3.2}$$

3.2 Lagrange Multiplier Method

One common approach to solve problems in shape optimization is the method of Lagrange multipliers. Consider the following equality constrained shape optimization problem

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}, \tilde{\boldsymbol{\nu}}(\boldsymbol{x})), \quad \boldsymbol{x} \in \mathbb{R}^n, \quad n \in \mathbb{N}_+ \\ \text{s.t.} \quad & \boldsymbol{R}_{\tilde{\nu}}(\boldsymbol{x}, \tilde{\boldsymbol{\nu}}(\boldsymbol{x})) = \boldsymbol{0} \end{aligned}$$

where f is a scalar valued (possible non-linear) objective function with explicit dependency on the vector of DVs $\boldsymbol{x}$, and implicit dependency via the vector of discrete Spalart-Allmaras state variables $\tilde{\boldsymbol{\nu}}(\boldsymbol{x})$. $\boldsymbol{R}_{\tilde{\nu}}$ is the vector of residuals, which in this context is the discrete residual form of the Spalart-Allmaras Eq. (2.11). It is assumed, that the velocity U_i is known, therefore $\tilde{\nu}$ is the only unknown.

Using the method of Lagrange multipliers, the Lagrange function reads

$$L = f(\boldsymbol{x}, \tilde{\boldsymbol{\nu}}(\boldsymbol{x})) + \boldsymbol{\psi}_{\tilde{\nu}}^{\mathrm{T}} \boldsymbol{R}_{\tilde{\nu}}(\boldsymbol{x}, \tilde{\boldsymbol{\nu}}(\boldsymbol{x})) \tag{3.3}$$

where $\boldsymbol{\psi}_{\tilde{\nu}}$ is the vector of Lagrange multipliers or adjoint variables.

[3] `https://en.wikipedia.org/wiki/Matrix_calculus#Layout_conventions` (21.07.2016)

The response sensitivity analysis is given by the total derivative, also called gradient ($\nabla_{\boldsymbol{x}} L$), of the Lagrange function w.r.t. the DVs $\boldsymbol{x}$, applying the chain rule, it follows

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{x}} &= \frac{\partial f}{\partial \boldsymbol{x}} + \frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}}\frac{\mathrm{d}\tilde{\boldsymbol{\nu}}}{\mathrm{d}\boldsymbol{x}} + \boldsymbol{\psi}_{\tilde{\nu}}^{\mathrm{T}}\left(\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}}\frac{\mathrm{d}\tilde{\boldsymbol{\nu}}}{\mathrm{d}\boldsymbol{x}}\right) \\ &= \frac{\partial f}{\partial \boldsymbol{x}} + \boldsymbol{\psi}_{\tilde{\nu}}^{\mathrm{T}}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{x}} + \left(\frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}} + \boldsymbol{\psi}_{\tilde{\nu}}^{\mathrm{T}}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}}\right)\frac{\mathrm{d}\tilde{\boldsymbol{\nu}}}{\mathrm{d}\boldsymbol{x}}\end{aligned} \tag{3.4}$$

The computation of $\mathrm{d}\tilde{\boldsymbol{\nu}}/\mathrm{d}\boldsymbol{x}$ for node based shape optimization and sensitivity analysis is very difficult and computational expensive. Since the adjoint variables $\boldsymbol{\psi}_{\tilde{\nu}}$ are unknown a priori, the computation can be avoided by choosing $\boldsymbol{\psi}_{\tilde{\nu}}$ such that the term in brackets in Eq. (3.4) becomes zero, i.e.

$$\begin{aligned}\frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}} + \boldsymbol{\psi}_{\tilde{\nu}}^{\mathrm{T}}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}} &= \boldsymbol{0}^{\mathrm{T}} \\ \boldsymbol{\psi}_{\tilde{\nu}}^{\mathrm{T}}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}} &= -\frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}} \\ \left[\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}}\right]^{\mathrm{T}}\boldsymbol{\psi}_{\tilde{\nu}} &= -\left[\frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}}\right]^{\mathrm{T}}\end{aligned} \tag{3.5}$$

Eq. (3.5) is known as the discrete adjoint equation. The matrix on the LHS is also called Jacobi matrix. Once the adjoint variables are known, the sensitivities can be computed as follows

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{x}} = \frac{\partial f}{\partial \boldsymbol{x}} + \boldsymbol{\psi}_{\tilde{\nu}}^{\mathrm{T}}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{x}} \tag{3.6}$$

The adjoint approach is efficient if the number of DVs is significantly larger then the number of objective functions. For aerodynamic shape optimization problems this is usually the case since the DVs are the nodal positions of a surface subjected to a fluid flow, and commonly only one or a few objective functions are considered.

The second term of the RHS in Eq. (3.6) can be approximated with the help of a finite difference approach. Using a central finite difference scheme

$$\boldsymbol{\psi}_{\tilde{\nu}}^{\mathrm{T}}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{x}} \approx \boldsymbol{\psi}_{\tilde{\nu}}^{\mathrm{T}}\frac{\boldsymbol{R}_{\tilde{\nu}}(\boldsymbol{x}+\boldsymbol{\Delta x}) - \boldsymbol{R}_{\tilde{\nu}}(\boldsymbol{x}-\boldsymbol{\Delta x})}{2\Delta \boldsymbol{x}} \tag{3.7}$$

Eq. (3.6) can be computed. This approach is usually sensitive w.r.t. to the step size $\Delta \boldsymbol{x}$, therefore a step size study is needed.

3.3 Shape Sensitivity Analysis for a Fully Coupled Turbulent Flow

Adjoint based sensitivity analysis in fluid dynamic context often refers to the RANS equations completed with a turbulence model. It is common practice to neglect shape variation of turbulent viscosity in order to simplify the computation. However, this is only an approximation known as frozen turbulence assumption. If the spatial variation of turbulent viscosity in the adjoint problem is considered, the adjoint variables must also be computed. Considering the Spalart-Allmaras turbulence model, the shape sensitivity analysis can be formulated as follows

$$\begin{aligned}\min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}), \tilde{\boldsymbol{\nu}}(\boldsymbol{x})), \quad \boldsymbol{x} \in \mathbb{R}^n, \quad n \in \mathbb{N}_+ \\ \text{s.t.} \quad & \boldsymbol{R}_{\mathrm{u}}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}), \tilde{\boldsymbol{\nu}}(\boldsymbol{x})) = \boldsymbol{0} \\ & \boldsymbol{R}_{\tilde{\nu}}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}), \tilde{\boldsymbol{\nu}}(\boldsymbol{x})) = \boldsymbol{0}\end{aligned}$$

where the residuals are split into two parts, RANS $\boldsymbol{R}_\mathrm{u}$ and Spalart-Allmaras $\boldsymbol{R}_{\tilde{\nu}}$ respectively. The vectors of state variables are $\boldsymbol{u}(\boldsymbol{x}) = \begin{bmatrix}\boldsymbol{U}(\boldsymbol{x}) & \boldsymbol{P}(\boldsymbol{x})\end{bmatrix}^\mathrm{T}$ and $\tilde{\boldsymbol{\nu}}(\boldsymbol{x})$.
Using the method of Lagrange multipliers, the Lagrange function reads

$$L = f(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}), \tilde{\boldsymbol{\nu}}(\boldsymbol{x})) + \boldsymbol{\psi}_\mathrm{u}^\mathrm{T} \boldsymbol{R}_\mathrm{u}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}), \tilde{\boldsymbol{\nu}}(\boldsymbol{x})) + \boldsymbol{\psi}_{\tilde{\nu}}^\mathrm{T} \boldsymbol{R}_{\tilde{\nu}}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}), \tilde{\boldsymbol{\nu}}(\boldsymbol{x})) \tag{3.8}$$

where $\boldsymbol{\psi}_\mathrm{u}$ and $\boldsymbol{\psi}_{\tilde{\nu}}$ are the vectors of Lagrange multipliers or adjoint variables.
The response sensitivity analysis is given by the total derivative of the Lagrange function w.r.t. the DVs $\boldsymbol{x}$, applying the chain rule it follows

$$\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{x}} &= \frac{\partial f}{\partial \boldsymbol{x}} + \frac{\partial f}{\partial \boldsymbol{u}}\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{x}} + \frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}}\frac{\mathrm{d}\tilde{\boldsymbol{\nu}}}{\mathrm{d}\boldsymbol{x}} + \boldsymbol{\psi}_\mathrm{u}^\mathrm{T}\left(\frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \boldsymbol{u}}\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{x}} + \frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \tilde{\boldsymbol{\nu}}}\frac{\mathrm{d}\tilde{\boldsymbol{\nu}}}{\mathrm{d}\boldsymbol{x}}\right) \\
&\qquad + \boldsymbol{\psi}_{\tilde{\nu}}^\mathrm{T}\left(\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{u}}\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{x}} + \frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}}\frac{\mathrm{d}\tilde{\boldsymbol{\nu}}}{\mathrm{d}\boldsymbol{x}}\right) \\
&= \frac{\partial f}{\partial \boldsymbol{x}} + \boldsymbol{\psi}_\mathrm{u}^\mathrm{T}\frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \boldsymbol{x}} + \boldsymbol{\psi}_{\tilde{\nu}}^\mathrm{T}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{x}} + \left(\frac{\partial f}{\partial \boldsymbol{u}} + \boldsymbol{\psi}_\mathrm{u}^\mathrm{T}\frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \boldsymbol{u}} + \boldsymbol{\psi}_{\tilde{\nu}}^\mathrm{T}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{u}}\right)\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{x}} \\
&\qquad + \left(\frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}} + \boldsymbol{\psi}_\mathrm{u}^\mathrm{T}\frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \tilde{\boldsymbol{\nu}}} + \boldsymbol{\psi}_{\tilde{\nu}}^\mathrm{T}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}}\right)\frac{\mathrm{d}\tilde{\boldsymbol{\nu}}}{\mathrm{d}\boldsymbol{x}}
\end{aligned} \tag{3.9}$$

which can be written after some matrix algebra

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{x}} = \frac{\partial f}{\partial \boldsymbol{x}} + \boldsymbol{\psi}_\mathrm{u}^\mathrm{T}\frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \boldsymbol{x}} + \boldsymbol{\psi}_{\tilde{\nu}}^\mathrm{T}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{x}} + \left(\begin{bmatrix}\left[\frac{\partial f}{\partial \boldsymbol{u}}\right]^\mathrm{T} \\ \left[\frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}}\right]^\mathrm{T}\end{bmatrix} + \begin{bmatrix}\frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \boldsymbol{u}} & \frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \tilde{\boldsymbol{\nu}}} \\ \frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{u}} & \frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}}\end{bmatrix}^\mathrm{T}\begin{bmatrix}\boldsymbol{\psi}_\mathrm{u} \\ \boldsymbol{\psi}_{\tilde{\nu}}\end{bmatrix}\right)^\mathrm{T}\begin{bmatrix}\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}\boldsymbol{x}} \\ \frac{\mathrm{d}\tilde{\boldsymbol{\nu}}}{\mathrm{d}\boldsymbol{x}}\end{bmatrix} \tag{3.10}$$

By setting the sum in the round brackets to be zero, the adjoint problem can be stated as follows

$$\begin{bmatrix}\frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \boldsymbol{u}} & \frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \tilde{\boldsymbol{\nu}}} \\ \frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{u}} & \frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}}\end{bmatrix}^\mathrm{T}\begin{bmatrix}\boldsymbol{\psi}_\mathrm{u} \\ \boldsymbol{\psi}_{\tilde{\nu}}\end{bmatrix} = -\begin{bmatrix}\left[\frac{\partial f}{\partial \boldsymbol{u}}\right]^\mathrm{T} \\ \left[\frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}}\right]^\mathrm{T}\end{bmatrix} \tag{3.11}$$

Solving Eq. (3.11) is not an easy task, since all Jacobi matrices are needed. One method is to use a segregated approach to compute the adjoint variables $\boldsymbol{\psi}_\mathrm{u}$ and $\boldsymbol{\psi}_{\tilde{\nu}}$ as explained below.
Once the adjoint variables are known, the sensitivities can be computed

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{x}} = \frac{\partial f}{\partial \boldsymbol{x}} + \boldsymbol{\psi}_\mathrm{u}^\mathrm{T}\frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \boldsymbol{x}} + \boldsymbol{\psi}_{\tilde{\nu}}^\mathrm{T}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{x}} \tag{3.12}$$

The second and third term of the RHS in Eq. (3.12) can be approximated with the help of a finite difference approach. Using a central finite difference scheme

$$\begin{aligned}
\boldsymbol{\psi}_\mathrm{u}^\mathrm{T}\frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \boldsymbol{x}} + \boldsymbol{\psi}_{\tilde{\nu}}^\mathrm{T}\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \boldsymbol{x}} &\approx \boldsymbol{\psi}_\mathrm{u}^\mathrm{T}\frac{\boldsymbol{R}_\mathrm{u}(\boldsymbol{x}+\boldsymbol{\Delta x}) - \boldsymbol{R}_\mathrm{u}(\boldsymbol{x}-\boldsymbol{\Delta x})}{2\boldsymbol{\Delta x}} \\
&\qquad + \boldsymbol{\psi}_{\tilde{\nu}}^\mathrm{T}\frac{\boldsymbol{R}_{\tilde{\nu}}(\boldsymbol{x}+\boldsymbol{\Delta x}) - \boldsymbol{R}_{\tilde{\nu}}(\boldsymbol{x}-\boldsymbol{\Delta x})}{2\boldsymbol{\Delta x}}
\end{aligned} \tag{3.13}$$

the sensitivities, i.e. Eq. (3.12), can be evaluated. This approach is usually sensitive w.r.t. the step size $\boldsymbol{\Delta x}$, therefore a step size study is necessary.
It has been mentioned that one can solve the coupled adjoint problem, i.e. Eq. (3.11), by a segregated approach. Therefore, it is assumed, that the adjoint variables $\boldsymbol{\psi}_\mathrm{u}$ have been computed before and now one seeks a solution for the adjoint variables $\boldsymbol{\psi}_{\tilde{\nu}}$. Evaluating the second row from Eq. (3.11), assuming that the objective function does not depend explicitly on $\tilde{\boldsymbol{\nu}}$

$$\left[\frac{\partial \boldsymbol{R}_\mathrm{u}}{\partial \tilde{\boldsymbol{\nu}}}\right]^\mathrm{T}\boldsymbol{\psi}_\mathrm{u} + \left[\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}}\right]^\mathrm{T}\boldsymbol{\psi}_{\tilde{\nu}} = -\left[\frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}}\right]^\mathrm{T} = \boldsymbol{0} \tag{3.14}$$

one can solve for the Spalart-Allmaras adjoint variables $\boldsymbol{\psi}_{\tilde{\nu}}$

$$\left[\frac{\partial \boldsymbol{R}_{\tilde{\nu}}}{\partial \tilde{\boldsymbol{\nu}}}\right]^{\mathrm{T}} \boldsymbol{\psi}_{\tilde{\nu}} = -\left[\frac{\partial \boldsymbol{R}_{\mathrm{u}}}{\partial \tilde{\boldsymbol{\nu}}}\right]^{\mathrm{T}} \boldsymbol{\psi}_{\mathrm{u}} = -\left[\boldsymbol{\psi}_{\mathrm{u}}^{\mathrm{T}} \frac{\partial \boldsymbol{R}_{\mathrm{u}}}{\partial \tilde{\boldsymbol{\nu}}}\right]^{\mathrm{T}} \tag{3.15}$$

with the objective function $f = \boldsymbol{\psi}_{\mathrm{u}}^{\mathrm{T}} \boldsymbol{R}_{\mathrm{u}}$.

In Eq. (3.15) the derivatives of the residuals for the RANS and the Spalart-Allmaras variables are needed. The term $[\partial \boldsymbol{R}_{\tilde{\nu}}/\partial \tilde{\boldsymbol{\nu}}]^{\mathrm{T}}$ is presented in chapter 4. The term $\partial \boldsymbol{R}_{\mathrm{u}}/\partial \tilde{\boldsymbol{\nu}}$ is discussed in chapter 11.

3.4 Total Finite Difference Approach

The sensitivity or gradient can also be computed using a total finite difference approach

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} \approx \frac{f(\boldsymbol{x} + \Delta\boldsymbol{x}) - f(\boldsymbol{x} - \Delta\boldsymbol{x})}{2\Delta\boldsymbol{x}} \tag{3.16}$$

where for every change in DV a solution of the complete problem is needed, i.e. in this context a complete CFD solution.

In order to verify the adjoint method, it is common practice to use a total finite difference approach. Since the gradient is sensitive w.r.t. the step size, a study for $\Delta\boldsymbol{x}$ is needed.

4 The Spalart-Allmaras Turbulence Model

The Spalart-Allmaras turbulence model is a one-equation model that solves a modeled transport equation for the kinematic turbulent or eddy viscosity ν_t. The Spalart-Allmaras model was designed specifically for aerospace applications involving wall-bounded flows and has been shown to give good results for boundary layers subjected to adverse pressure gradients. It is not calibrated for general industrial flows, and does produce relatively larger errors for some free shear flows, especially plane and round jet flows. In addition, it cannot be relied on to predict the decay of homogeneous, isotropic turbulence.[4]

The Spalart-Allmaras model with various modifications is well documented on the NASA Langley Turbulence Modeling Resource website.[5]

4.1 The "Standard" Spalart-Allmaras Model

The "Standard" Spalart-Allmaras one-equation turbulence model as given in [SA92] or [AJS12], obeys the transport equation

$$\underbrace{\frac{\partial \tilde{\nu}}{\partial t} + U_j \frac{\partial \tilde{\nu}}{\partial x_j}}_{\text{Material derivative}} = \underbrace{c_{\mathrm{b1}} \left(1 - f_{\mathrm{t2}}\right) \tilde{S} \tilde{\nu}}_{\text{Production terms}} + \underbrace{\frac{1}{\sigma} \left[\frac{\partial}{\partial x_j} \left(\left(\nu + \tilde{\nu}\right) \frac{\partial \tilde{\nu}}{\partial x_j} \right) + c_{\mathrm{b2}} \frac{\partial \tilde{\nu}}{\partial x_j} \frac{\partial \tilde{\nu}}{\partial x_j} \right]}_{\text{Diffusion terms}} \underbrace{- \left(c_{\mathrm{w1}} f_\mathrm{w} - \frac{c_{\mathrm{b1}}}{\kappa^2} f_{\mathrm{t2}} \right) \left(\frac{\tilde{\nu}}{d} \right)^2}_{\text{Destruction terms}} \tag{4.1}$$

where $\tilde{\nu}$ is the Spalart-Allmaras variable related to the turbulent kinematic viscosity ν_t by

$$\nu_\mathrm{t} = \tilde{\nu} f_{\mathrm{v1}} \qquad f_{\mathrm{v1}} = \frac{\chi^3}{\chi^3 + c_{\mathrm{v1}}^3} \qquad \chi = \frac{\tilde{\nu}}{\nu}$$

Note that the turbulent kinematic viscosity is a field quantity and therefore in general a function of space and time, i.e. $\nu_\mathrm{t} = \nu_\mathrm{t}(\boldsymbol{x}, t)$. However, it is often assumed, that the spatial and time dependency is neglected for the adjoint problem, i.e. the turbulent kinematic viscosity is constant, known as "frozen turbulence" assumption. This leads to erroneous results as shown in chapter 1. The aim is to remedy the frozen turbulence assumption by considering the variation of the turbulent viscosity for the adjoint problem.

For simplicity, the flow is considered to be stationary, i.e. $\partial \tilde{\nu} / \partial t = 0$, and incompressible, therefore the material derivative simplifies to the convection term

$$U_j \frac{\partial \tilde{\nu}}{\partial x_j} = \frac{\partial \left(U_j \tilde{\nu}\right)}{\partial x_j} - \tilde{\nu} \frac{\partial U_j}{\partial x_j} = \frac{\partial \left(U_j \tilde{\nu}\right)}{\partial x_j} \tag{4.2}$$

where advantage of mass conservation has been taken in order to drop $\tilde{\nu} \partial U_j / \partial x_j$.

[4] `https://en.wikipedia.org/wiki/Spalart-Allmaras_turbulence_model` (12.07.2016)
[5] `http://turbmodels.larc.nasa.gov/spalart.html` (16.07.2016)

4.2 The "Turbulent" Spalart-Allmaras Model

For fully turbulent flows, Eq. (4.1) can be simplified. The coefficient f_{t2}, which was a numerical fix in the original model for stabilization reasons, is often set to zero for reasonably high Reynolds numbers. For writing convenience, an effective kinematic turbulent viscosity $\tilde{\nu}_{\text{eff}} = (\nu + \tilde{\nu})/\sigma$ is introduced. Eq. (4.1) is then written as follows

$$\underbrace{\frac{\partial (U_j \tilde{\nu})}{\partial x_j}}_{\text{Convection term}} = \underbrace{c_{b1}\tilde{S}\tilde{\nu}}_{\text{Production term}} + \underbrace{\frac{\partial}{\partial x_j}\left(\tilde{\nu}_{\text{eff}}\frac{\partial \tilde{\nu}}{\partial x_j}\right) + \frac{c_{b2}}{\sigma}\frac{\partial \tilde{\nu}}{\partial x_j}\frac{\partial \tilde{\nu}}{\partial x_j}}_{\text{Diffusion terms}} - \underbrace{c_{w1} f_w \left(\frac{\tilde{\nu}}{d}\right)^2}_{\text{Destruction term}} \tag{4.3}$$

Here $\tilde{S}$ is the modified vorticity,

$$\tilde{S} = \max\left(\Omega + \frac{\tilde{\nu}}{\kappa^2 d^2} f_{v2}, c_s \Omega\right) \qquad f_{v2} = 1 - \frac{\chi}{1 + \chi f_{v1}}$$

where $\Omega = \sqrt{\Omega_{ij}\Omega_{ij}}$ is the magnitude of the rotation rate tensor $\Omega_{ij} = \frac{1}{2}(\partial U_i/\partial x_j - \partial U_j/\partial x_i)$ and d the distance to the closest wall. The function f_w is

$$f_w = g\left(\frac{1 + c_{w3}^6}{g^6 + c_{w3}^6}\right)^{1/6} \qquad g = r + c_{w2}\left(r^6 - r\right) \qquad r = \min\left(\frac{\tilde{\nu}}{\tilde{S}\kappa^2 d^2}, 10.0\right)$$

The constants are

$$\kappa = 0.41 \qquad \sigma = \frac{2}{3} \qquad c_{b1} = 0.1355 \qquad c_{b2} = 0.622 \qquad c_s = 0.3 \qquad c_{v1} = 7.1$$
$$c_{w1} = \frac{c_{b1}}{\kappa^2} + \frac{1 + c_{b2}}{\sigma} \qquad c_{w2} = 0.3 \qquad c_{w3} = 2.0$$

Boundary conditions for $\tilde{\nu}$ are set in accordance to "Guidelines for Specification of Turbulence at Inlet Boundaries", published by the ESI CFD Support Team.[6] They may be chosen from the turbulent viscosity ratio $\beta = \nu_t/\nu$. For internal flows, β can be scaled with the Reynolds number. Some guidelines (determined with numerical experiments) for fully developed pipe flows are given in Tbl. 4.1. For a Reynolds number of 100 000 or greater, a value of 100 is a

Re	3000	5000	10 000	15 000	20 000
β	11.6	16.5	26.7	34.0	50.1

Table 4.1: β in dependence of Reynolds number for internal flows

reasonable estimate for β.

For external flows, the freestream turbulent viscosity is of the order of laminar viscosity, so small values of β are appropriate. The respective boundary conditions are as follows

no-slip wall: $\tilde{\nu} = 0$ inlet/freestream: $\beta \approx 0.1 - 1.0$ outlet: $\frac{\partial \tilde{\nu}}{\partial \boldsymbol{n}} = 0$ (zero Gradient)

The implementation of Eq. (4.3) can be found in the source code file `SpalartAllmaras.C`. For convenience, the implementation is given in Lst. 4.1.

Listing 4.1: Implementation of the Spalart-Allmaras Eq. (4.3) in OpenFOAM (SpalartAllmaras.C)

```
...
tmp<fvScalarMatrix> nuTildaEqn
```

[6] `http://www.esi-cfd.com/content/view/877/192/` (15.07.2016)

```
(
    fvm::ddt(alpha, rho, nuTilda_)
  + fvm::div(alphaRhoPhi, nuTilda_)
  - fvm::laplacian(alpha*rho*DnuTildaEff(), nuTilda_)
  - Cb2_/sigmaNut_*alpha*rho*magSqr(fvc::grad(nuTilda_))
  ==
    Cb1_*alpha*rho*Stilda*nuTilda_
  - fvm::Sp(Cw1_*alpha*rho*fw(Stilda)*nuTilda_/sqr(y_), nuTilda_)
);
...
```

Note that the factor alpha is simply one and has no effect, since alpha is used for simulating porous materials. Note also that the time derivative is present, however, for steady state flows it is not considered by the solver.

To implement the adjoint equation in a discrete manner in OpenFOAM, it is necessary to express all spatial differential operators in implicit form in order to setup the Jacobi matrix. Since the gradient operator, i.e. `fvc::grad(nuTilda_)`, is only available in explicit form, one solution is to reformulate the expression into operators that are available in implicit form. In general, the magnitude squared of a tensor is the r-th inner product of the tensor of rank r with itself, to produce a scalar. The gradient of $\tilde{\nu}$ has rank one, therefore the magnitude squared of the gradient of $\tilde{\nu}$, is the inner product of the gradients of $\tilde{\nu}$, i.e. $|\nabla\tilde{\nu}|^2 = \nabla\tilde{\nu}\cdot\nabla\tilde{\nu} = \frac{\partial\tilde{\nu}}{\partial x_j}\frac{\partial\tilde{\nu}}{\partial x_j}$. Remember that $\tilde{\nu}_{\text{eff}} = (\nu+\tilde{\nu})/\sigma$. Making use of the identity

$$\begin{aligned}\frac{\partial}{\partial x_j}\left(\tilde{\nu}_{\text{eff}}\frac{\partial\tilde{\nu}}{\partial x_j}\right) &= \frac{\partial\tilde{\nu}_{\text{eff}}}{\partial x_j}\frac{\partial\tilde{\nu}}{\partial x_j} + \tilde{\nu}_{\text{eff}}\frac{\partial^2\tilde{\nu}}{\partial x_j^2} \\ &= \frac{1}{\sigma}\left(\frac{\partial\nu}{\partial x_j}\frac{\partial\tilde{\nu}}{\partial x_j} + \frac{\partial\tilde{\nu}}{\partial x_j}\frac{\partial\tilde{\nu}}{\partial x_j}\right) + \tilde{\nu}_{\text{eff}}\frac{\partial^2\tilde{\nu}}{\partial x_j^2}\end{aligned} \tag{4.4}$$

it follows that

$$\frac{1}{\sigma}\frac{\partial\tilde{\nu}}{\partial x_j}\frac{\partial\tilde{\nu}}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\tilde{\nu}_{\text{eff}}\frac{\partial\tilde{\nu}}{\partial x_j}\right) - \tilde{\nu}_{\text{eff}}\frac{\partial^2\tilde{\nu}}{\partial x_j^2} \tag{4.5}$$

with $\partial\nu/\partial x_j = 0$, i.e. the kinematic molecular viscosity ν is constant. Substituting (4.5) in (4.3), the reformulated form of the "Turbulent" Spalart-Allmaras model is written as

$$\underbrace{\frac{\partial\left(U_j\tilde{\nu}\right)}{\partial x_j}}_{T_1} = \underbrace{c_{\text{b1}}\tilde{S}\tilde{\nu}}_{T_2} + \underbrace{\left(1+c_{\text{b2}}\right)\frac{\partial}{\partial x_j}\left(\tilde{\nu}_{\text{eff}}\frac{\partial\tilde{\nu}}{\partial x_j}\right)}_{T_3} - \underbrace{c_{\text{b2}}\tilde{\nu}_{\text{eff}}\frac{\partial^2\tilde{\nu}}{\partial x_j^2}}_{T_4} - \underbrace{c_{w1}f_{\text{w}}\left(\frac{\tilde{\nu}}{d}\right)^2}_{T_5} \tag{4.6}$$

where T_1 is the convectional derivative term, T_2 the production term, T_3 and T_4 the diffusion terms and T_5 the destruction term. Note that the diffusion terms are now written in terms of the Laplace operator rather then in terms of the gradient operator.

To implement Eq. (4.6) in OpenFOAM, a new turbulence model named `mySpalartAllmaras` has been created. How to add a new turbulence model in OpenFOAM v3.0.1 is documented in chapter 9. For convenience, the implementation of Eq. (4.6) is given in Lst. 4.2.

Listing 4.2: Implementation of the reformulated Spalart-Allmaras Eq. (4.6) in OpenFOAM (mySpalartAllmaras.C)

```
...
tmp<fvScalarMatrix> nuTildaEqn
(
    fvm::ddt(alpha, rho, nuTilda_)
  + fvm::div(alphaRhoPhi, nuTilda_)
  - (1.0 + Cb2_)*fvm::laplacian(alpha*rho*DnuTildaEff(), nuTilda_)
```

```
  + Cb2_*alpha*rho*DnuTildaEff()*fvm::laplacian(nuTilda_)
  ==
    Cb1_*alpha*rho*Stilda*nuTilda_
  - fvm::Sp(Cw1_*alpha*rho*fw(Stilda)*nuTilda_/sqr(y_), nuTilda_)
);
...
```

Remark on the Laplace Operator

The implicit Laplace operator, i.e. `fvm::laplacian(...)`, is split into two parts, an orthogonal and non-orthogonal part. In OpenFOAM only the orthogonal part is implemented implicitly, whereas the non-orthogonal part is implemented explicitly. The implementation of the Laplace operator is very well explained in [MMD15]. That means for orthogonal meshes the Jacobi matrix is exact, however, for non-orthogonal meshes the Jacobi matrix is not exact. For all examples in this theses the effect seemed to be not significant, however, it would be worth to account for the effect. See also chapter 11.

4.3 Discrete Differentiation of the "Turbulent" Spalart-Allmaras Model

To setup the discrete adjoint equation for the Spalart-Allmaras turbulence model, i.e. Eq (3.5), Eq. (4.6) needs to be differentiated w.r.t. $\tilde{\nu}$. The idea is to differentiate the equation using concepts of variational calculus as well as total derivatives, and then using OpenFOAM operators to setup the Jacobi matrix $[\partial \boldsymbol{R}_{\tilde{\nu}}/\partial \tilde{\boldsymbol{\nu}}]^{\mathrm{T}}$. This will be discussed on a term by term basis in the following.

It is assumed, that the functional

$$\begin{aligned} \tilde{\nu}\colon \mathbb{R}^3 &\to \mathbb{R} \\ \boldsymbol{x} &\mapsto \tilde{\nu}(\boldsymbol{x}) \end{aligned}$$

and its Gâteaux derivative exist and is unique. In mathematics, the Gâteaux derivative is a generalization of the concept of directional derivative in differential calculus.[7] If $F = F[\tilde{\nu}]$, then the Gâteaux derivative can be computed as follows

$$\frac{\partial F[\tilde{\nu}]}{\partial \tilde{\nu}} = \frac{\mathrm{d}}{\mathrm{d}\epsilon} F[\tilde{\nu} + \epsilon \delta\tilde{\nu}]\bigg|_{\epsilon=0} \tag{4.7}$$

where $\delta\tilde{\nu}$ is the direction in which the derivative is computed. Note that for the notation of the Gâteaux derivative the partial derivative symbol is used. In literature different notations can be found, however, it is chosen for consistency.

4.3.1 Convection Term (T_1)

The convection term reads

$$T_1[\tilde{\nu}] = \frac{\partial (U_j \tilde{\nu})}{\partial x_j} \tag{4.8}$$

[7] `https://en.wikipedia.org/wiki/Gâteaux_derivative` (08.07.2016)

i.e. the divergence of the product of velocity U_j and the Spalart-Allmaras variable $\tilde{\nu}$. This term is linear in $\tilde{\nu}$. Due to the divergence operator, the Gâteaux derivative is applied

$$\begin{aligned}\frac{\partial(T_1[\tilde{\nu}])}{\partial\tilde{\nu}} &= \frac{\mathrm{d}}{\mathrm{d}\epsilon}T_1[\tilde{\nu}+\epsilon\delta\tilde{\nu}]\bigg|_{\epsilon=0} \\ &= \frac{\mathrm{d}}{\mathrm{d}\epsilon}\frac{\partial}{\partial x_j}\left(U_j\left(\tilde{\nu}+\epsilon\delta\tilde{\nu}\right)\right)\bigg|_{\epsilon=0} \\ &= \frac{\mathrm{d}}{\mathrm{d}\epsilon}\left(\frac{\partial\left(U_j\tilde{\nu}\right)}{\partial x_j}+\epsilon\frac{\partial\left(U_j\delta\tilde{\nu}\right)}{\partial x_j}\right)\bigg|_{\epsilon=0} \\ &= \frac{\partial\left(U_j\delta\tilde{\nu}\right)}{\partial x_j}\end{aligned} \tag{4.9}$$

Since the divergence operator is available in implicit form by OpenFOAM, this term can easily be implemented as given in Lst. 6.4 denoted by `T1`. Note that phi is the face flux field, i.e. the scalar product of the linear interpolated velocity vector and the surface normal vector at the faces.

4.3.2 Production Term (T_2)

The production term is written in full dependency of $\tilde{\nu}$ as follows

$$T_2[\tilde{\nu}] = c_{\mathrm{b1}}\tilde{S}(\tilde{\nu}, f_{\mathrm{v2}}(\chi(\tilde{\nu}), f_{\mathrm{v1}}(\chi(\tilde{\nu}))))\tilde{\nu} \tag{4.10}$$

Since the production term includes no spatial differential operator, i.e. the term does not depend on adjacent cells, the total derivative is applied. Using the product and chain rule, the total derivative is given by

$$\frac{\partial T_2}{\partial\tilde{\nu}} = c_{\mathrm{b1}}\left[\left(\frac{\partial\tilde{S}}{\partial\tilde{\nu}}+\frac{\partial\tilde{S}}{\partial f_{\mathrm{v2}}}\frac{\partial f_{\mathrm{v2}}}{\partial\chi}\frac{\partial\chi}{\partial\tilde{\nu}}+\frac{\partial\tilde{S}}{\partial f_{\mathrm{v2}}}\frac{\partial f_{\mathrm{v2}}}{\partial f_{\mathrm{v1}}}\frac{\partial f_{\mathrm{v1}}}{\partial\chi}\frac{\partial\chi}{\partial\tilde{\nu}}\right)\tilde{\nu}+\tilde{S}\right] \tag{4.11}$$

The differentiation is carried out with the help of Wolfram Mathematica. It can be written in terms of $\tilde{\nu}$ in a closed form as follows

$$\frac{\partial T_2}{\partial\tilde{\nu}} = \frac{c_{\mathrm{b1}}}{d^2\kappa^2\nu}\left(d^2\kappa^2\nu\Omega+2\nu\tilde{\nu}-3\tilde{\nu}^2+\frac{7\tilde{\nu}^6}{\tilde{\nu}^3\left(\tilde{\nu}+\nu\right)+\nu^4c_{\mathrm{v1}}^3}-\frac{\left(4\tilde{\nu}+3\nu\right)\tilde{\nu}^9}{\left(\tilde{\nu}^3\left(\tilde{\nu}+\nu\right)+\nu^4c_{\mathrm{v1}}^3\right)^2}\right) \tag{4.12}$$

The implementation in OpenFOAM is given in Lst. 6.4 denoted by `T2`. Alternatively one can evaluate the partial derivatives and apply Eq. (4.11). The partial derivatives are derived in section 4.3.5.

4.3.3 Diffusion Term (T_3)

The diffusion term is split into two parts, T_3 and T_4, from Eq. (4.6)

$$T_3[\tilde{\nu}] = \left(1+c_{\mathrm{b2}}\right)\frac{\partial}{\partial x_j}\left(\tilde{\nu}_{\mathrm{eff}}\frac{\partial\tilde{\nu}}{\partial x_j}\right) \tag{4.13}$$

i.e. the product of the coefficient $(1+c_{\mathrm{b2}})$ and the Laplace operator with coefficient $\tilde{\nu}_{\mathrm{eff}}$. Note that this term is nonlinear due to $\tilde{\nu}_{\mathrm{eff}}$. Due to the Laplace operator, the Gâteaux derivative is

applied

$$\begin{aligned}
\frac{\partial(T_3[\tilde{\nu}])}{\partial\tilde{\nu}} &= \frac{\mathrm{d}}{\mathrm{d}\epsilon}T_3[\tilde{\nu}+\epsilon\delta\tilde{\nu}]\bigg|_{\epsilon=0} \\
&= \frac{\mathrm{d}}{\mathrm{d}\epsilon}\left(1+c_{\mathrm{b}2}\right)\frac{1}{\sigma}\frac{\partial}{\partial x_j}\left(\left(\nu+\tilde{\nu}+\epsilon\delta\tilde{\nu}\right)\frac{\partial\left(\tilde{\nu}+\epsilon\delta\tilde{\nu}\right)}{\partial x_j}\right)\bigg|_{\epsilon=0} \\
&= \frac{\mathrm{d}}{\mathrm{d}\epsilon}\left(1+c_{\mathrm{b}2}\right)\frac{1}{\sigma}\frac{\partial}{\partial x_j}\left(\nu\frac{\partial\tilde{\nu}}{\partial x_j}+\epsilon\nu\frac{\partial\delta\tilde{\nu}}{\partial x_j}+\tilde{\nu}\frac{\partial\tilde{\nu}}{\partial x_j}+\epsilon\tilde{\nu}\frac{\partial\delta\tilde{\nu}}{\partial x_j}+\epsilon\delta\tilde{\nu}\frac{\partial\tilde{\nu}}{\partial x_j}+\epsilon^2\delta\tilde{\nu}\frac{\partial\delta\tilde{\nu}}{\partial x_j}\right)\bigg|_{\epsilon=0} \\
&= \left(1+c_{\mathrm{b}2}\right)\frac{1}{\sigma}\frac{\partial}{\partial x_j}\left(\nu\frac{\partial\delta\tilde{\nu}}{\partial x_j}+\tilde{\nu}\frac{\partial\delta\tilde{\nu}}{\partial x_j}+\frac{\partial\tilde{\nu}}{\partial x_j}\delta\tilde{\nu}+2\epsilon\delta\tilde{\nu}\frac{\partial\delta\tilde{\nu}}{\partial x_j}\right)\bigg|_{\epsilon=0} \\
&= \left(1+c_{\mathrm{b}2}\right)\frac{1}{\sigma}\frac{\partial}{\partial x_j}\left(\nu\frac{\partial\delta\tilde{\nu}}{\partial x_j}+\tilde{\nu}\frac{\partial\delta\tilde{\nu}}{\partial x_j}+\frac{\partial\tilde{\nu}}{\partial x_j}\delta\tilde{\nu}\right) \\
&= \underbrace{\left(1+c_{\mathrm{b}2}\right)\frac{\partial}{\partial x_j}\left(\tilde{\nu}_{\mathrm{eff}}\frac{\partial\delta\tilde{\nu}}{\partial x_j}\right)}_{T_{3\mathrm{a}}}+\underbrace{\left(1+c_{\mathrm{b}2}\right)\frac{\partial}{\partial x_j}\left(\frac{1}{\sigma}\frac{\partial\tilde{\nu}}{\partial x_j}\delta\tilde{\nu}\right)}_{T_{3\mathrm{b}}}
\end{aligned} \tag{4.14}$$

The implementation in OpenFOAM is given in Lst. 6.4 denoted by `T3a` and `T3b`. Note that due to the nonlinearity the derivative consist of two terms.

4.3.4 Diffusion Term (T_4)

From Eq. (4.6)

$$T_4[\tilde{\nu}] = c_{\mathrm{b}2}\tilde{\nu}_{\mathrm{eff}}\frac{\partial^2\tilde{\nu}}{\partial x_j^2} \tag{4.15}$$

i.e. the product of the coefficient $c_{\mathrm{b}2}\tilde{\nu}_{\mathrm{eff}}$ and the Laplace operator. Note that this term is nonlinear due to $\tilde{\nu}_{\mathrm{eff}}$. Applying the Gâteaux derivative

$$\begin{aligned}
\frac{\partial(T_4[\tilde{\nu}])}{\partial\tilde{\nu}} &= \frac{\mathrm{d}}{\mathrm{d}\epsilon}T_4[\tilde{\nu}+\epsilon\delta\tilde{\nu}]\bigg|_{\epsilon=0} \\
&= \frac{\mathrm{d}}{\mathrm{d}\epsilon}c_{\mathrm{b}2}\frac{1}{\sigma}\left(\nu+\tilde{\nu}+\epsilon\delta\tilde{\nu}\right)\frac{\partial^2\left(\tilde{\nu}+\epsilon\delta\tilde{\nu}\right)}{\partial x_j^2}\bigg|_{\epsilon=0} \\
&= \frac{\mathrm{d}}{\mathrm{d}\epsilon}c_{\mathrm{b}2}\frac{1}{\sigma}\left(\nu\frac{\partial^2\tilde{\nu}}{\partial x_j^2}+\epsilon\nu\frac{\partial^2\delta\tilde{\nu}}{\partial x_j^2}+\tilde{\nu}\frac{\partial^2\tilde{\nu}}{\partial x_j^2}+\epsilon\tilde{\nu}\frac{\partial^2\delta\tilde{\nu}}{\partial x_j^2}+\epsilon\delta\tilde{\nu}\frac{\partial^2\tilde{\nu}}{\partial x_j^2}+\epsilon^2\delta\tilde{\nu}\frac{\partial^2\delta\tilde{\nu}}{\partial x_j^2}\right)\bigg|_{\epsilon=0} \\
&= c_{\mathrm{b}2}\frac{1}{\sigma}\left(\nu\frac{\partial^2\delta\tilde{\nu}}{\partial x_j^2}+\tilde{\nu}\frac{\partial^2\delta\tilde{\nu}}{\partial x_j^2}+\frac{\partial^2\tilde{\nu}}{\partial x_j^2}\delta\tilde{\nu}+2\epsilon\delta\tilde{\nu}\frac{\partial^2\delta\tilde{\nu}}{\partial x_j^2}\right)\bigg|_{\epsilon=0} \\
&= c_{\mathrm{b}2}\frac{1}{\sigma}\left(\nu\frac{\partial^2\delta\tilde{\nu}}{\partial x_j^2}+\tilde{\nu}\frac{\partial^2\delta\tilde{\nu}}{\partial x_j^2}+\frac{\partial^2\tilde{\nu}}{\partial x_j^2}\delta\tilde{\nu}\right) \\
&= \underbrace{c_{\mathrm{b}2}\tilde{\nu}_{\mathrm{eff}}\frac{\partial^2\delta\tilde{\nu}}{\partial x_j^2}}_{T_{4\mathrm{a}}}+\underbrace{c_{\mathrm{b}2}\frac{1}{\sigma}\frac{\partial^2\tilde{\nu}}{\partial x_j^2}\delta\tilde{\nu}}_{T_{4\mathrm{b}}}
\end{aligned} \tag{4.16}$$

The implementation in OpenFOAM is given in Lst. 6.4 denoted by `T4a` and `T4b`.

4.3.5 Destruction Term (T_5)

The destruction term is similar to the production term, i.e. no spatial differential operator is applied. In full dependence of $\tilde{\nu}$ it reads

$$T_5[\tilde{\nu}] = c_{\mathrm{w1}} f_{\mathrm{w}}(g(r(\tilde{\nu}, \tilde{S}(\tilde{\nu}, f_{\mathrm{v2}}(\chi(\tilde{\nu}), f_{\mathrm{v1}}(\chi(\tilde{\nu})))))))\left(\frac{\tilde{\nu}}{d}\right)^2 \tag{4.17}$$

applying the chain and product rules, the total derivative is given by

$$\begin{aligned}\frac{\partial T_5}{\partial \tilde{\nu}} = c_{\mathrm{w1}} \Bigg[\Bigg(&\frac{\partial f_{\mathrm{w}}}{\partial g}\frac{\partial g}{\partial r}\frac{\partial r}{\partial \tilde{\nu}} + \frac{\partial f_{\mathrm{w}}}{\partial g}\frac{\partial g}{\partial r}\frac{\partial r}{\partial \tilde{S}}\frac{\partial \tilde{S}}{\partial \tilde{\nu}} + \frac{\partial f_{\mathrm{w}}}{\partial g}\frac{\partial g}{\partial r}\frac{\partial r}{\partial \tilde{S}}\frac{\partial \tilde{S}}{\partial f_{\mathrm{v2}}}\frac{\partial f_{\mathrm{v2}}}{\partial \chi}\frac{\partial \chi}{\partial \tilde{\nu}} \\ &+ \frac{\partial f_{\mathrm{w}}}{\partial g}\frac{\partial g}{\partial r}\frac{\partial r}{\partial \tilde{S}}\frac{\partial \tilde{S}}{\partial f_{\mathrm{v2}}}\frac{\partial f_{\mathrm{v1}}}{\partial \chi}\frac{\partial \chi}{\partial \tilde{\nu}}\Bigg)\left(\frac{\tilde{\nu}}{d}\right)^2 + 2\frac{f_{\mathrm{w}}}{d^2}\tilde{\nu}\Bigg]\end{aligned} \tag{4.18}$$

Computing the total derivative in terms of $\tilde{\nu}$ with the help of a computer algebra system is possible. However, the result ends up in a very lengthy expression inconvenient to write down on paper and even worse to implement in OpenFOAM. Therefore, each partial derivative contributing to Eq. (4.18) is computed separately, i.e.

$$\frac{\partial \chi}{\partial \tilde{\nu}} = \frac{1}{\nu} \tag{4.19}$$

$$\frac{\partial f_{\mathrm{v1}}}{\partial \chi} = \frac{3c_{\mathrm{v1}}^3\chi^2}{\left(c_{\mathrm{v1}}^3 + \chi^3\right)^2} \tag{4.20}$$

$$\frac{\partial f_{\mathrm{v2}}}{\partial f_{\mathrm{v1}}} = \frac{\chi^2}{\left(1 + \chi f_{\mathrm{v1}}\right)^2} \tag{4.21}$$

$$\frac{\partial \tilde{S}}{\partial f_{\mathrm{v2}}} = \frac{\tilde{\nu}}{\kappa^2 d^2} \tag{4.22}$$

$$\frac{\partial r}{\partial \tilde{S}} = -\frac{\tilde{\nu}}{\kappa^2 d^2 \tilde{S}^2} \tag{4.23}$$

$$\frac{\partial g}{\partial r} = 1 + c_{\mathrm{w2}}\left(-1 + 6r^5\right) \tag{4.24}$$

$$\frac{\partial f_{\mathrm{w}}}{\partial g} = \frac{c_{\mathrm{w3}}^6\left(\frac{1+c_{\mathrm{w3}}^6}{g^6+c_{\mathrm{w3}}^6}\right)^{7/6}}{1 + c_{\mathrm{w3}}^6} \tag{4.25}$$

$$\frac{\partial f_{\mathrm{v2}}}{\partial \chi} = -\frac{1}{\left(1 + \chi f_{\mathrm{v1}}\right)^2} \tag{4.26}$$

$$\frac{\partial \tilde{S}}{\partial \tilde{\nu}} = \frac{f_{\mathrm{v2}}}{\kappa^2 d^2} \tag{4.27}$$

$$\frac{\partial r}{\partial \tilde{\nu}} = \frac{1}{\tilde{S}\kappa^2 d^2} \tag{4.28}$$

Each partial derivative is implemented as a function in OpenFOAM, the implementation is given in Lst. 6.6.

5 CFD Verification

As it has been derived in chapter 4, Eq. (4.3) and (4.6) are mathematically equivalent. However, numerically the results in OpenFOAM are of interest. In the following, two cases are computed in order to show the difference in numerical results for OpenFOAM. It might be already mentioned in advance that the difference is negligible and therefore one can either use Eq. (4.3) or (4.6) to solve the problem numerically with OpenFOAM.

5.1 NACA 2412 Airfoil

As the first example, the NACA 2412 airfoil is used. The case is tailer made for the Spalart-Allmaras turbulence model which is most suitable for external flows. The mesh can be found online.[8] More details on the NACA 2412 airfoil profile can also be found online.[9] The geometry and mesh is shown in Fig. 5.1. The mesh consists of 6060 hexahedra cells and 12 322 nodes. The mesh non-orthogonality is maximum 16.03 and in average 1.75.

The airfoil has a cord length of 1.0 m and the radius of the domain is 110 m. The BCs are given in Tbl. 5.1. Note that the boundary layer is not properly resolved ($y^+ \gg 1$) and therefore a wall

Variable	Location	BC type	Value
P in $\mathrm{m^2\,s^{-2}}$	farfield	freestreamPressure	-
	wall	zeroGradient	-
	sides	empty	-
U in $\mathrm{m\,s^{-1}}$	farfield	freestream	uniform (19.97259 1.04671 0)
	wall	fixedValue	uniform (0 0 0)
	sides	empty	-
ν_t in $\mathrm{m^2\,s^{-1}}$	farfield	freestream	uniform 0.1E−5
	wall	nutUSpaldingWallFunction	uniform 0
	sides	empty	-
$\tilde{\nu}$ in $\mathrm{m^2\,s^{-1}}$	farfield	freestream	uniform 0.1E−5
	wall	fixedValue	uniform 0
	sides	empty	-

Table 5.1: Boundary conditions NACA 2412

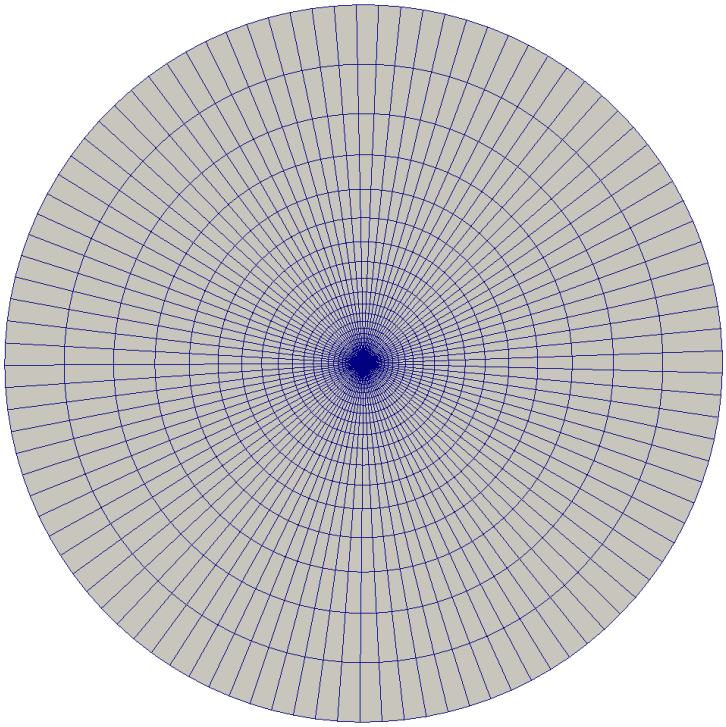
function (nutUSpaldingWallFunction) is used. The velocity magnitude is set to $U = 20.0\,\mathrm{m\,s^{-1}}$. The cord line of the airfoil is aligned with the horizontal x-axis. The inlet velocity components are computed in dependency of the AoA as follows

$$U_x = U\cos(\alpha) \tag{5.1}$$

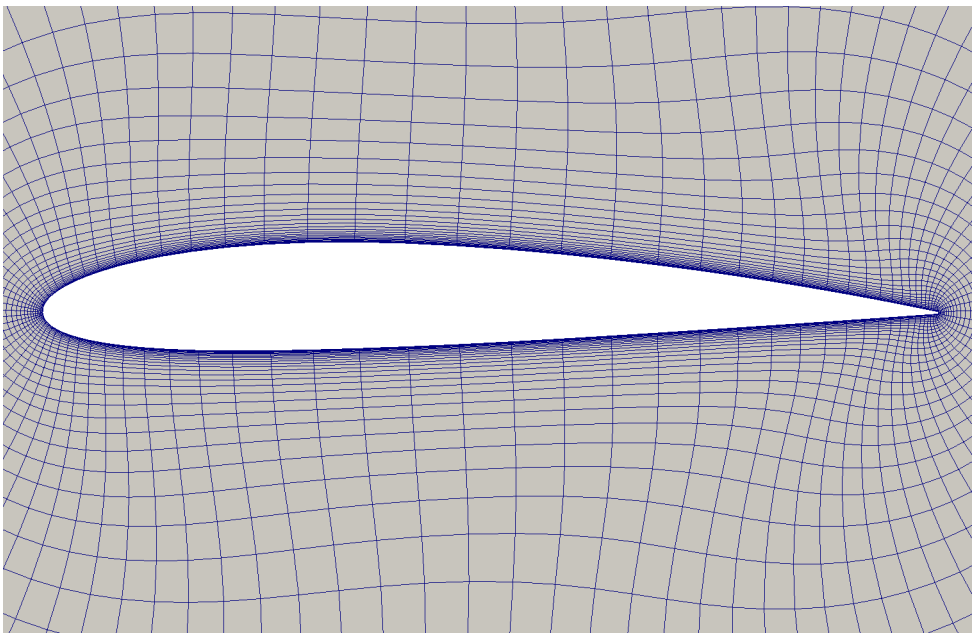
$$U_y = U\sin(\alpha) \tag{5.2}$$

The AoA is set to $\alpha = 3.0°$, density $\rho = 1.0\,\mathrm{kg\,m^{-3}}$, kinematic viscosity $\nu = 1\mathrm{E}{-5}\,\mathrm{m^2\,s^{-1}}$ and Reynolds number $\mathrm{Re} = 2.0\mathrm{E}6$.

[8] `https://github.com/traviscarrigan/OpenFOAM-NACA2412` (21.08.2016)
[9] `http://airfoiltools.com/airfoil/details?airfoil=naca2412-il#polars` (21.08.2016)



(a) Mesh

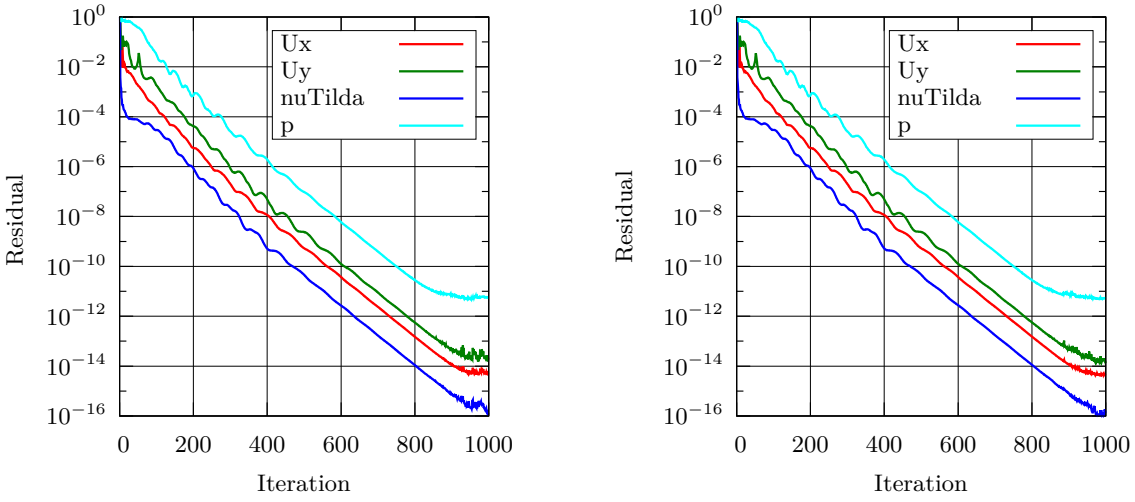


(b) Airfoil

Figure 5.1: Domain and mesh NACA 2412

The `simpleFoam` solver is invoked and the residuals are plotted in Fig. 5.2. The subscript `ref` refers to the results of the Spalart-Allmaras Eq. (4.3), whereas `mod` refers to the results of the reformulated Spalart-Allamras Eq. (4.6). The residuals are dropping to the same oder of magnitude as well as being almost identical in the trend.

The lift and drag forces as well as the absolut relative error, computed according to Eq. (5.3),

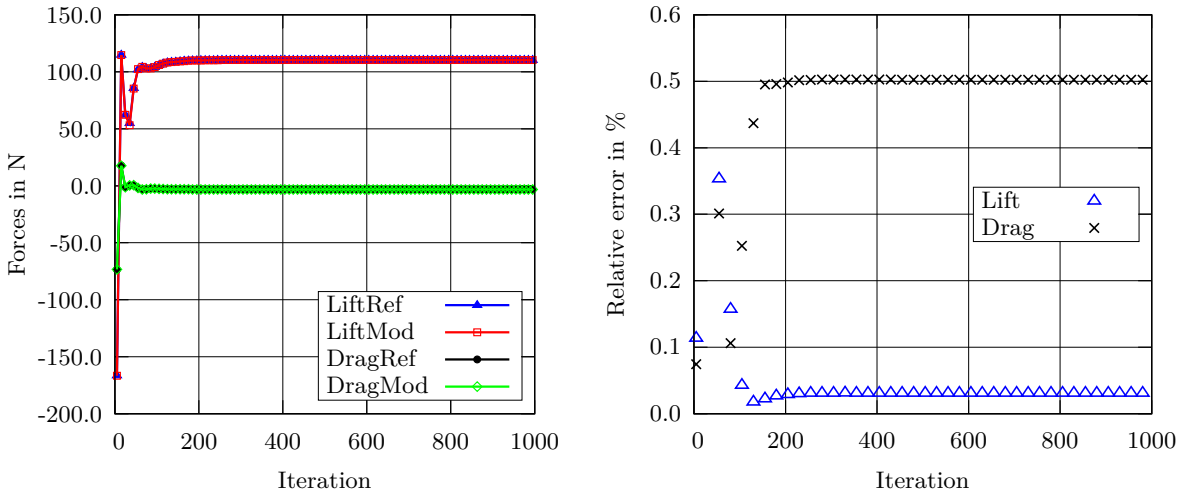


(a) `ref` (b) `mod`

Figure 5.2: Residual plots NACA 2412

are shown in Fig. 5.3.

$$\text{Relative error} = \frac{|\text{Force}_{\text{ref}} - \text{Force}_{\text{mod}}|}{|\text{Force}_{\text{ref}}|} 100 \quad [\%] \tag{5.3}$$



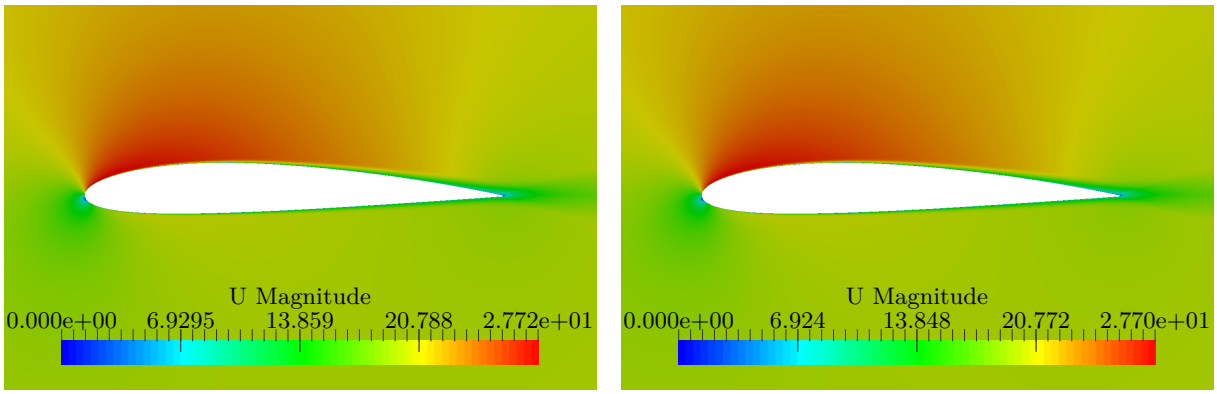
(a) Forces (b) Absolute relative error

Figure 5.3: Lift and drag forces and absolute relative error NACA 2412

The absolute relative error is less than 0.5 % for the drag and less than 0.05 % for the lift. Concluding that results are matching very well.

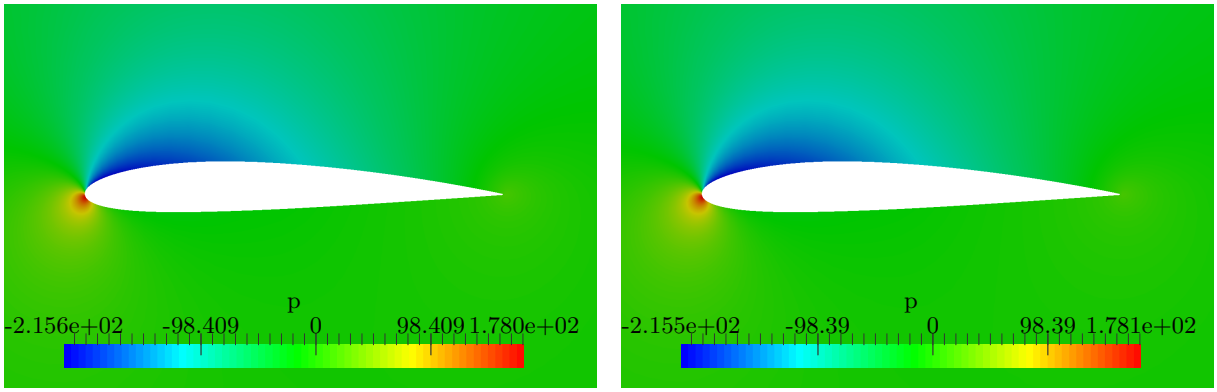
Contour plots for velocity magnitude, pressure and nuTilda are shown in Fig. 5.4, 5.5 and 5.6 respectively. Flow direction is from left to right.

It is observed, that visually the solutions are the same and for the values there are negligible differences, concluding that either Eq. (4.3) or (4.6) can be used to compute the Spalart-Allmaras variable.



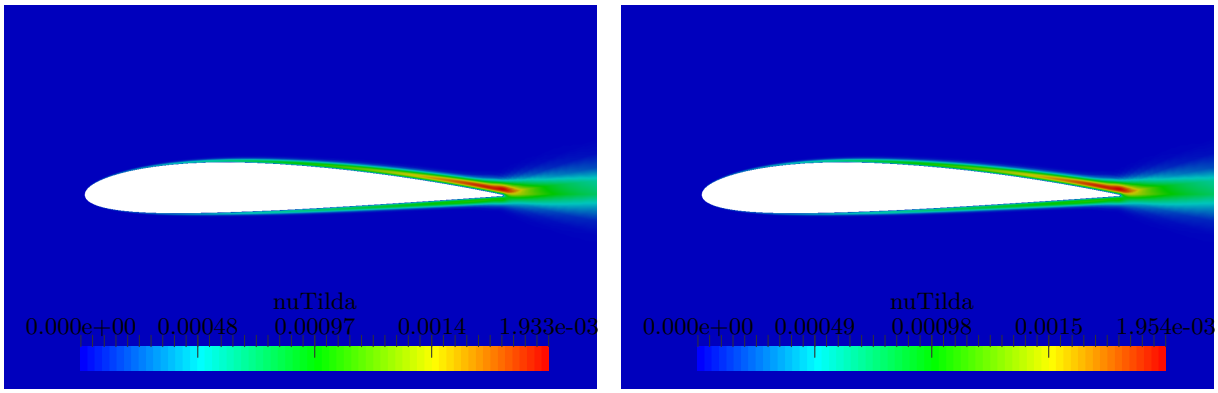
(a) `ref` (b) `mod`

Figure 5.4: Velocity magnitude field NACA 2412



(a) `ref` (b) `mod`

Figure 5.5: Pressure field NACA 2412

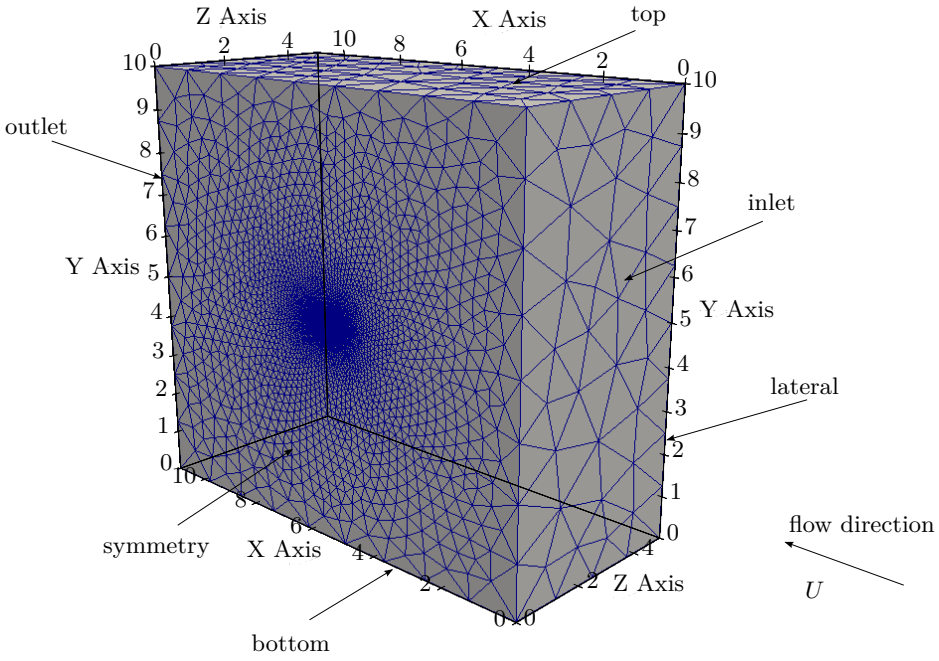


(a) `ref` (b) `mod`

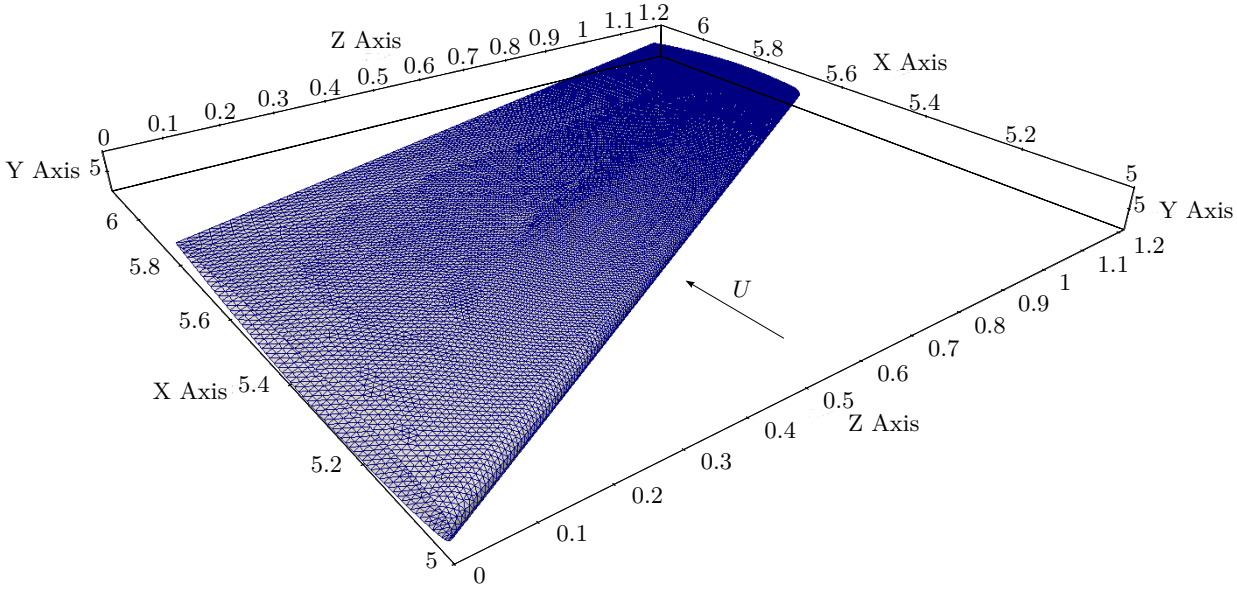
Figure 5.6: NuTilda field NACA 2412

5.2 ONERA M6 Wing

As the second example, a fully turbulent flow over the ONERA M6 wing is considered. The geometry and mesh is shown in Fig. 5.7. The overall domain bounding box has length 11 m × 10 m × 5 m, and the wing bounding box 1.2 m × 0.06 m × 1.2 m. The mesh consists out of 341 797 tetrahedra cells and 72 791 nodes. The mesh non-orthogonality is maximum 63.93 and in average 20.52.



(a) Domain bounding box



(b) Wing bounding box

Figure 5.7: Geometry and mesh ONERA M6

The velocity magnitude is $U = 15.0\,\mathrm{m\,s^{-1}}$, AoA $\alpha = 3.0°$, density $\rho = 1.412\,\mathrm{kg\,m^{-3}}$, kinematic viscosity $\nu = 1.132\text{E}{-5}\,\mathrm{m^2\,s^{-1}}$ and Reynolds number Re $\approx 1.33\text{E}6$. The BCs are given in Tbl. 5.2. Note that the boundary layer is not properly resolved ($y^+ \gg 1$) and therefore a wall function

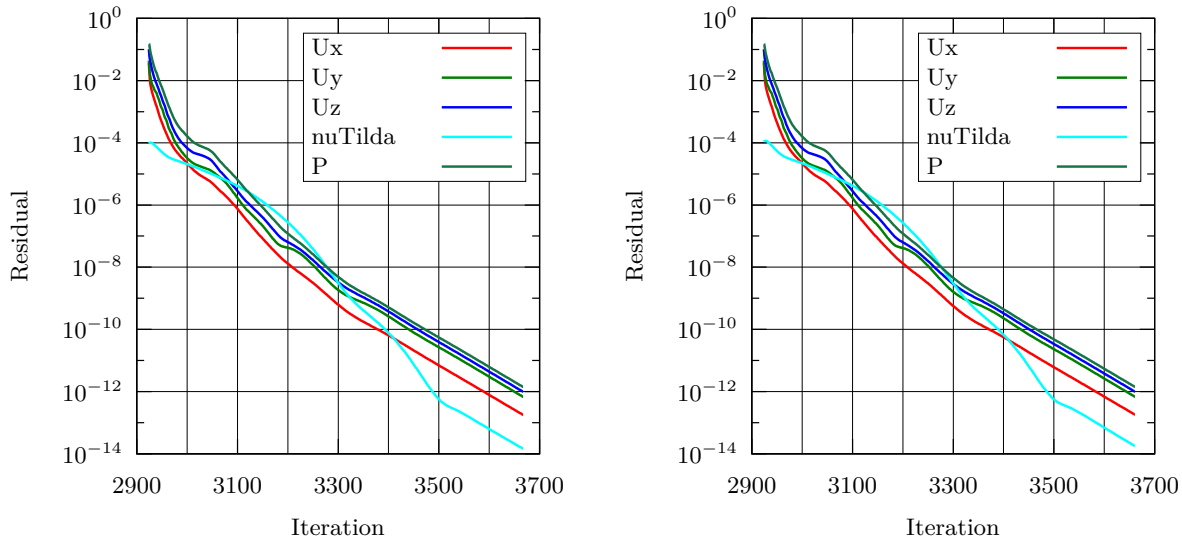
Variable	Location	BC type	Value
P in $\mathrm{m^2\,s^{-2}}$	inlet	zeroGradient	-
	outlet	fixedValue	uniform 0
	wing	zeroGradient	-
	top	slip	-
	bottom	slip	-
	lateral	slip	-
	symmetry	symmetryPlane	-
U in $\mathrm{m\,s^{-1}}$	inlet	fixedValue	uniform (14.97944302 0.78504 0)
	outlet	zeroGradient	-
	wing	fixedValue	uniform (0 0 0)
	top	slip	-
	bottom	slip	-
	lateral	slip	-
	symmetry	symmetryPlane	-
ν_t in $\mathrm{m^2\,s^{-1}}$	inlet	freestream	uniform 5.5E−5
	outlet	freestream	uniform 5.5E−5
	wing	nutUSpaldingWallFunction	uniform 0
	top	slip	-
	bottom	slip	-
	lateral	slip	-
	symmetry	symmetryPlane	-
$\tilde{\nu}$ in $\mathrm{m^2\,s^{-1}}$	inlet	freestream	uniform 5.5E−5
	outlet	freestream	uniform 5.5E−5
	wing	fixedValue	uniform 0
	top	slip	-
	bottom	slip	-
	lateral	slip	-
	symmetry	symmetryPlane	-

Table 5.2: Boundary conditions ONERA M6

(nutUSpaldingWallFunction) is used at the wing.

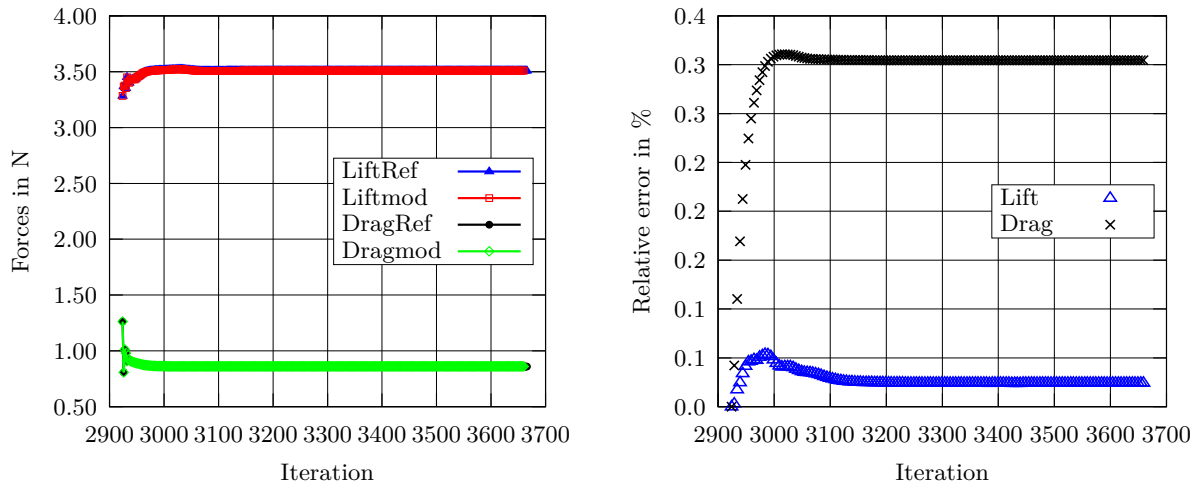
The `simpleFoam` solver is invoked and the residuals are plotted in Fig. 5.8. The subscript `ref` refers to the results of the Spalart-Allmaras Eq. (4.3), whereas `mod` refers to the results of the reformulated Spalart-Allamras Eq. (4.6). The residuals are visually identical. However, the reformulated solution needed six iterations less to converge, hence less computational cost. This is common for implicit computed solutions.

The lift and drag forces for the wing as well as the absolute relative error, computed according to Eq. (5.3), are shown in Fig. 5.9. The absolute relative error is less than 0.3 % for the drag and less than 0.05 % for the lift. Further comparison is given for the turbulent viscosity as well as the Spalart-Allmaras variable in Fig. 5.10, 5.11 and 5.12 respectively. The flow direction is from left to right. It is observed, that the solution is visual identical, concluding that either Eq. (4.3) or (4.6) can be used to compute the Spalart-Allmaras variable.



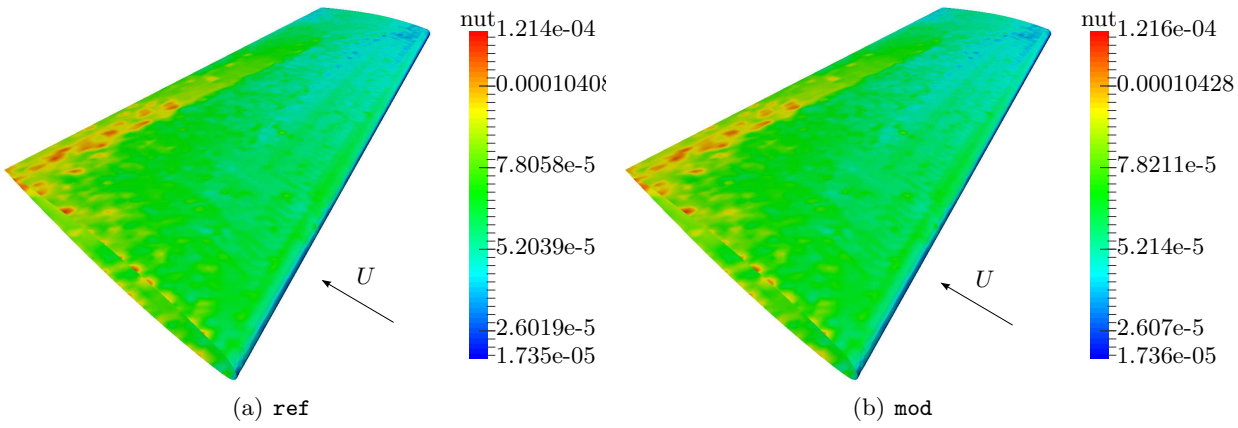
(a) `ref` (b) `mod`

Figure 5.8: Residual plots ONERA M6



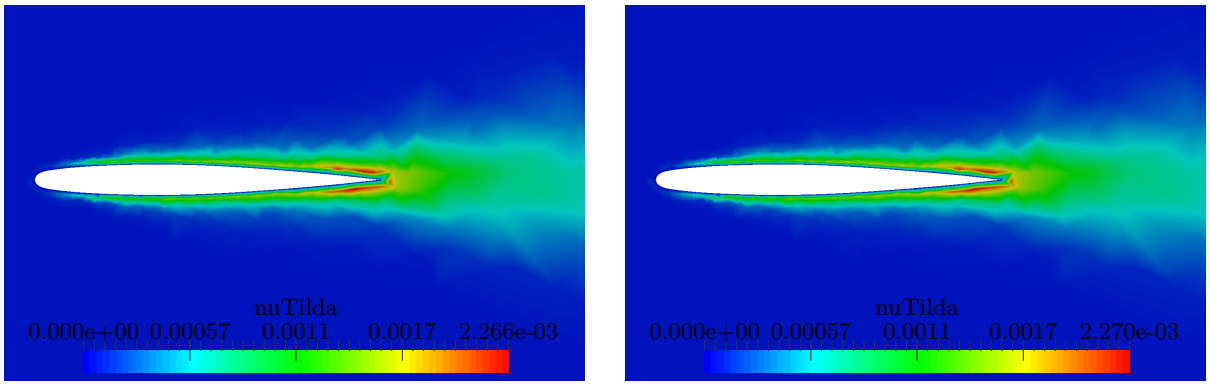
(a) Forces (b) Relative error

Figure 5.9: Lift and drag forces, relative error ONERA M6



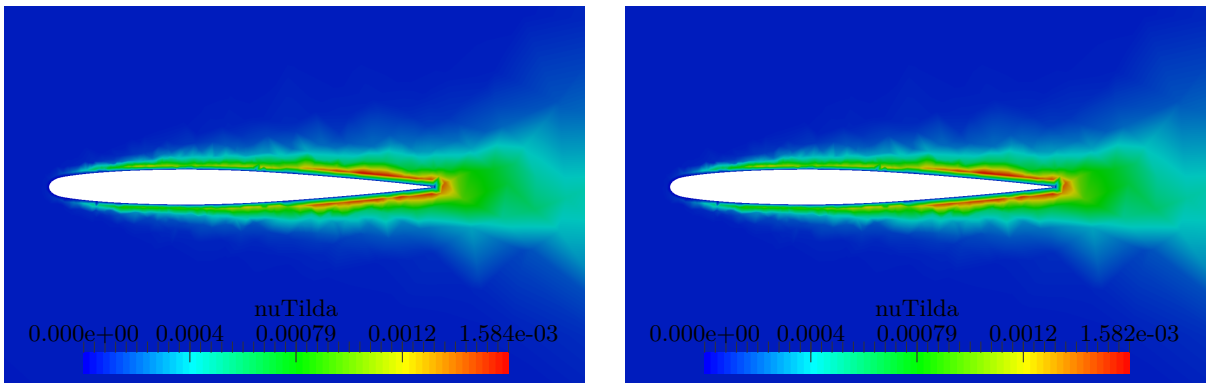
(a) `ref` (b) `mod`

Figure 5.10: Turbulent viscosity ONERA M6



(a) `ref` (b) `mod`

Figure 5.11: Spalart-Allmaras variable in xy-plane at $z = 0.2\,\mathrm{m}$ ONERA M6



(a) `ref` (b) `mod`

Figure 5.12: Spalart-Allmaras variable in xy-plane at $z = 0.8\,\mathrm{m}$ ONERA M6

6 Implementation of the Adjoint Spalart-Allmaras Solver

In the following, the implementation of the Spalart-Allmaras adjoint solver in OpenFOAM v3.0.1 is presented. All files can be downloaded online.[10] The directory tree structure is as follows

```
OpenFOAM/
└── <user>-3.0.1/
    └── applications/
        └── solvers/
            └── adjointSpalartAllmaras/
                ├── Make/
                │   ├── files
                │   └── options
                ├── adjointMethod.H
                ├── adjointProblem.H
                ├── adjointSpalartAllmaras.C
                ├── createConstants.H
                ├── createFields.H
                ├── FDMethod.H
                ├── navierStokes.H
                ├── primalProblem.H
                ├── readInParameters.H
                ├── sensitivityAnalysis.H
                └── spalartAllmaras.H
```

To compile the solver files, execute the `wmake` command within the `adjointSpalartAllmaras` folder. The solver can be called by executing `adjointSpalartAllmaras` within the case folder.

The implementation is demonstrated with the help of an example. For simplicity, the geometry is chosen to be a unit cube with little amount of cells to keep the computational cost low. Although this case does not present a physical reasonable example, it is sufficient for the purpose of guidance through the implementation. Physically reasonable verification examples are presented in chapter 7. The domain is equally discretized by $n \times n \times n = n^3$ cells, where $n = 4$ is the number of cells in each coordinate direction. Total numbers of cells is 64. The discretized unit cube domain and the optimization points on the optimization patch are shown in Fig. 6.1. The BCs are given in Tbl. 6.1. The velocity magnitude at the inlet is set to $U = U_x = 1.0\,\mathrm{m\,s^{-1}}$.

[10] `https://github.com/DennisKasper/OpenFOAM301/tree/master/applications/solvers/adjointSpalartAllmaras`

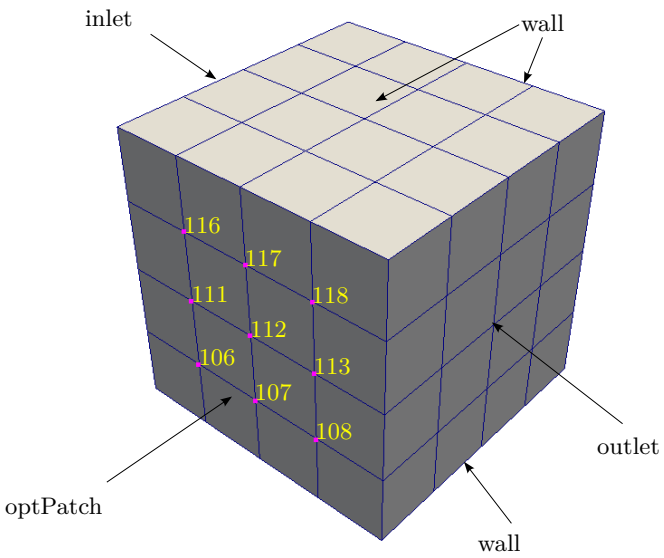


Figure 6.1: Discretized domain with 64 cells and optimization points Unit Cube

Variable	Location	BC type	Value
P in $\mathrm{m^2\,s^{-2}}$	inlet	zeroGradient	-
	outlet	fixedValue	uniform 0
	wall	zeroGradient	-
	optPatch	zeroGradient	-
U in $\mathrm{m\,s^{-1}}$	inlet	fixedValue	uniform (1 0 0)
	outlet	zeroGradient	-
	wall	fixedValue	uniform (0 0 0)
	optPatch	fixedValue	uniform (0 0 0)
ν_t in $\mathrm{m^2\,s^{-1}}$	inlet	fixedValue	uniform 1e-05
	outlet	zeroGradient	-
	wall	fixedValue	uniform 0
	optPatch	fixedValue	uniform 0
$\tilde{\nu}$ in $\mathrm{m^2\,s^{-1}}$	inlet	fixedValue	uniform 1e-05
	outlet	zeroGradient	-
	wall	fixedValue	uniform 0
	optPatch	fixedValue	uniform 0

Table 6.1: Boundary conditions Unit Cube

The files for the `unitCubeSpalartAllmaras` case can also be downloaded online.[11] The directory tree structure is as follows

```
OpenFOAM/
└── <user>-3.0.1/
    └── run/
        └── unitCubeSpalartAllmaras/
            ├── 0/
            │   ├── nut
            │   ├── nuTilda
            │   ├── p
            │   └── U
            ├── constant/
            │   ├── optimizationProperties
            │   ├── transportProperties
            │   └── turbulenceProperties
            └── system/
                ├── blockMeshDict
                ├── controlDict
                ├── fvSchemes
                └── fvSolution
```

To create the mesh, execute `blockMesh` in the `unitCubeSpalartAllmaras` folder. The CFD solution is computed by executing the turbulent steady state solver `simpleFoam`. The solution is converge to an order of magnitude of 10^{-8} within approximately 54 iterations depending on the machine. In order to run the `adjointSpalartAllmaras` solver, the file `nuTilda` located in the last time step folder needs to be copied and renamed to `adjnuTilda`, which in this example is the time folder 54.

The `main` function is located in the `adjointSpalartAllmaras.C` file given in Lst. 6.1. Comments as well as screen output are suppressed an indicated by the "..." symbol. The code is structured in separated header files.

Listing 6.1: Implementation of the adjoint Spalart-Allmaras solver in OpenFOAM v3.0.1 (`adjointSpalartAllmaras.C`)

```
...
// Include OpenFOAM directive
#include "bound.H"
#include "fvCFD.H"
#include "pointMesh.H"
#include "pointFields.H"
#include "singlePhaseTransportModel.H"
#include "turbulentTransportModel.H"
#include "valuePointPatchFields.H"
#include "wallDist.H"

// Include  C++ directive
#include <fstream>

// Include User directive
#include "spalartAllmaras.H"
#include "navierStokes.H"
#include "sensitivityAnalysis.H"


int main(int argc, char *argv[])
```

[11] https://github.com/DennisKasper/OpenFOAM301/tree/master/run/unitCubeSpalartAllmaras

```
{
#include "setRootCase.H"      // Set the correct path
#include "createTime.H"       // Create the time
#include "createMesh.H"       // Create the mesh
#include "createFields.H"     // Create the fields
#include "createConstants.H"  // Create the Spalart-Allmaras constants
#include "readInParameters.H" // Read in the paramters

// Advance in pseudo time
runTime++;
Info << "runTime = " << runTime.timeName() << nl << endl;

#include "primalProblem.H"  // optional
...
#include "adjointProblem.H"
...
#include "adjointMethod.H"
...
#include "FDMethod.H"

...

return 0;
}
```

Header files are included for OpenFOAM, C++ and the user. The users `spalartAllmaras.H` header contains the functions and derivative terms needed for the Spalart-Allmaras turbulence model as presented in Chap. 4. The users `navierStokes.H` header is given in preparation for the coupling term $\partial \boldsymbol{R}_{\mathrm{u}}/\partial \tilde{\boldsymbol{\nu}}$ as discussed in Chap. 11. The users `sensitivityAnalysis.H` header contains the functions to compute the semi-analytic finite difference, i.e. Eq. (3.7) and the total finite difference, i.e. Eq. (3.16).

In order to read in the optimization parameters, a file named `optimizationProperties` needs to be saved within the `constant` folder. Lst. 6.2 presents the contend for the `unitCubeSpalartAllmaras` case.

Listing 6.2: Content of `optimizationProperties` file located in the `constant` folder

```
...
// Optimization patch names
optimizationPatchNames  (optPatch);  // (patch1 patch2 ...)

// Parameter for step size study
stepStart               1e-10;
stepStop                1e-1;
stepSize                1e+1;

// Analysis for (all selective none) nodes
adjoint                 selective;
finiteDifference        selective;

// Select the global IDs
globalIDsAdj            (106 107 108 111 112 113 116 117 118);
globalIDsFD             (106 107 108 111 112 113 116 117 118);

// Sultion criteria for primal problem and total finite difference
maxItr                  2000;
eps                     1e-12;
relaxFac                0.7;

// Case
caseExample             unitCubeSpalartAllmaras;
```

```
// Objective Function
objectiveFunction       artificial;
```

The analysis is split into four subproblems, the primal Problem (`primalProblem.H`) which is optional to double check the implemented Spalart-Allmaras equation, the adjoint problem (`adjointProblem.H`) and the sensitivity analysis split into adjoint and total finite difference method (`adjointMethod.H` and `FDMethod.H`). Each subproblem is discussed in the following sections.

6.1 Primal Problem

Executing the `primalProblem.H` is optional to double check the implemented Spalart-Allmaras equation. Implemented is the reformulated Spalart-Allmaras Eq. (4.6). The implementation is given in Lst. 6.3. Due to the nonlinearity of the problem, it is solved in an iterative manner. The solution parameters `maxItr`, `eps` and `relaxFac` can be set in the `optimizationProperties` file.

Listing 6.3: Implementation of the primal problem in OpenFOAM v3.0.1 (`primalProblem.H`)

```
// Initialize parameters
label iter = 0;
scalar maxResidual = 1.0;

// Solve primal problem
while (iter < maxItr && maxResidual > eps)
{
    iter += 1.0;
    Info<<"Iteration " << iter << endl;

    nuTilda.storePrevIter();

    const volScalarField chi = SpalartAllmaras::chi(nuTilda, nu);
    const volScalarField fv1 = SpalartAllmaras::fv1(chi, cv1);
    const volScalarField Stilda = SpalartAllmaras::Stilda(chi, fv1, nuTilda, U,
        d, kappa, cs);

    fvScalarMatrix nuTildaEqn
    (
        // T1
        fvm::div(phi, nuTilda)

        // T2
        - fvm::SuSp(cb1*Stilda, nuTilda)

        // T3
        - (1.0 + cb2)*fvm::laplacian(SpalartAllmaras::nuTildaEff(nuTilda, sigma,
            nu), nuTilda)

        // T4
        + cb2*SpalartAllmaras::nuTildaEff(nuTilda, sigma, nu)*fvm::laplacian(
            nuTilda)

        // T5
        + fvm::SuSp(cw1*SpalartAllmaras::fw(Stilda, nuTilda, d, kappa, cw2, cw3)
            *nuTilda/sqr(d), nuTilda)
    );

    nuTildaEqn.relax(relaxFac);
    maxResidual = solve(nuTildaEqn, mesh.solver("nuTilda")).initialResidual();
    bound(nuTilda, dimensionedScalar("0", nuTilda.dimensions(), 0.0));
```

```
    nuTilda.correctBoundaryConditions();

    nut = nuTilda*fv1;
    nut.correctBoundaryConditions();
}

runTime.write();
```

The solution of the primal problem is given in Fig. 6.2.

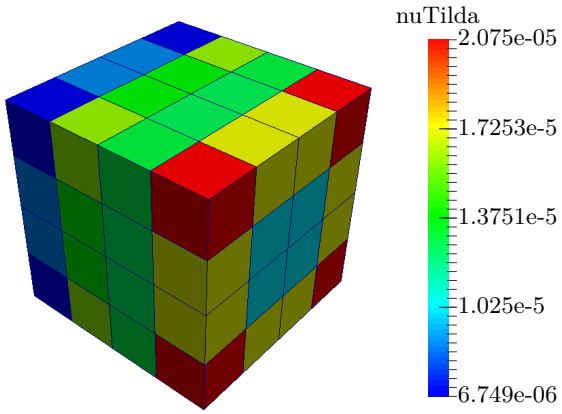


Figure 6.2: Solution for the primal problem Unit Cube

6.2 Adjoint Problem

The implementation of the adjoint problem, i.e. Eq. (3.5), is given in Lst. 6.4.

Listing 6.4: Implementation of the adjoint problem in OpenFOAM v3.0.1 (`adjointProblem.H`)

```
// Computing the derivative of T2
const volScalarField dT2dNuTilda = SpalartAllmaras::calcDT2dNuTilda(d, nuTilda,
    U, nu, kappa, cb1, cv1);

// Computing the derivative of T5
const volScalarField dT5dNuTilda = SpalartAllmaras::calcDT5dNuTilda(d, nuTilda,
    U, nu, cv1, kappa, cw1, cw2, cw3, cs);

// Setup the Jacobi matrix
fvScalarMatrix adjnuTildaEqn
(
  // T1
  fvm::div(phi, adjnuTilda)

  // T2
  - fvm::Sp(dT2dNuTilda, adjnuTilda)

  // T3a
  - (1.0 + cb2)*fvm::laplacian(SpalartAllmaras::nuTildaEff(nuTilda, sigma, nu),
      adjnuTilda)

  // T3b
  - (1.0 + cb2)*fvm::div(fvc::snGrad(nuTilda)*mesh.magSf()/sigma, adjnuTilda)

  // T4a
  + cb2*SpalartAllmaras::nuTildaEff(nuTilda, sigma, nu)*fvm::laplacian(
      adjnuTilda)
```

```
  // T4b
  + cb2*fvm::Sp(fvc::laplacian(nuTilda)/sigma, adjnuTilda)

  // T5
  + fvm::Sp(dT5dNuTilda, adjnuTilda)
);

// Get the off-diagonal terms
scalarField& upper = adjnuTildaEqn.upper();
scalarField& lower = adjnuTildaEqn.lower();

// Transpose the matrix
forAll(lower, i)
{
  scalar uI = upper[i];
  scalar lI = lower[i];
  lower[i] = uI;
  upper[i] = lI;
}

// Get the source term
scalarField& sourceCoeff = adjnuTildaEqn.source();

// Set the source term to zero
forAll(sourceCoeff, sourceCoeffI)
  sourceCoeff[sourceCoeffI] = scalar(0.0);

if(objectiveFunction.match("artificial"))
{
  ...
  else if(caseExample.match("unitCubeSpalartAllmaras"))
  {
    // Set the source term value of the last cell to -1
    sourceCoeff[sourceCoeff.size() - 1] = scalar(-1.0);
  }
}
...
else
{
  FatalErrorIn("adjointProblem.H or scalar SensitivityAnalysis::calcnuTildaFD
      (...)")
  << "Wrong objectiveFunction in optimzationProperties"
  << abort(FatalError);
}

// Set the boundary coefficients to zero
forAll(adjnuTilda.boundaryField(), patchI)
{
  forAll(adjnuTildaEqn.boundaryCoeffs()[patchI], faceI)
  {
    adjnuTildaEqn.boundaryCoeffs()[patchI][faceI] = scalar(0.0);
  }
}

// Assign the source term to the field adjSource
forAll(sourceCoeff, sourceCoeffI)
{
  adjSource[sourceCoeffI] = sourceCoeff[sourceCoeffI];
}

// Solve for the adjoint variables
adjnuTildaEqn.solve();
```

```
// Write to disk
runTime.write();
```

First of all, the derivatives of $T2$ and $T5$, i.e. Eq. (4.12) and (4.18), are computed by calling the functions `calcDT2dNuTilda` and `calcDT5dNuTilda`. These functions are located in the `spalartAllmaras.H` file, the implementation is given in Lst. 6.5.

Listing 6.5: Implementation of the derivatives

```
// Defining the total derivative of term T2
tmp<volScalarField> calcDT2dNuTilda
(
  ...
)
{
  const volScalarField Omega = Foam::sqrt(2.0)*mag(skew(fvc::grad(U)));

  return cb1/(nu*sqr(kappa*d))
      *(
      nu*Omega*sqr(kappa*d)
      + 2.0*nu*nuTilda
      - 3.0*sqr(nuTilda)
      + 7.0*pow6(nuTilda)/(pow4(nu)*pow3(cv1) + pow3(nuTilda)*(nu + nuTilda))
      - (4.0*nuTilda + 3.0*nu)*pow(nuTilda, 9.0)/pow(pow3(nuTilda)*(nuTilda + nu
          ) + pow4(nu)*pow3(cv1), 2.0)
      );
}


// Defining the total derivative of term T5
tmp<volScalarField> calcDT5dNuTilda
(
  ...
)
{
  const volScalarField chi = SpalartAllmaras::chi(nuTilda, nu);
  const volScalarField fv1 = SpalartAllmaras::fv1(chi, cv1);
  const volScalarField fv2 = SpalartAllmaras::fv2(chi, fv1);
  const volScalarField Stilda = SpalartAllmaras::Stilda(chi, fv1, nuTilda, U, d,
       kappa, cs);
  const volScalarField fw = SpalartAllmaras::fw(Stilda, nuTilda, d, kappa, cw2,
       cw3);

  volScalarField r
  (
    min
    (
      nuTilda
      /(
        max
        (
          Stilda,
          dimensionedScalar("SMALL", Stilda.dimensions(), SMALL)
        )
        *sqr(kappa*d)
      ),
      scalar(10.0)
    )
  );
  r.boundaryField() == 0.0;

  const volScalarField g = r + cw2*(pow6(r) - r);

  return cw1*((pFwpG(g, cw3)*pGpR(r, cw2)*pRpNuTilda(Stilda, d, kappa)
```

```
      + pFwpG(g, cw3)*pGpR(r, cw2)*pRpStilda(nuTilda, d, Stilda, kappa)*
          pStildapNuTilda(fv2, d, kappa)
      + pFwpG(g, cw3)*pGpR(r, cw2)*pRpStilda(nuTilda, d, Stilda, kappa)*
          pStildapFv2(nuTilda, d, kappa)*pFv2pChi(chi, fv1)*pChipNuTilda(nu)
      + pFwpG(g, cw3)*pGpR(r, cw2)*pRpStilda(nuTilda, d, Stilda, kappa)*
          pStildapFv2(nuTilda, d, kappa)*pFv2pFv1(chi, fv1)*pFv1pChi(chi, cv1)*
          pChipNuTilda(nu)
      )*sqr(nuTilda/d) + 2.0*fw/sqr(d)*nuTilda);
}
```

Note that for the derivative of $T5$ in the return statement, the partial derivatives, i.e. Eq. (4.19) to (4.28), are called via functions. These functions are also located in the file `spalartAllmaras.H`. The implementation is given in Lst. 6.6.

Listing 6.6: Implementation of the partial derivatives

```
// Defining partial derivatives
dimensionedScalar pChipNuTilda
(
  const dimensionedScalar nu
)
{
  return 1.0/nu;
}

tmp<volScalarField> pFv1pChi
(
  const volScalarField& chi,
  const dimensionedScalar cv1
)
{
  const volScalarField chi2 = Foam::pow(chi, 2);
  const volScalarField chi3 = Foam::pow3(chi);

  return 3.0*pow3(cv1)*chi2/pow(pow3(cv1) + chi3, 2);
}

tmp<volScalarField> pFv2pFv1
(
  const volScalarField& chi,
  const volScalarField& fv1
)
{
  const volScalarField chi2 = pow(chi, 2);

  return chi2/pow(1.0 + chi*fv1, 2);
}

tmp<volScalarField> pStildapFv2
(
  const volScalarField& nuTilda,
  const volScalarField& d,
  const dimensionedScalar kappa
)
{
  return nuTilda/sqr(kappa*d);
}

tmp<volScalarField> pRpStilda
(
  const volScalarField& nuTilda,
  const volScalarField& d,
  const volScalarField& Stilda,
  const dimensionedScalar kappa
```

```
)
{
  return -nuTilda/sqr(kappa*d*Stilda);
}

tmp<volScalarField> pGpR
(
  const volScalarField& r,
  const dimensionedScalar cw2
)
{
  const volScalarField r5 = pow5(r);

  return 1.0 + cw2*(-1.0 + 6.0*r5);
}

tmp<volScalarField> pFwpG
(
  const volScalarField& g,
  const dimensionedScalar cw3
)
{
  const volScalarField g6 = pow6(g);

  return pow6(cw3)*pow((1.0 + pow6(cw3))/(g6 + pow6(cw3)), 7/6)/(1.0 + pow6(cw3)
      );
}

tmp<volScalarField> pFv2pChi
(
  const volScalarField& chi,
  const volScalarField& fv1
)
{
  return -1.0/sqr(1.0 + chi*fv1);
}

tmp<volScalarField> pStildapNuTilda
(
  const volScalarField& fv2,
  const volScalarField& d,
  const dimensionedScalar kappa
)
{
  return fv2/sqr(kappa*d);
}

tmp<volScalarField> pRpNuTilda
(
  const volScalarField& Stilda,
  const volScalarField& d,
  const dimensionedScalar kappa
)
{
  return 1.0
    /(
    max // avoid division by zero with SMALL
      (
      Stilda,
      dimensionedScalar("SMALL", Stilda.dimensions(), SMALL)
      )
    *sqr(kappa*d)
    );
```

```
}
```

After the Jacobi matrix is set up, it is transposed by swapping the coefficients of the off diagonal matrix elements.

The objective function is chosen such that the last cell has the value negative one, i.e.

$$f = \tilde{\nu}_{64} \tag{6.1}$$

Therefore the RHS, or the source term of the adjoint Eq. (3.5) is

$$-\left[\frac{\partial f}{\partial \tilde{\boldsymbol{\nu}}}\right]^{\mathrm{T}} = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 & 0 & -1 \end{bmatrix}^{\mathrm{T}} \tag{6.2}$$

it is visualized in Fig. 6.3.

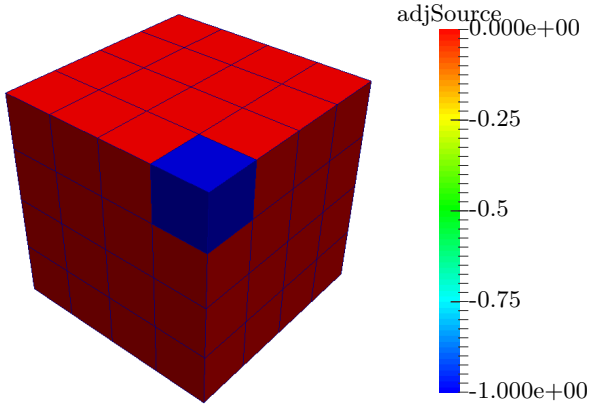


Figure 6.3: RHS or source term Unit Cube

Finally the adjoint equation is solved. The solution of the adjoint problem is given in Fig. 6.4.

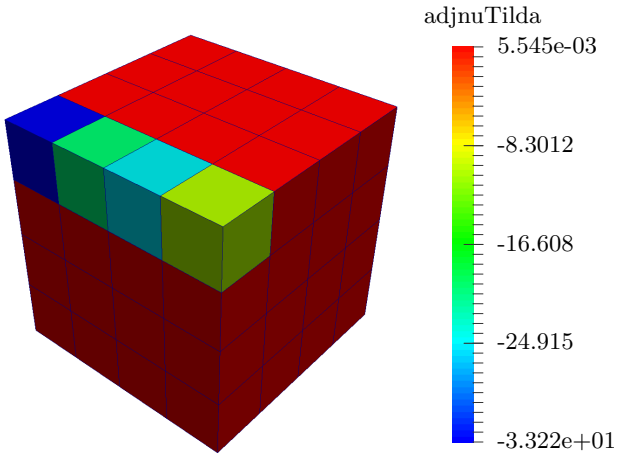


Figure 6.4: Solution of the adjoint problem Unit Cube

6.3 Adjoint Sensitivity Analysis

The implementation of the adjoint sensitivity analysis is given in Lst. 6.7.

Listing 6.7: Implementation of the adjoint sensitivity method in OpenFOAM v3.0.1 (`adjointMethod.H`)

```
if (adjAnalysis.match("all") || adjAnalysis.match("selective"))
{
...
  forAll(optimizationPatchIDs, optimizationPatchIDsI)
  {
    // Get global point IDs of the optimization patch
    labelList myPoints;
    if(adjAnalysis.match("all"))
    {
      myPoints = mesh.boundaryMesh()[optimizationPatchIDs[optimizationPatchIDsI
          ]].meshPoints();
    }
    else if(adjAnalysis.match("selective"))
    {
      myPoints = globalIDsAdj;
    }

    // Get the point normal vectors of the optPatch
    const vectorField normalPointVector = mesh.boundaryMesh()[
        optimizationPatchIDs[optimizationPatchIDsI]].pointNormals();

    // Save the initial mesh point positions
    const pointField initMeshPoints = mesh.points();

    // Point counter
    label pointCounter = myPoints.size();

    forAll(myPoints, myPointI)
    {
      ...
      // Get global point ID
      label globalID = myPoints[myPointI];
      ...
      for(scalar FDStepLoop = FDStepStart; FDStepLoop <= FDStepStop; FDStepLoop
          *= FDStepSize)
      {
        scalar FDStep = FDStepLoop;
        ...
        // Forward perturbation

        // Store the initial mesh points coordinates
        pointField pertMeshPoints = initMeshPoints;

        // Save initial position of point
        vector oldPos = initMeshPoints[globalID];

        // Define vector of perturbation direction
        vector pertVec = normalPointVector[myPointI];

        // Assign new position to point myPointI
        vector newPos = oldPos + FDStep*pertVec;

        // Update position in the mesh
        pertMeshPoints[globalID] = newPos;

        // Move mesh
```

```
        mesh.movePoints(pertMeshPoints);

        // Calculate adjoint dot residual
        scalar lambdaDotResForw = SensitivityAnalysis::calcAdjointDotResidual(
             mesh, runTime, nuTilda, nut, adjnuTilda, U, p, dStar, rho, cb1, cb2
             , kappa, sigma, cs, nu, cw1, cw2, cw3, cv1, objectiveFunction,
             vectorForce, optimizationPatchIDsI);


        // Backward perturbation

        // Store the initial mesh points coordinates
        pertMeshPoints = initMeshPoints;

        // Save initial position of point
        oldPos = initMeshPoints[globalID];

        // Assign new position to point myPointI
        newPos = oldPos - FDStep*pertVec;

        // Update position in the mesh
        pertMeshPoints[globalID] = newPos;

        // Move mesh
        mesh.movePoints(pertMeshPoints);

        // Calculate adjoint dot residual
        scalar lambdaDotResBack = SensitivityAnalysis::calcAdjointDotResidual(
             mesh, runTime, nuTilda, nut, adjnuTilda, U, p, dStar, rho, cb1, cb2
             , kappa, sigma, cs, nu, cw1, cw2, cw3, cv1, objectiveFunction,
             vectorForce, optimizationPatchIDsI);


        // Calculate sensitivity
        scalar adjSens = (lambdaDotResForw - lambdaDotResBack)/(2.0*FDStep);

        // Assign values to adjSensField field
        adjSensField[globalID] = adjSens;
        ...
        // set mesh to initial state
        mesh.movePoints(initMeshPoints);

      } // Close the loop over the step size
    } // Close the loop over the points on the opt patch
  }  // Close the loop over the opt patches
  ...
} // Close adjoint sensitivity analysis
```

Three nested loops are performed, one loop over the optimization patches, one loop over the optimization points and one loop over the step sizes. The optimization patches, the optimization points as well as the step size parameters can be selected within the `optimizationProperties` file. The mesh nodes are moved normal to the surface. The function `calcAdjointDotResidual` computes the scalar product for the semi-analytic finite difference, e.g. for a forward perturbation $\boldsymbol{\psi}_{\tilde{\nu}}^{\mathrm{T}} \boldsymbol{R}_{\tilde{\nu}}(\boldsymbol{x} + \Delta\boldsymbol{x})$. The implementation of the function is given in Lst. 6.8.

Listing 6.8: Implementation of the function `calcAdjointDotResidual`

```
scalar calcAdjointDotResidual
(
...
)
{
  const volScalarField d = SpalartAllmaras::d(mesh);
```

```
  const volScalarField chi = SpalartAllmaras::chi(nuTilda, nu);
  const volScalarField fv1 = SpalartAllmaras::fv1(chi, cv1);
  const volScalarField fv2 = SpalartAllmaras::fv2(chi, fv1);
  const volScalarField Stilda = SpalartAllmaras::Stilda(chi, fv1, nuTilda, U, d,
       kappa, cs);

  const surfaceScalarField phi = linearInterpolate(U) & mesh.Sf();

  // Compute Spalart-Allmaras residual
  volScalarField  RSA =
    // T1
    fvc::div(phi, nuTilda)

    // T2
    - cb1*Stilda*nuTilda

    // T3
    - (1.0 + cb2)*fvc::laplacian(SpalartAllmaras::nuTildaEff(nuTilda, sigma, nu)
        , nuTilda)

    // T4
    +  cb2*SpalartAllmaras::nuTildaEff(nuTilda, sigma, nu)*fvc::laplacian(
        nuTilda)

    // T5
    +  cw1*SpalartAllmaras::fw(Stilda, nuTilda, d, kappa, cw2, cw3)*sqr(nuTilda/
        d);

  forAll(RSA, cI)
  {
    RSA[cI] *= mesh.V()[cI];
  }

  scalar toReturn = 0.0;

  if(objectiveFunction.match("artificial"))
  {
    forAll(RSA, cI)
    {
      toReturn += adjnuTilda[cI]*RSA[cI];
    }
  }
  ...

  return toReturn;
}
```

The sensitivity is computed according to Eq. (3.6). Note that the objective function is not explicitly a function of DVs $\boldsymbol{x}$, hence $\partial f/\partial \boldsymbol{x} = \boldsymbol{0}^{\mathrm{T}}$. The sensitivity equation reduces to

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{x}} = \boldsymbol{\phi}_{\mathrm{adj}}^{\mathrm{T}} \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} \approx \boldsymbol{\phi}_{\mathrm{adj}}^{\mathrm{T}} \frac{\boldsymbol{R}(\boldsymbol{x} + \boldsymbol{\Delta x}) - \boldsymbol{R}(\boldsymbol{x} - \boldsymbol{\Delta x})}{2\Delta \boldsymbol{x}} \tag{6.3}$$

The computation of the residuals is a post processing operation and therefore low in computational cost compared to the primal problem where a non-linear equation system needs to be solved. The solution, i.e. the gradient field $\mathrm{d}L/\mathrm{d}\boldsymbol{x}$ is given in Fig. 6.5(a).

6.4 Total Finite Difference Sensitivity Analysis

The implementation of the total finite difference sensitivity analysis is given in Lst. 6.9.

Listing 6.9: Implementation of the total finite difference method in OpenFOAM v3.0.1 (`FDMethod.H`)

```
if(fdAnalysis.match("all") || fdAnalysis.match("selective"))
{
  ...
  forAll(optimizationPatchIDs, optimizationPatchIDsI)  // Loop over all
      optimization patches
  {
    // Get global point IDs of the optimization patch
    labelList myPoints;
    if(fdAnalysis.match("all"))
    {
      myPoints = mesh.boundaryMesh()[optimizationPatchIDs[optimizationPatchIDsI
          ]].meshPoints();
    }
    else if(fdAnalysis.match("selective"))
    {
      myPoints = globalIDsFD;
    }

    // Get the point normal vectors of the optPatch
    const vectorField normalPointVector = mesh.boundaryMesh()[
        optimizationPatchIDs[optimizationPatchIDsI]].pointNormals();

    // Save the initial mesh point positions
    const pointField initMeshPoints = mesh.points();

    // Point counter
    label pointCounter = myPoints.size();

    forAll(myPoints, myPointI)  // loop over all points on the optimization
        patch
    {
      ...
      // Get global point ID
      label globalID = myPoints[myPointI];
      ...

      for(scalar FDStepLoop = FDStepStart; FDStepLoop <= FDStepStop; FDStepLoop
          *= FDStepSize)    // loop over the step sizes
      {
        scalar FDStep = FDStepLoop;
        ...
        // Forward perturbation

        // Save the initial mesh points coordinates
        pointField pertMeshPoints = initMeshPoints;

        // Save initial position of point
        vector oldPos = initMeshPoints[globalID];

        // Define vector of perturbation direction
        vector pertVec = normalPointVector[myPointI];

        // Assign new position of point
        vector newPos = oldPos + FDStep*pertVec;

        // Update position in the mesh
        pertMeshPoints[globalID] = newPos;

        // Move mesh
        mesh.movePoints(pertMeshPoints);
```

```
        // Compute nuTilda
        scalar nuTildaForw = SensitivityAnalysis::calcnuTildaFD(mesh, runTime, U
            , p, dStar, nut, nuTilda, rho, cb1, cb2, kappa, sigma, cs, nu, cw1,
             cw2, cw3, cv1, objectiveFunction, maxItr, eps, relaxFac,
            caseExample, vectorForce, optimizationPatchIDsI,
            optimizationPatchIDs);


        // Backward perturbation

        // Store the initial mesh points coordinates
        pertMeshPoints = initMeshPoints;

        // Save initial position of point
        oldPos = initMeshPoints[globalID];

        // Assign new position of point i
        newPos = oldPos - FDStep*pertVec;

        // Update position in the mesh
        pertMeshPoints[globalID] = newPos;

        // Move mesh
        mesh.movePoints(pertMeshPoints);

        // Compute nuTilda
        scalar nuTildaBack = SensitivityAnalysis::calcnuTildaFD(mesh, runTime, U
            , p, dStar, nut, nuTilda, rho, cb1, cb2, kappa, sigma, cs, nu, cw1,
             cw2, cw3, cv1, objectiveFunction, maxItr, eps, relaxFac,
            caseExample, vectorForce, optimizationPatchIDsI,
            optimizationPatchIDs);

        // Calculate sensitivity
        scalar fdSens = (nuTildaForw - nuTildaBack)/(2.0*FDStep);

        // Assign values to fdSensField field
        fdSensField[globalID] = fdSens;
        ...
        // set mesh to initial state
        mesh.movePoints(initMeshPoints);

      } // step size loop
    } // points on opti patch loop
  }  // patches loop
  ...
} // Close FD sensitivity analysis
```

The implementation is basically the same as in the adjoint method but instead of calling `calcAdjointDotResidual`, the function `calcnuTildaFD` given in Lst. 6.10 is called.

Listing 6.10: Implementation of the function `calcnuTildaFD`

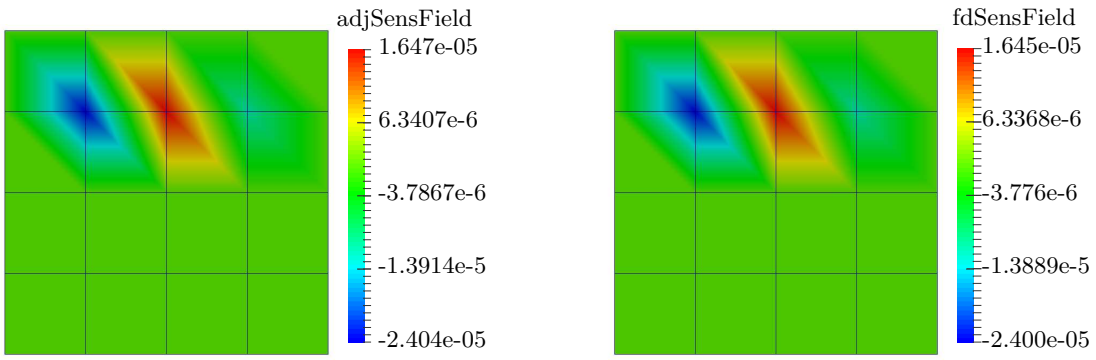
```
scalar calcnuTildaFD
(
...
)
{
  const volScalarField d = SpalartAllmaras::d(mesh);

  const surfaceScalarField phi = linearInterpolate(U) & mesh.Sf();

  label iter = 0;
  scalar maxResidual = 1.0;
```

```
  while (iter < maxItr && maxResidual > eps)
  {
    iter += 1.0;
    Info<< "\nIteration = " << iter << endl;

    nuTilda.storePrevIter();

    const volScalarField chi = SpalartAllmaras::chi(nuTilda, nu);
    const volScalarField fv1 = SpalartAllmaras::fv1(chi, cv1);
    const volScalarField Stilda = SpalartAllmaras::Stilda(chi, fv1, nuTilda, U,
        d, kappa, cs);

    fvScalarMatrix nuTildaEqnFD
    (
      // T1
      fvm::div(phi, nuTilda)

      // T2
      - fvm::SuSp(cb1*Stilda, nuTilda)

      // T3
      - (1.0 + cb2)*fvm::laplacian(SpalartAllmaras::nuTildaEff(nuTilda, sigma,
          nu), nuTilda)

      // T4
      + cb2*SpalartAllmaras::nuTildaEff(nuTilda, sigma, nu)*fvm::laplacian(
          nuTilda)

      // T5
      + fvm::SuSp(cw1*SpalartAllmaras::fw(Stilda, nuTilda, d, kappa, cw2, cw3)*
          nuTilda/sqr(d), nuTilda)
    );

    nuTildaEqnFD.relax(relaxFac);
    maxResidual = solve(nuTildaEqnFD, mesh.solver("nuTilda")).initialResidual();
    bound(nuTilda, dimensionedScalar("0", nuTilda.dimensions(), 0.0));
    nuTilda.correctBoundaryConditions();

    nut = nuTilda*fv1;
    nut.correctBoundaryConditions();
  }

  scalar toReturn = 0.0;

  if(objectiveFunction.match("artificial"))
  {
    ...
    else if(caseExample.match("unitCubeSpalartAllmaras"))
    {
      toReturn = nuTilda[nuTilda.size() - 1];
    }
  }

  return toReturn;
}
```

Note that for each perturbation the primal problem needs to be solved whereas in the adjoint method only post processing operations were used. The sensitivity is computed according to Eq. (3.16). The solution for the gradient field $\mathrm{d}L/\mathrm{d}\boldsymbol{x}$ for both methods is shown in Fig. 6.5.

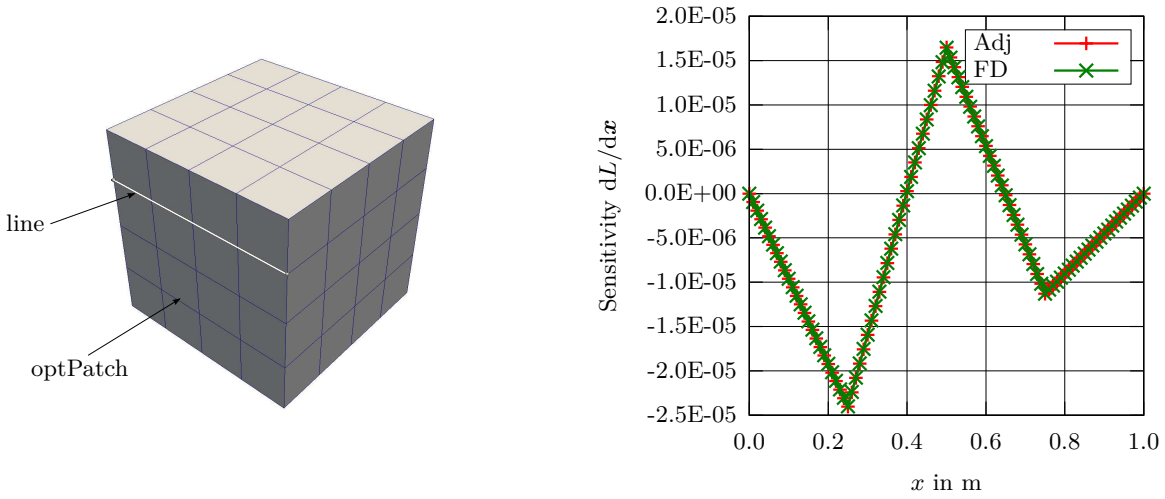
It is observed that both methods give the same results. A step size study is presented in the next section.

Another way of presenting the solution is to plot the sensitivities over a line as shown in Fig. 6.6.



(a) Adjoint method (b) Total finite difference method

Figure 6.5: Sensitivity contour plot for step size 1E−6 m Unit Cube



(a) Location of line (b) Results plotted over line

Figure 6.6: Sensitivity plot over line for step size 1E−6 m Unit Cube

6.5 Step Size Study

A step size study can be performed by setting the parameters `stepStart`, `stepStop` and `stepSize` as well as the perturbation points in the `optimizationProperties` file. The perturbation points can be selected via the `globalIDsAdj` and `globalIDsFD` key word for the adjoint and total finite difference method separately. For both, the adjoint and total finite difference method a step size study is carried for the `unitcubeSpalartAllmaras` case.

Two points from the hot spot region are picked, i.e. point 116 and 117. The result is given in Fig. 6.7.

The numbers next to the markers indicate the relative error in percent computed according to the following equation

$$\text{Relative error} = \frac{|\text{fdSens} - \text{adjSens}|}{|\text{fdSens}|} 100 \quad [\%] \tag{6.4}$$

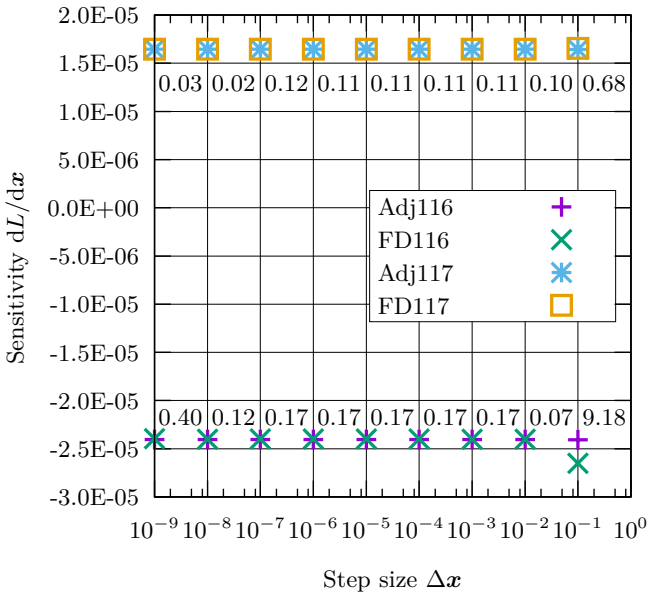


Figure 6.7: Step size study Unit Cube

It is observed that the relative error is small and the results are matching as expected but only for a rather large step size of 0.1 m the error is not acceptable. The step size should be chosen such that the mesh is not significantly disturbed, e.g. the mesh should not overlap.

7 Verification of the Adjoint Spalart-Allmaras Solver

In the following, the verification of the adjoint solver implementation is presented. The adjoint method is compared with a total finite difference approach computed according to Eq. (3.16). Furthermore a step size study is carried out. Three examples are considered, an internal channel flow in 2D for Re = 21 730, an external flow over the NACA 2412 airfoil in 2D for Re = 2.0E6 and an external flow over the ONERA M6 wing in 3D for Re ≈ 1.33E6.

7.1 Objective Function

For simplicity, the objective function is constructed such that the i-th entry of the RHS of Eq. (3.5) takes the value negative one and all other entries the value zero. This can be stated as

$$f_{\text{artificial}} = \sum_{i \in I} \tilde{\nu}_i \tag{7.1}$$

where $I \subset \mathbb{N}$ is a set of numbers containing the corresponding cell numbers.

7.2 Channel

As the first example, a channel flow is considered. The channel is adapted from [Arc11] for Re = 21 730, except here it is computed for two dimensions to reduce computational cost. The domain is of size 100 m × 1 m as shown in Fig. 7.1. Note that there is only one cell in z-direction as shown in Fig. 7.4(b). The mesh consists of 157 921 hexahedra cells and 320 000 nodes. The mesh non-orthogonality is zero.

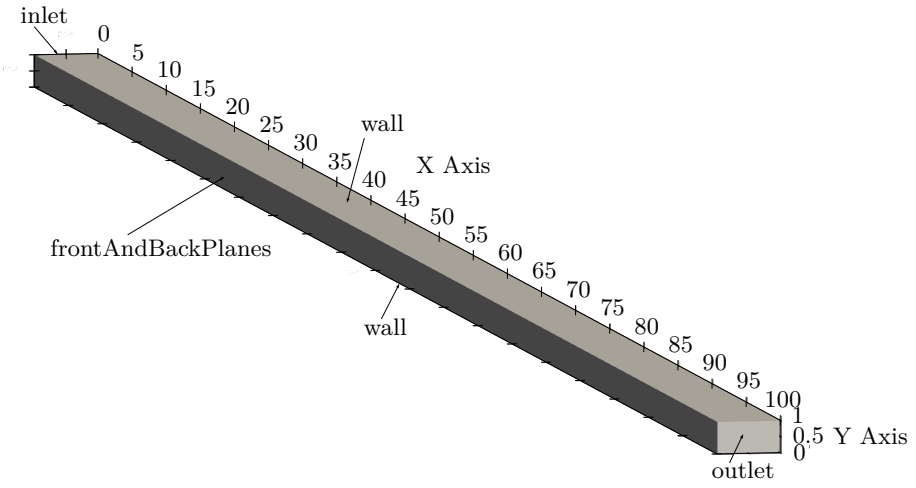


Figure 7.1: 2D domain with one cell in z-direction Channel

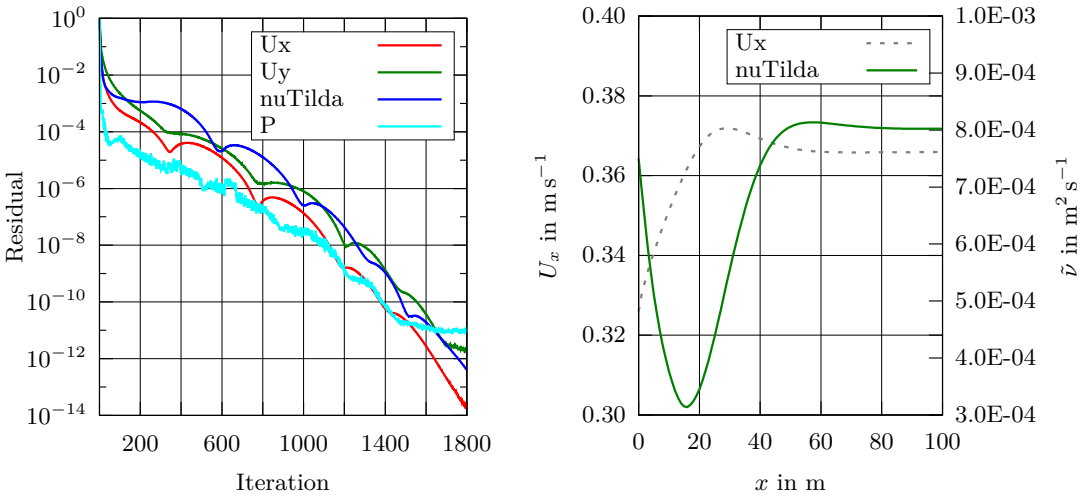
The fluid properties are the density $\rho = 1.0\,\mathrm{kg\,m^{-3}}$ and the kinematic viscosity $\nu = 1.5\text{E}{-5}\,\mathrm{m^2\,s^{-1}}$.

The inlet velocity is $U = U_x = 0.325\,95\,\mathrm{m\,s^{-1}}$ to match Reynolds number and desired $y^+ \approx 1$. The BCs are given in Tbl. 7.1.

Variable	Location	BC type	Value
P in $\mathrm{m^2\,s^{-2}}$	inlet	zeroGradient	-
	outlet	fixedValue	uniform 0
	wall	zeroGradient	-
	frontAndBackPlanes	empty	-
U in $\mathrm{m\,s^{-1}}$	inlet	fixedValue	uniform (0.32595 0 0)
	outlet	zeroGradient	-
	wall	fixedValue	uniform (0 0 0)
	frontAndBackPlanes	empty	-
ν_t in $\mathrm{m^2\,s^{-1}}$	inlet	fixedValue	uniform 75E−5
	outlet	zeroGradient	-
	wall	fixedValue	uniform 0
	frontAndBackPlanes	empty	-
$\tilde{\nu}$ in $\mathrm{m^2\,s^{-1}}$	inlet	fixedValue	uniform 75E−5
	outlet	zeroGradient	-
	wall	fixedValue	uniform 0
	frontAndBackPlanes	empty	-

Table 7.1: Boundary conditions Channel

The `simpleFOAM` solver is invoked and the residuals are plotted in Fig. 7.2(a). The solution is converged to an order of magnitude of at least 10^{-11} at 1800 iterations. In Fig. 7.2(b) results are plotted for the velocity in x-direction U_x and the Spalart-Allmaras variable $\tilde{\nu}$ at the centerline of the channel, i.e. at $y = 0.5\,\mathrm{m}$. Note that towards the end of the channel, results are not changing anymore w.r.t. x, i.e. for $x \gtrsim 95\,\mathrm{m}$. The flow has fully developed.

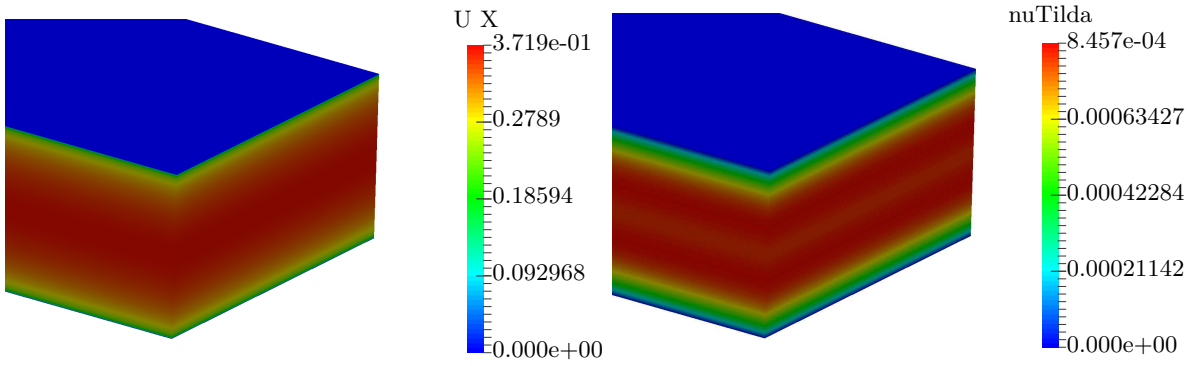


(a) Residuals

(b) CFD results at center line

Figure 7.2: CFD solution Channel

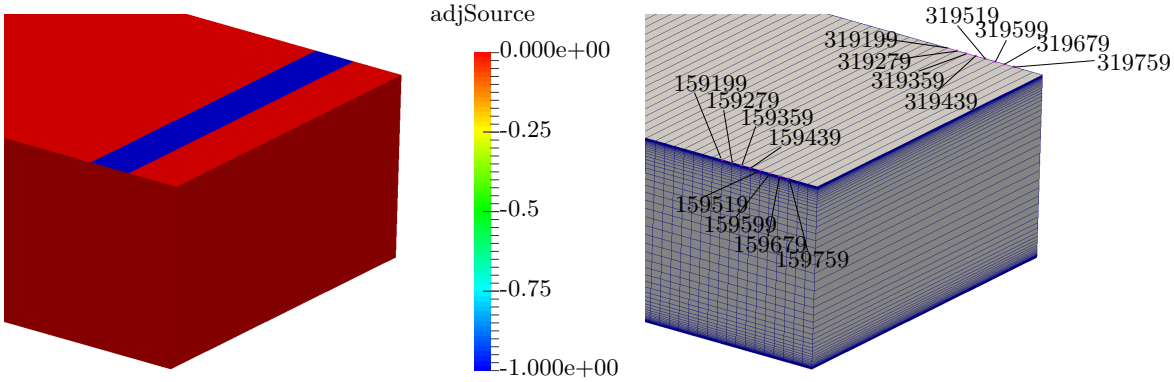
Fig. 7.3 shows the distribution over the height of the channel for the velocity in x-direction U_x and the Spalart-Allmaras variable $\tilde{\nu}$ at the end of the channel. For the velocity a typical parabolic like profile is observed with the highest velocity in the middle of the channel, i.e. for $y = 0.5\,\mathrm{m}$.



(a) Velocity U_x (b) Spalart-Allmaras variable $\tilde{\nu}$

Figure 7.3: CFD solution distribution Channel

For the objective function four cells are chosen at the end of the channel, the set of numbers for the objective is $I = \{157288, 157367, 157446, 157525\}$. The source term is visualized in Fig. 7.4(a).



(a) Adjoint source term (b) Perturbation nodes

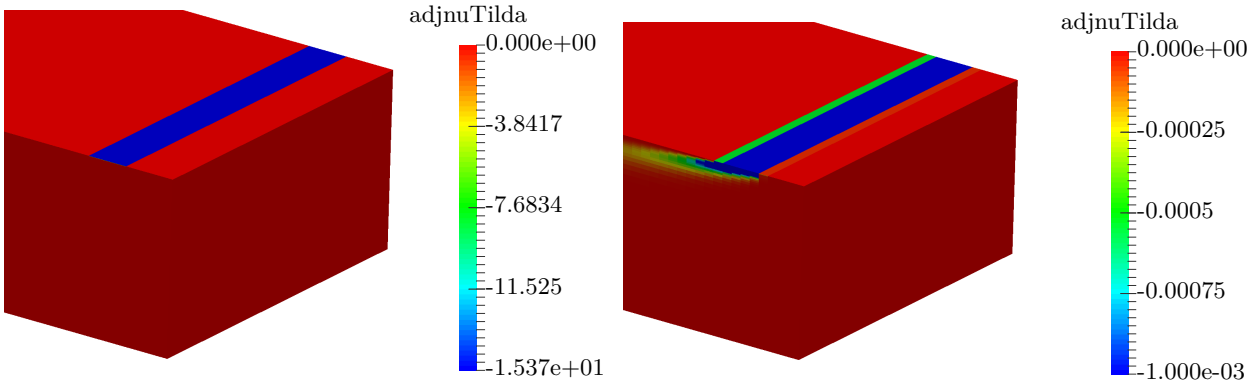
Figure 7.4: Adjoint source term and perturbation nodes Channel

The solution for the adjoint problem is presented in Fig. 7.5. Note that in Fig. 7.5(a) the solution is given for the full range, whereas in Fig. 7.5(b) the solution is scaled to $[0.0, 1.0\text{E}{-3}]$. In the scaled solution it is observed that the solution is developed in the reversed flow direction. Note also that for the solution with full range the distribution can not be seen.

The shape sensitivities are computed for the nodes depicted in Fig. 7.4(b). They are obviously located around the source term cells. The shape sensitivities are shown in Fig. 7.6 for a step size of $1.0\text{E}{-6}$ m. As expected, both solution are equal with minor difference in numerical values. There is no physical interpretation in the solution since the objective function is chosen artificially. It is to be shown that the solution are equivalent.

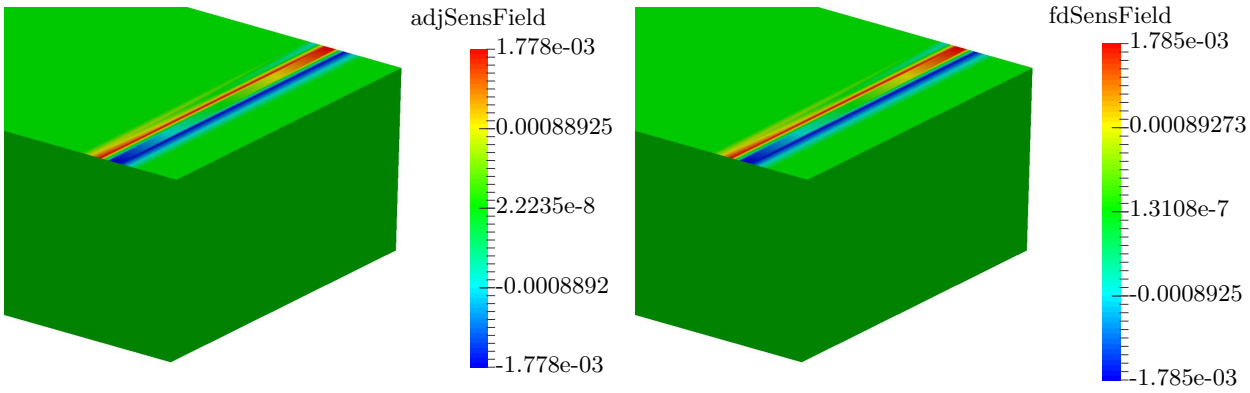
A step size study is carried out for node 159 359 and 319 519. They are picked such that they are in the positive and negative hot spot area, cf. Fig. 7.4(b) and 7.6. The results are plotted in Fig. 7.7. The numbers next to the markers indicate the absolut relative error in percent computed according to Eq. (6.4). Note that the biggest step size is $1.0\text{E}{-4}$ m, that is because the first grid point is located $1.06\text{E}{-4}$ m from the wall. The error is acceptable for all step sizes and smallest for a step size of $1.0\text{E}{-9}$ m. It is not sensitive w.r.t. the step size.

The solution for the adjoint and total finite difference method are matching very well, concluding that the adjoint solver for this example is verified.



(a) Adjoint solution full range (b) Adjoint solution scaled to $[0.0, 1.0\text{E}{-3}]$

Figure 7.5: Adjoint solution Channel



(a) Adjoint method (b) Total finite difference method

Figure 7.6: Sensitivity Channel

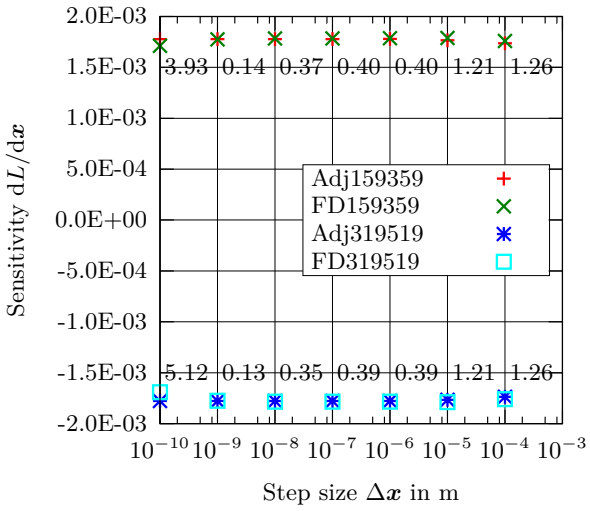


Figure 7.7: Steps size study Channel

7.3 NACA 2412 Airfoil

As the second example, the NACA 2412 airfoil is considered. The primal solution was already presented and discussed in chapter 5. In the following the adjoint solution as well as the shape sensitivity analysis is presented.

For the objective function the set I is chosen as the set of numbers corresponding to all cells adjacent to the airfoil. The source term is visualized in Fig. 7.8 for the leading edge of the airfoil.

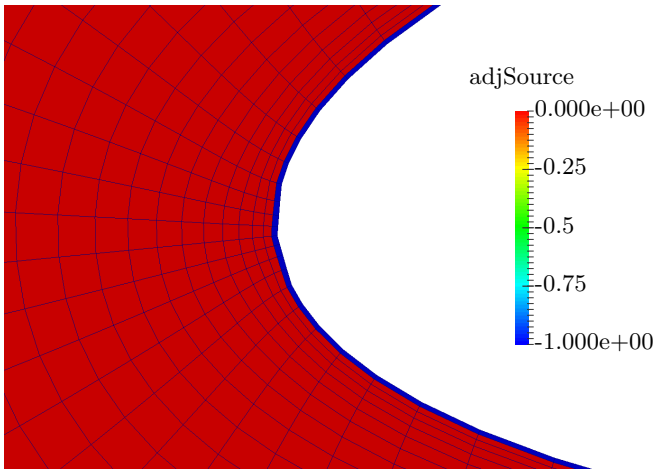
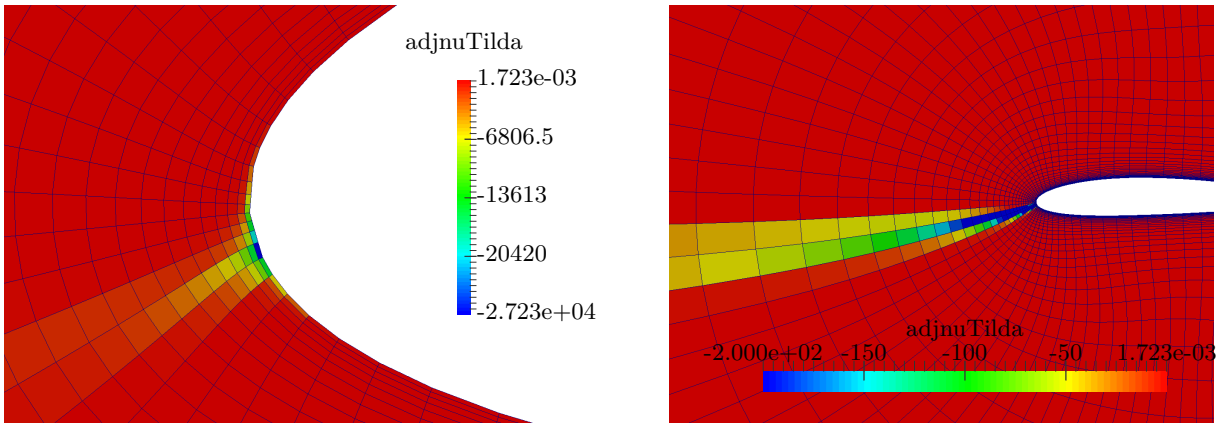


Figure 7.8: Adjoint source term NACA 2412

The adjoint solution is presented in Fig. 7.9. Note that in Fig. 7.9(a) the solution is given for the full range, whereas in Fig. 7.9(b) the solution is scaled to $[-2.0\text{E}2, 1.723\text{E}{-3}]$. It is observed that the solution is developed in the reversed flow direction. The highest value is a the leading edge, i.e. where the flow hits the airfoil perpendicular.

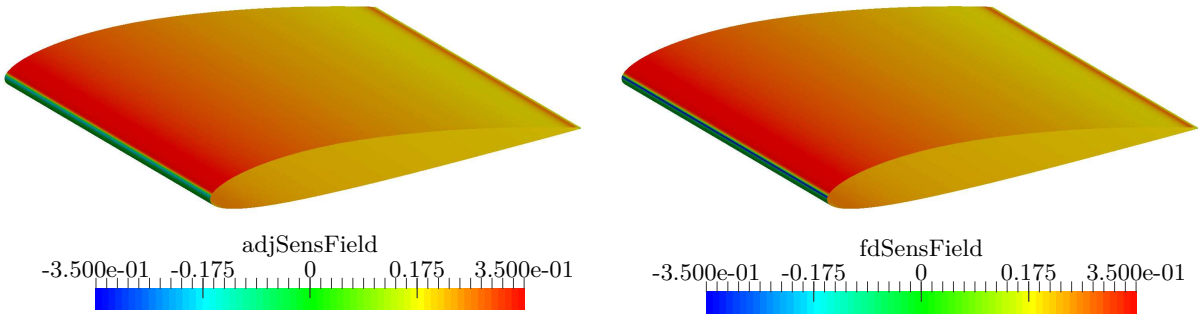


(a) Adjoint solution full range

(b) Adjoint solution scaled to $[-2.0\text{E}2, 1.723\text{E}{-3}]$

Figure 7.9: Adjoint solution NACA 2412

The shape sensitivities are computed for all nodes around the airfoil for both methods, adjoint and total finite difference method. A contour plot of the sensitivities is shown in Fig. 7.10 for a step size of $1.0\text{E}{-7}$ m. The solution is scaled to a range of $[-0.35, 0.35]$. The solution is also compared in Fig. 7.11 where the arc length starts from the trailing edge and goes around the airfoil in clockwise direction. The trailing edge is at zero and two meter and the leading edge at one meter. Visually the solution seems to match for both methods as expected. However, there



(a) Adjoint method (b) Total finite difference method

Figure 7.10: Sensitivity scaled NACA 2412

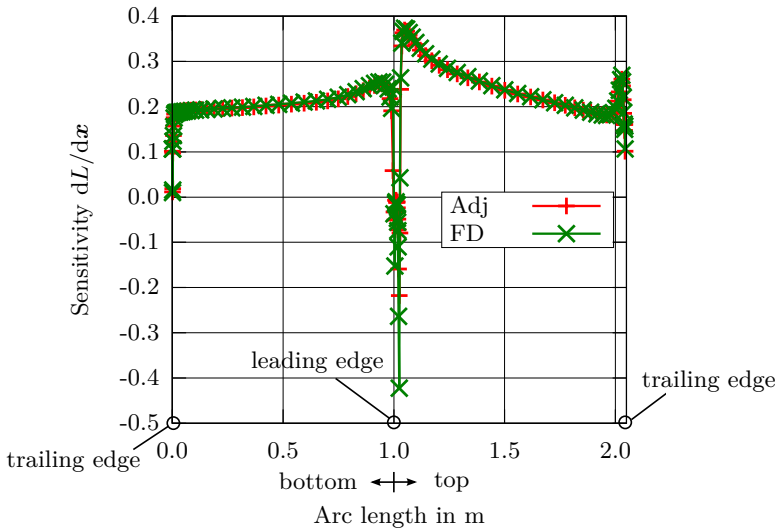
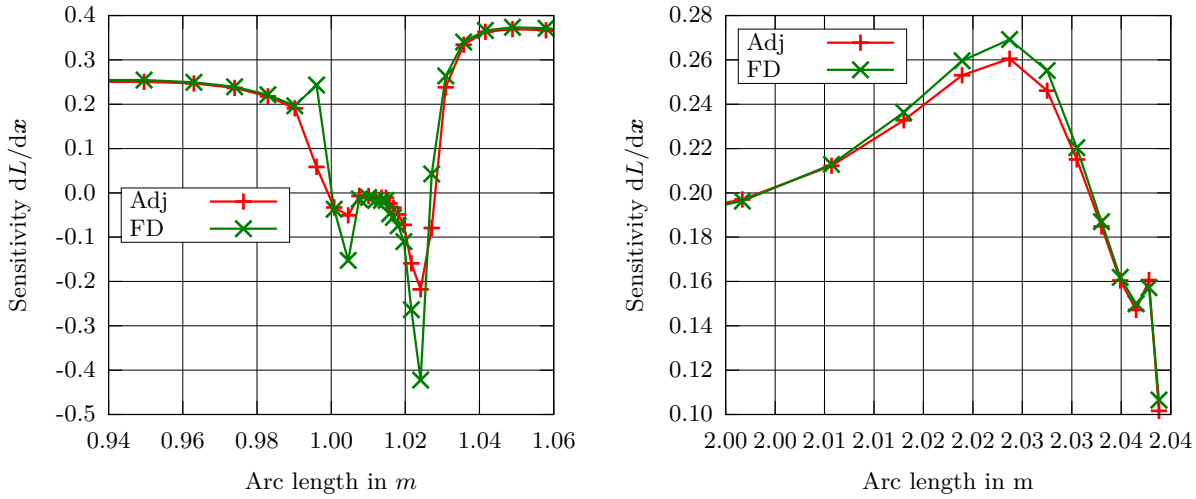


Figure 7.11: Sensitivity plot for step size 1.0E−7 m NACA 2412

are some differences in the solution at the leading and trailing edge as depicted in Fig. 7.12. Fig. 7.12(a) shows the solution at the leading edge and Fig. 7.12(b) the solution at the trailing edge. The difference might be due to the non-orthogonality of the mesh. It has been mentioned before that the Jacobi matrix is exact only for orthogonal meshes. However, the trend is captured and only minor differences are observed for only a very small region. Moreover the trailing and leading edge might not even be of interest.

A step size study is carried out for node 6222 and 6239. They are picked such that they are in the positive hot spot region as well as in a moderate region, cf. Fig. 7.13 and 7.10. The results are plotted in Fig. 7.14. Note that the biggest step size is 1.0E−4 m due to the mesh size at the airfoil. The numbers next to the markers indicate the absolute relative error in percent computed according to Eq. (6.4). The error is acceptable for all step sizes. The solution is not sensitive w.r.t. the step size with exception for the step size 1.0E−4 m.

The solution for the adjoint and total finite difference method are matching very well except for the trailing and leading edge, concluding that the adjoint solver for this example is verified.



(a) Leading edge (b) Trailing edge

Figure 7.12: Sensitivity at leading and trailing edge NACA 2412

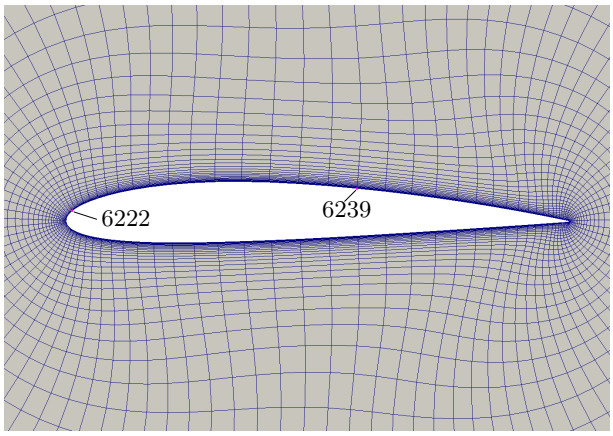


Figure 7.13: Perturbation points for the step size study NACA 2412

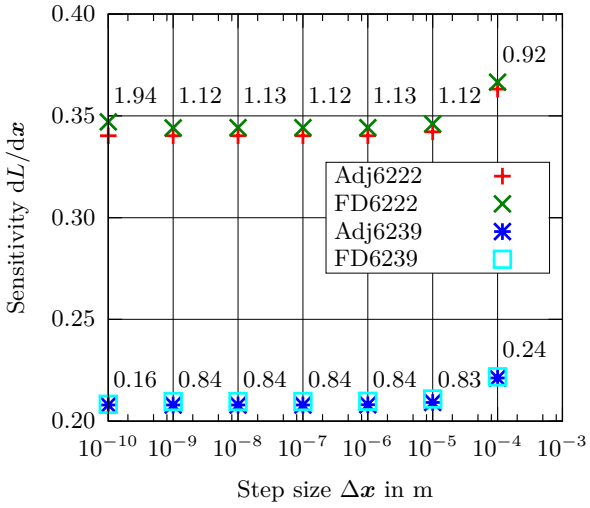


Figure 7.14: Steps size study NACA2412

7.4 ONERA M6 Wing

As the third example, the ONERA M6 wing is considered. The primal solution was already presented and discussed in chapter 5. In the following the adjoint solution as well as the shape sensitivity analysis is presented.

For the objective function the set I is chosen as the set of numbers corresponding to all faces adjacent to the airfoil. The source term is visualized for a cut in the xy-plane at $z = 0.2\,\mathrm{m}$ in Fig. 7.15.

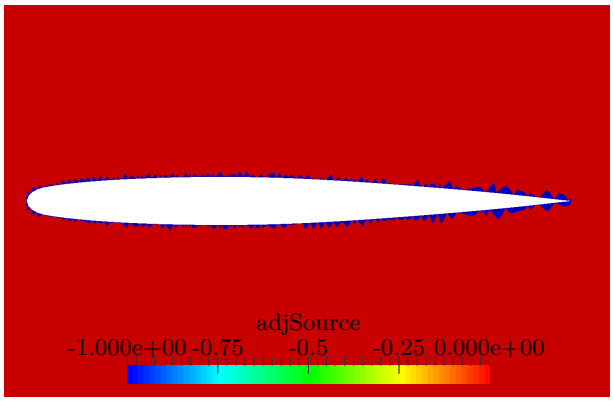
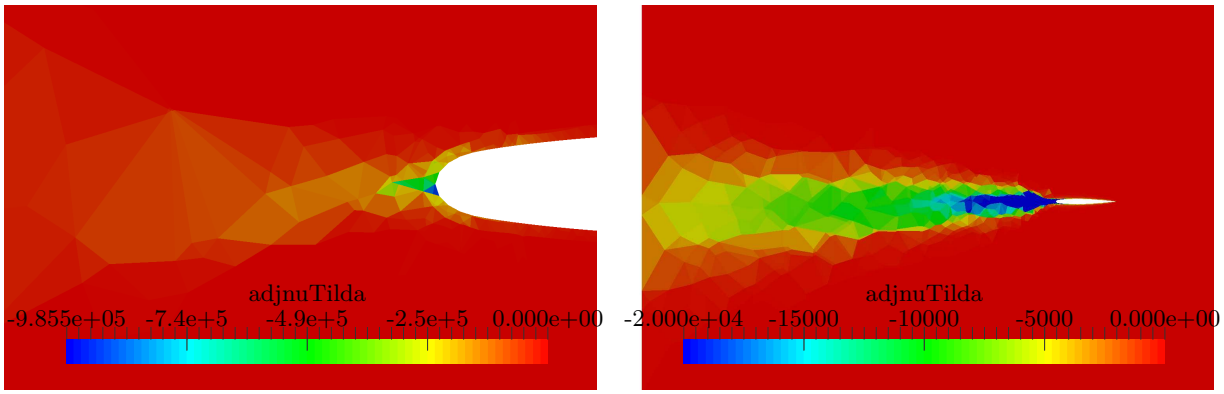


Figure 7.15: Adjoint source term in the xy-plane at $z = 0.2\,\mathrm{m}$ ONERA M6

The adjoint solution is presented in Fig. 7.16 for a cut in the xy-plane at $z = 0.2\,\mathrm{m}$. Note that in



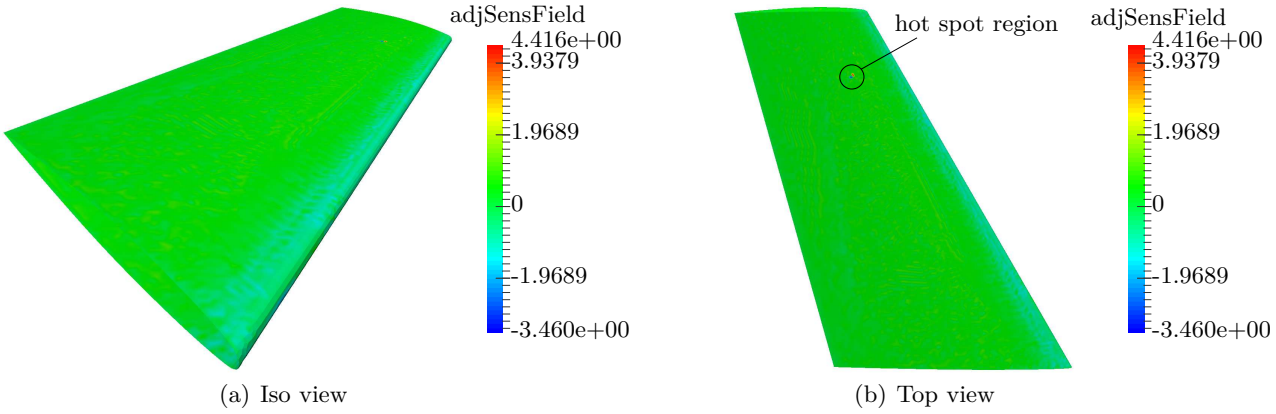
(a) Adjoint solution full range

(b) Adjoint solution scaled to $[-2.0\text{E}4, 0.0]$

Figure 7.16: Adjoint solution in the xy-plane at $z = 0.2\,\mathrm{m}$ ONERA M6

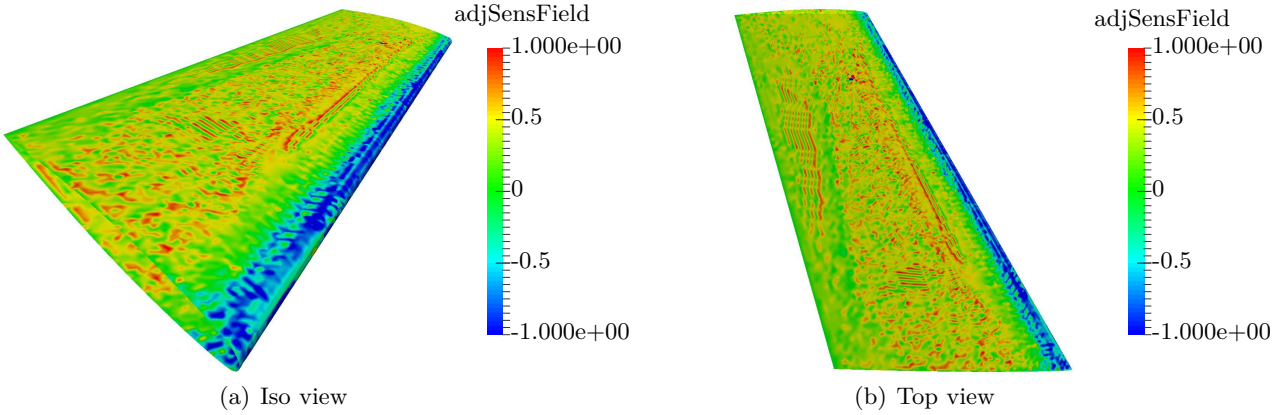
Fig. 7.16(a) the solution is given for the full range, whereas in Fig. 7.16(b) the solution is scaled to$[-2.0\text{E}4, 0.0]$. It is observed that the solution is developed in the reversed flow direction. The same behaviour has been recognized for the NACA2412 airfoil and the channel.

The shape sensitivities are computed for all nodes at the wing but only for the adjoint method with a step size of $1.0\text{E}{-8}\,\mathrm{m}$. A contour plot for iso and top view is shown in Fig. 7.17 and 7.18. Note that the solution is scaled to a range of $[-1.0, 1.0]$ in Fig. 7.18, where the solution seems more reasonable. Note the depicted hot spot region in Fig. 7.17(b). Compared to the other points the result seems not correct since usually the sensitivities do not vary this much from point to point. This error might result out of the course mesh. It might also be the case that the error is due to the non-orthogonality of the mesh as discussed in chapter 10. The solution might be improved by mesh refinement. From this region, two nodes 21 594 and 21 595 are picked for a step size study and results are plottet Fig. 7.19. It is observed that the sensitivities are sensitive



(a) Iso view (b) Top view

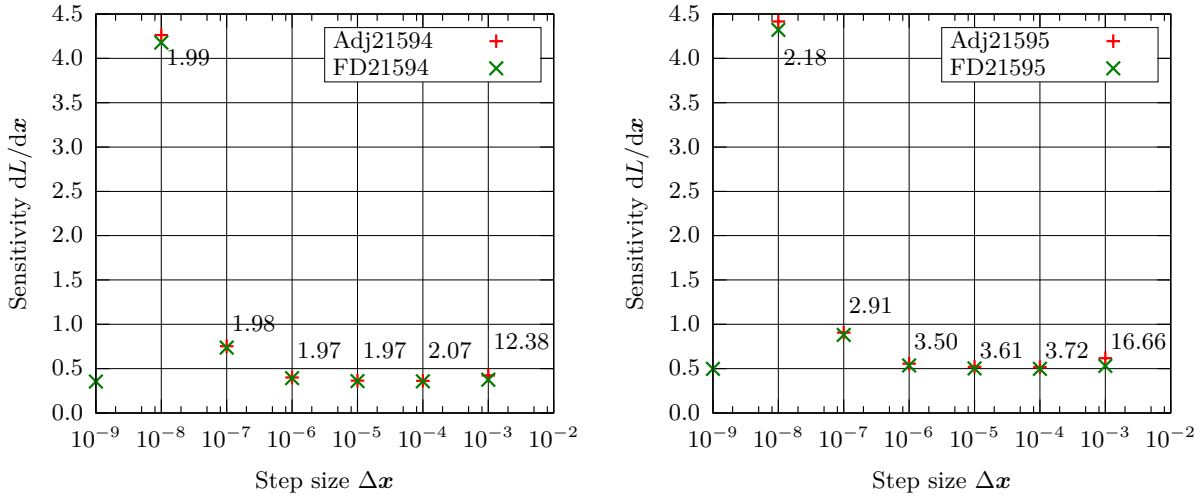
Figure 7.17: Adjoint sensitivity for ONERA M6



(a) Iso view (b) Top view

Figure 7.18: Adjoint sensitivity scaled to $[-1.0, 1.0]$ ONERA M6

w.r.t. the step size, especially for the computed step size of $1.0\text{E}{-8}\,\text{m}$. However, in comparison to the total finite difference method, the absolute relative error in percent computed according to Eq. (6.4) is acceptable, concluding that the adjoint solver for this example is verified.



(a) Node 21594 (b) Node 21595

Figure 7.19: Steps size study ONERA M6

7.5 Remark on the Adjoint Solver

It has been observed, that the solution of the adjoint problem is sensitive w.r.t. the discretization scheme. Especially the divergence schemes are of importance. Initially the scheme `bounded Gauss upwind` is a good start, however, `bounded Gauss linear`, `Gauss linear` or `Gauss upwind` might also be used to get correct results. The `upwind` schemes are more robust but less accurate, the `linear` schemes are more accurate but less stable. The proper choice of schemes depends on the problem.

It has also been found, that the residual in the total finite difference approach should drop to an order of magnitude of at least 1.0E−10, otherwise results might not be correct. Again, this also depends on the problem.

8 OpenFOAM Version Issue

The implementation and verification of the Spalart-Allmaras adjoint equation was presented in chap. 6 and 7. It has been emphasized, that all spatial operators are needed in implicit form, otherwise the adjoint equation cannot be setup. However, it has been observed, that the implicit Laplace operator (`fvm::laplacian(nuTilda_)`) showed wrong results in dependence on the OpenFOAM version. To emphasize the problem, Eq. (4.6) is repeated in Lst. 8.1.

Listing 8.1: Implementation of the reformulated Spalart-Allmaras Eq. (4.6) in OpenFOAM

```
...
tmp<fvScalarMatrix> nuTildaEqn
(
    fvm::ddt(alpha, rho, nuTilda_)
  + fvm::div(alphaRhoPhi, nuTilda_)
  - (1.0 + Cb2_)*fvm::laplacian(alpha*rho*DnuTildaEff(), nuTilda_)
  + Cb2_*alpha*rho*DnuTildaEff()*fvm::laplacian(nuTilda_)
  ==
    Cb1_*alpha*rho*Stilda*nuTilda_
  - fvm::Sp(Cw1_*alpha*rho*fw(Stilda)*nuTilda_/sqr(y_), nuTilda_)
);
...
```

Note that the Laplace operator in line seven is used in implicit form.

In the following, the pitzDaily tutorial case for a steady turbulent flow over a backward facing step in 2D is used to show the discrepancy in results. The case files can be found online.[12] OpenFOAM version 2.3.1, 2.4, 3.0+, 3.0.1, 4.0, 3.1ext and 3.2ext have been tested, where ext stands for the OpenFOAM Extend Project. It might be already mentioned, that only version 2.4, 3.0+, 3.0.1 and 4.0 gave the correct results.

Although the Spalart-Allmaras turbulence model might not be the proper choice to model the correct physics for the backward facing step problem, it is chosen for convenience. However, a comparison to the standard $k-\epsilon$ turbulence model, known to give good physical results, is presented as a reference solution.

The geometry and mesh is shown in Fig. 8.1. The height at the inlet is 0.0254 m and at the outlet

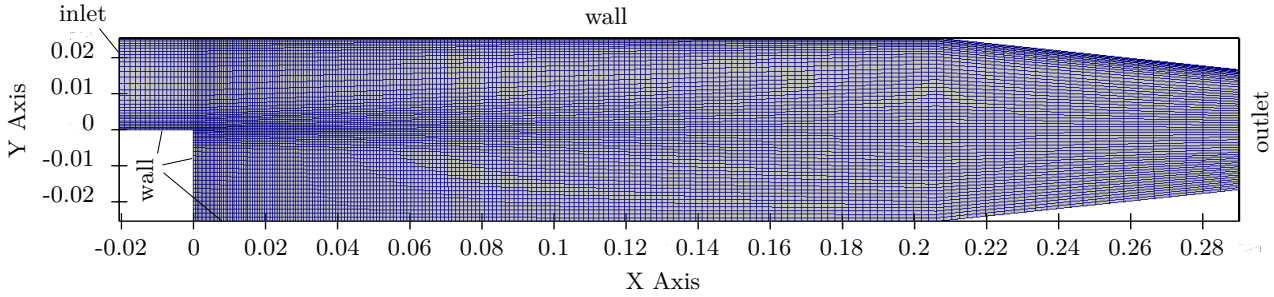


Figure 8.1: Geometry and Mesh pitzDaily

0.0332 m, the length of the step is 0.0206 m and the height is 0.0254 m. Total height of the channel is 0.0508 m and total length is 0.311 m. The inlet velocity magnitude is $U = U_x = 10.0\,\mathrm{m\,s^{-1}}$, the

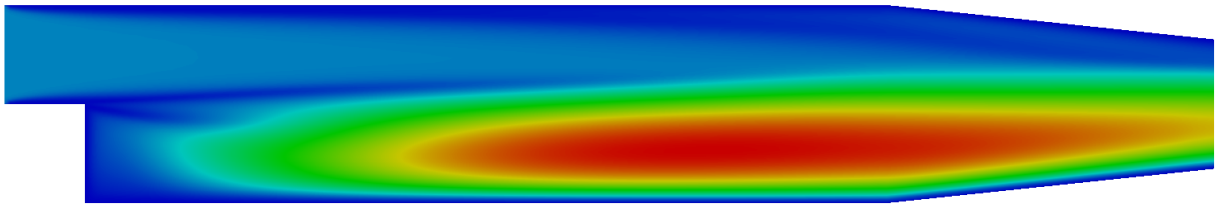
[12] `https://github.com/DennisKasper/OpenFOAM301/tree/master/run`

y-component is zero. The kinematic viscosity is $\nu = 1.0\text{E}{-5}\,\text{m}^2\,\text{s}^{-1}$ and the Reynolds number

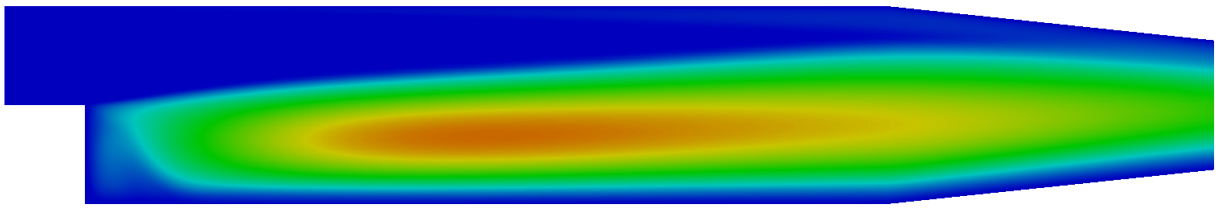
$$\text{Re} = \frac{10.0\,\text{m}\,\text{s}^{-1} 0.0254\,\text{m}}{1.0\text{E}{-5}\,\text{m}^2\,\text{s}^{-1}} = 25\,400.$$

Note that the boundary layer is not properly resolved ($y^+ \gg 1$) and therefore a wall function (nutkWallFunction, nutWallFunction resp. nutUSpaldingWallFunction) is used.

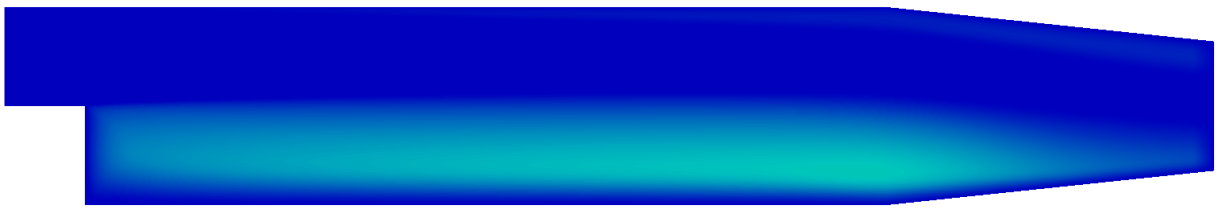
The `simpleFoam` solver is invoked and all solutions are converged. Contour plots are shown for the turbulent viscosity field ν_t and the velocity magnitude field U in Fig. 8.2 and 8.3 respectively. From top to bottom the solutions are given for the $k-\epsilon$ model computed with OpenFOAM v3.0.1, followed by the Spalart-Allmaras model computed with OpenFOAM v3.0.1 and OpenFOAM v3.1ext. Note the significant difference between the two solutions computed with the Spalart-



(a) $k-\epsilon$ model OpenFOAM v3.0.1



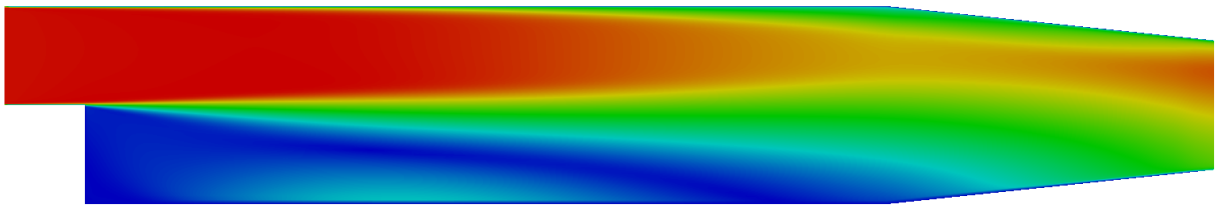
(b) Spalart-Allmaras model OpenFOAM v3.0.1



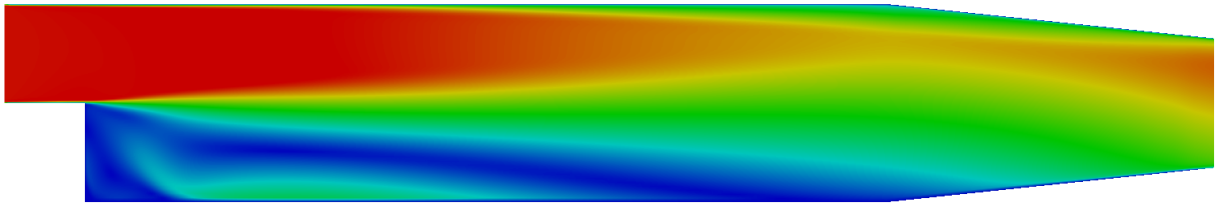
(c) Spalart-Allmaras model OpenFOAM v3.1ext

Figure 8.2: Turbulent viscosity field ν_t pitzDaily

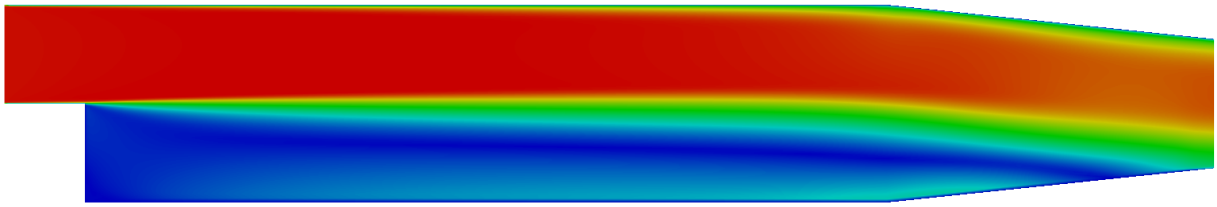
Allmaras model. Again, resulting only from using different OpenFOAM versions. Comparing the solutions to the $k-\epsilon$ model, which is considered as a reference solution, it is obvious that the OpenFOAM v3.1ext computes incorrect results. This can be seen most clearly in the turbulent viscosity field ν_t in Fig. 8.2, where the solutions with OpenFOAM v3.0.1 is qualitatively close to the reference solution, but the solution with OpenFOAM v3.1ext is not close at all. This also applies for the velocity field in Fig. 8.3. This behavior has also been observed with OpenFOAM v2.3.1 and v3.2ext. Concluding that only OpenFOAM v2.4, v3.0+, v3.0.1 and v4.0 can be used to compute the adjoint problem for the Spalart-Allmaras model.



(a) $k - \epsilon$ model OpenFOAM v3.0.1



(b) Spalart-Allmaras model OpenFOAM v3.0.1



(c) Spalart-Allmaras model OpenFOAM v3.1ext

Figure 8.3: Velocity magnitude field U pitzDaily

Remark

By changing the Laplace operator from implicit to explicit form, i.e. `fvc::laplacian(nuTilda_)` in Lst. 8.1 line seven, all OpenFOAM versions computed the correct results for Eq. (4.6). However, to setup the adjoint equation, the Laplace operator is needed in implicit form and therefore one needs to chose the proper OpenFOAM version. Furthermore the source code of the Laplace operator has been compared for the different OpenFOAM versions but no coding errors have been found, however, it has been observed, that the namespace `incompressible` has been removed.

9 How to Implement a new User defined Turbulence Model in OpenFOAM v3.0.1

In order to make changes to the source code in OpenFOAM, it is good practice to setup the users own files, e.g. for a new solver, a new boundary condition or a new turbulence model. Also it is recommended to keep the same directory tree structure. How this can be done, is more or less well documented online, e.g. to implement a user defined RAS based k-epsilon turbulence model, look up the online course "MSc/PhD course in CFD with OpenSource software, 7.5hec , 2015" by Chalmers University of Technology, Dept of Thermo and Fluid Dynamics.[13]

However, since OpenFOAM recently updated to version 3.0.1 and 4.0, the implementation for turbulence models has been changed. Roughly speaking, the namespaces incompressible and compressible have been removed.

In the following, a guideline is presented how to implement the mySpalartAllmaras turbulence model in OpenFOAM v3.0.1. It has also been successfully tested for OpenFOAM v4.0.

The directory tree for the Spalart-Allmaras turbulence model in OpenFOAM v3.0.1 reads as follows

```
OpenFOAM/
├── <user>-3.0.1/
└── OpenFOAM-3.0.1/
    └── src/
        └── TurbulenceModels/
            ├── compressible/
            ├── incompressible/
            │   ├── Make/
            │   │   ├── files
            │   │   └── options
            │   └── incompressibleTurbulenceModel.C
            ├── phaseCompressible/
            ├── phaseIncompressible/
            ├── turbulenceModels/
            │   ├── Make/
            │   │   ├── files
            │   │   └── options
            │   ├── RAS/
            │   │   └── SpalartAllmaras/
            │   │       ├── SpalartAllmaras.C
            │   │       └── SpalartAllmaras.H
            │   ├── turbulenceModel.C
            │   └── turbulenceModel.H
            └── Allwmake
```

[13] http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2008/implementTurbulenceModel.pdf (15.07.2016)

To create a new user defined Spalart-Allmaras turbulence model, copy the necessary files into the `<user>-3.0.1` folder as follows.

```
OpenFOAM/
├── <user>-3.0.1/
│   └── TurbulenceModels/
│       ├── incompressible/
│       │   ├── Make/
│       │   │   ├── files
│       │   │   └── options
│       │   └── myIncompressibleTurbulenceModels.C
│       ├── turbulenceModels/
│       │   ├── Make/
│       │   │   ├── files
│       │   │   └── options
│       │   ├── RAS/
│       │   │   └── mySpalartAllmaras/
│       │   │       ├── mySpalartAllmaras.C
│       │   │       └── mySpalartAllmaras.H
│       │   ├── turbulenceModel.C
│       │   └── turbulenceModel.H
│       └── Allwmake
└── OpenFOAM-3.0.1/
```

Note that the folder `SpalartAllmaras`, as well as the files `incompressibleTurbulenceModel.C`, `SpalarAllmaras.C` and `SpalartAllmaras.H` have been renamed to `mySpalartAllmaras`, `myIncompressibleTurbulenceModels.C`, `mySpalartAllmaras.C` and `mySpalartAllmaras.H` respectively. Now make the following changes.

Listing 9.1: `OpenFOAM/<user>-3.0.1/TurbulenceModels/incompressible/Make/files`

```
myIncompressibleTurbulenceModels.C

LIB = $(FOAM_USER_LIBBIN)/mylibIncompressibleTurbulenceModels
```

Listing 9.2: `OpenFOAM/<user>-3.0.1/TurbulenceModels/incompressible/Make/options`

```
EXE_INC = \
    -I../turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/incompressible/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude

LIB_LIBS = \
    -lincompressibleTransportModels \
    -lturbulenceModels \
    -lfiniteVolume \
    -lmeshTools
```

Listing 9.3: `OpenFOAM/<user>-3.0.1/TurbulenceModels/incompressible/myIncompressibleTurbulenceModels.C`

```
...
// -------------------------------------------------------------------------- //
// RAS models
// -------------------------------------------------------------------------- //
```

```
#include "mySpalartAllmaras.H"
makeRASModel(mySpalartAllmaras);
```

Listing 9.4: OpenFOAM/<user>-3.0.1/TurbulenceModels/turbulenceModels/Make/files

```
turbulenceModel.C

LIB = $(FOAM_USER_LIBBIN)/mylibTurbulenceModels
```

Listing 9.5: OpenFOAM/<user>-3.0.1/TurbulenceModels/turbulenceModels/Make/options

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools
```

Listing 9.6: OpenFOAM/<user>-3.0.1/TurbulenceModels/turbulenceModels/RAS/mySpalartAllmaras/mySpalartAllmaras.C

```
...
template<class BasicTurbulenceModel>
void mySpalartAllmaras<BasicTurbulenceModel>::correct()
{
    if (!this->turbulence_)
    {
        return;
    }

    // Local references
    const alphaField& alpha = this->alpha_;
    const rhoField& rho = this->rho_;
    const surfaceScalarField& alphaRhoPhi = this->alphaRhoPhi_;

    eddyViscosity<RASModel<BasicTurbulenceModel> >::correct();

    const volScalarField chi(this->chi());
    const volScalarField fv1(this->fv1(chi));

    const volScalarField Stilda(this->Stilda(chi, fv1));

    Info<< "Solving mySpalartAllmaras nuTildaEqn" << endl;

    tmp<fvScalarMatrix> nuTildaEqn
    (
        fvm::ddt(alpha, rho, nuTilda_)
      + fvm::div(alphaRhoPhi, nuTilda_)
      - (1.0 + Cb2_)*fvm::laplacian(alpha*rho*DnuTildaEff(), nuTilda_)
      + Cb2_*alpha*rho*DnuTildaEff()*fvm::laplacian(nuTilda_)
      ==
        Cb1_*alpha*rho*Stilda*nuTilda_
      - fvm::Sp(Cw1_*alpha*rho*fw(Stilda)*nuTilda_/sqr(y_), nuTilda_)
    );

    nuTildaEqn().relax();
    solve(nuTildaEqn);
    bound(nuTilda_, dimensionedScalar("0", nuTilda_.dimensions(), 0.0));
    nuTilda_.correctBoundaryConditions();

    correctNut(fv1);
}
...
```

Note that within the files `mySpalartAllmaras.C` and `mySpalartAllmaras.H`, all appearances of `SpalartAllmaras` have to be renamed to `mySpalartAllmaras`.

Listing 9.7: `OpenFOAM/<user>-3.0.1/TurbulenceModels/Allwmake`

```
#!/bin/sh
cd ${0%/*} || exit 1    # Run from this directory

# Parse arguments for library compilation
targetType=libso
. $WM_PROJECT_DIR/wmake/scripts/AllwmakeParseArguments
set -x

wmake $targetType turbulenceModels
wmake $targetType incompressible

# ----------------------------------------------------------------- end-of-file
```

Now the code can be compiled by executing `./Allwmake` within the `TurbulenceModels` folder. Two new library files are created in the `lib` folder

```
OpenFOAM/
└── <user>-3.0.1/
    └── platforms/
        └── linux64GccDPInt32Opt/.....................(linux64GccDPInt32Debug/)
            └── lib/
                ├── mylibIncompressibleTurbulenceModels.so
                └── mylibTurbulenceModels.so
```

To make the new turbulence model available at run time, the new library has to be added to the `system/controlDict` dictionary within the case folder

```
...
libs
(
  "mylibIncompressibleTurbulenceModels.so"
)
```

And finally, the new turbulence model is selected in the `constant/turbulenceProperties` dictionary.

```
...
simulationType RAS;

RAS
{
    RASModel        mySpalartAllmaras;

    turbulence      on;

    printCoeffs     on;
}
```

10 Conclusions

A discrete adjoint solver for the Spalart-Allmaras turbulence model for high Reynolds number flow has been implemented in OpenFOAM v3.0.1. The RANS equations together with the Spalart-Allmaras turbulence model equation are presented to describe the turbulent flow. The discrete adjoint equation for the Spalart-Allmaras turbulence model as well as the adjoint equations for a coupled system of RANS and Spalart-Allmaras has been derived. The Spalart-Allmaras turbulence model is presented and its discrete derivative is derived on a term by term basis. The implementation of the discrete adjoint solver in OpenFOAM has been presented and discussed. Its functionality has been verified against a total finite difference approach for an artificial objective function for three flow examples, i.e. a channel flow for $\mathrm{Re} = 21\,730$, a flow over a NACA 2412 airfoil in 2D for $\mathrm{Re} = 2.0\text{E}6$ and a flow over the ONERA M6 wing in 3D for $\mathrm{Re} \approx 1.33\text{E}6$. The solver has been found to show very good results compared to the total finite difference approach. However, there has been some issue with the implicit Laplace operator, i.e. only the orthogonal part is discretized in implicit form, whereas the non-orthogonal part is discretized explicitly. Moreover, for some OpenFOAM version the Laplace operator showed quit wrong results. However, for all examples in this thesis the effect of discretization was negligible. Finally, a guideline on how to implement a new turbulence model is presented. It was necessary because in recent OpenFOAM versions, changes have been made to the source code such that the available tutorials online were not applicable.

11 Outlook

In the following suggestions are made for further development and improvement of the discrete adjoint solver.

Implementing the non-orthogonal part for the implicit Laplace operator in implicit form

The Laplacian term is integrated over a control volume and linearised as follows

$$\int_V \nabla \cdot (\Gamma \nabla \phi) \mathrm{d}V = \int_S (\Gamma \nabla \phi) \cdot \mathrm{d}\boldsymbol{S} = \sum_f \Gamma_f (\nabla \phi)_f \cdot \boldsymbol{S}_f \tag{11.1}$$

where Γ_f is the diffusion coefficient at the cell face, $(\nabla\phi)_f$ the face gradient and $\boldsymbol{S}_f$ the surface vector [Gre15]. Compare also Fig. 11.1. Note that this is still exact.

The face gradient discretisation is implicit when the length vector $\boldsymbol{d}_{CF}$ between the centre of the cell of interest C and the centre of a neighbouring cell F is orthogonal to the face plane, i.e. parallel to $\boldsymbol{S}_f$

$$(\nabla \phi)_f \cdot \boldsymbol{S}_f = S_f \frac{\phi_F - \phi_C}{d_{CF}} \tag{11.2}$$

However, for non-orthogonal meshes, the surface vector is split into two parts, one orthogonal and one non-orthogonal part, i.e. $\boldsymbol{S}_f = \boldsymbol{E}_f + \boldsymbol{T}_f$, where only the orthogonal part is implemented implicitly.

$$(\nabla \phi)_f \cdot \boldsymbol{S}_f = \underbrace{(\nabla \phi)_f \cdot \boldsymbol{E}_f}_{\text{orthogonal like contribution}} + \underbrace{(\nabla \phi)_f \cdot \boldsymbol{T}_f}_{\text{non-orthogonal like contribution}} \tag{11.3}$$

The second term on the RHS of Eq. (11.3) is called cross-diffusion or non-orthogonal diffusion and is due to the non-orthogonality of the mesh [MMD15].

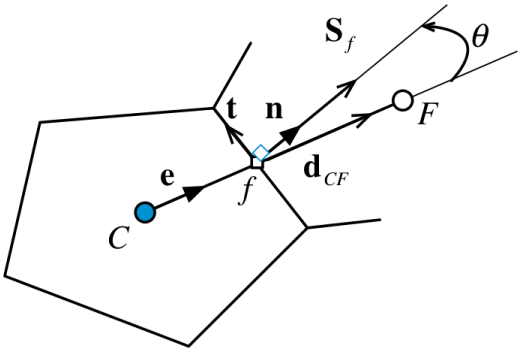


Figure 11.1: A cell in a non-orthogonal mesh system [MMD15]

In order to set up the exact Jacobi matrix, both parts should be implemented implicitly. A very good explanation how the Laplace operator is implemented in OpenFOAM can be found in [MMD15]. For this task, one needs to derive an implicit scheme for the non-orthogonal part and implement it in OpenFOAM.

Implementing the coupling term of the RANS equation

In chapter 3 the coupled system of equations to solve for the adjoint variables $\boldsymbol{\psi}_{\tilde{\nu}}$ and $\boldsymbol{\psi}_{\mathrm{u}}$ was derived. In an segregated solution approach, one needs to compute the Jacobi matrices. One Jacobi matrix is the term $\partial \boldsymbol{R}_{\mathrm{u}}/\partial \tilde{\boldsymbol{\nu}}$, i.e. the derivative of the RANS equation w.r.t. the Spalart-Allmaras variable. Recall the RANS equation from chapter 2

$$\frac{\partial U_i}{\partial x_i} = 0 \tag{11.4}$$

$$U_j \frac{\partial U_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial P}{\partial x_i} + \frac{\partial}{\partial x_j}\left(2\left(\nu + \nu_{\mathrm{t}}\right) S_{ij}\right) \tag{11.5}$$

The only term taking into account is the second one on the RHS of Eq. (11.5). Differentiating w.r.t $\tilde{\nu}$ results in

$$\frac{\partial}{\partial \tilde{\nu}}\frac{\partial}{\partial x_j}\left(2\left(\nu + \nu_{\mathrm{t}}\right) S_{ij}\right) = \frac{\partial}{\partial x_j}\left(2\frac{\partial \nu_{\mathrm{t}}}{\partial \tilde{\nu}} S_{ij}\right) \tag{11.6}$$

with $\nu_{\mathrm{t}} = f_{\mathrm{v1}}(\chi(\tilde{\nu}))\tilde{\nu}$ it follows

$$\frac{\partial \nu_{\mathrm{t}}}{\partial \tilde{\nu}} = \frac{\partial f_{\mathrm{v1}}}{\partial \chi}\frac{\partial \chi}{\partial \tilde{\nu}}\tilde{\nu} + f_{\mathrm{v1}} = \frac{3c_{\mathrm{v1}}^3\chi^3}{\left(c_{\mathrm{v1}}^3 + \chi^3\right)^2} + f_{\mathrm{v1}} \tag{11.7}$$

This term has already been implemented and verified. The implementation can be found in the file `couplingTerm.H` within the solver folder `adjointSpalartAllmaras`.

The corresponding objective function of the coupling term is

$$f_{\mathrm{coupling}} = \boldsymbol{\psi}_{\mathrm{u}}^{\mathrm{T}} \boldsymbol{R}_{\mathrm{u}} \tag{11.8}$$

So far, the residual have been computed in explicit way, i.e. using `fvc` namespace, however, the results for the adjoint method did not match with the total finite difference approach. The task is to implement the residuals in implicit form.

Implementing physical meaningful objective functions (force, drag, lift, etc.)

The objective function implemented so far is

$$f_{\mathrm{artificial}} = \sum_{i \in I} \tilde{\nu}_i \tag{11.9}$$

where $I \subset \mathbb{N}$ is a set of numbers containing the corresponding cell numbers that are set to negative one. This objective was sufficient in order to verify the discrete adjoint solver. However, for further examples one should consider physical meaningful objective functions, e.g. minimizing the drag or maximizing the lift etc.

Implementing different turbulence models ($k-\epsilon$ model, $k-\omega$ model, etc.)

This thesis focused on the implementation of the discrete adjoint solver for the "turbulent" Spalart-Allmaras turbulence model. This model is suited for external aerodynamic flow. Models of interest could also be the $k-\epsilon$ or $k-\omega$ model. Theses models are the most used ones in industry.

Implementing the solver in an object oriented way (OOP)

Further development of the code should be made by implementing the solver in considering object oriented implementation.

List of Figures

List of Tables

Bibliography

[AJS12] S. Allmaras, F. Johnson, and P. Spalart. Modifications and Clarifications for the Implementation of the Spalart-Allmaras Turbulence Model. *Seventh International Conference on Computational Fluid Dynamics (ICCFD7)*, pages 1–11, 2012.

[Arc11] N. B. Arciniega. Implementation and validation of the Spalart-Allmaras Turbulence Model. *ResearchGate*, pages 1–14, 2011.

[BOCPZ12] A. Bueno-Orovio, C. Castro, F. Palacios, and E. Zuazua. Continuous adjoint approach for the Spalart-Allmaras model in aerodynamic optimization. *AIAA Journal 50(3):631-46*, 2012.

[Gre15] C. J. Greenshields. OpenFOAM Programmer's Guide. 2015.

[MMD15] F. Moukalled, L. Mangani, and M. Darwish. *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab.* Springer Publishing Company, Incorporated, 1st edition, 2015.

[Oth08] C. Othmer. A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows. *International Journal for Numerical Methods in Fluids*, 58(8):861–877, 2008.

[SA92] P. Spalart and S. Allmaras. A one-equation turbulence model for aerodynamic flows. In *AIAA Paper, 92-0439*, 1992.

[TJ08] D. Thévenin and G. Janiga. *Optimization and Computational Fluid Dynamics.* Springer-Verlag Berlin Heidelberg, 2008.

[Wil06] D. C. Wilcox. *Turbulence Modeling for CFD (Third Edition).* DCW Industries, Inc., 2006.

[ZPGO09] A. S. Zymaris, D. I. Papadimitriou, K. C. Giannakoglou, and C. Othmer. Continuous adjoint approach to the Spalart-Allmaras turbulence model for incompressible flows. *Computers and Fluids*, 38(8):1528–1538, 2009.

Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

München, 30. September 2016

Dennis Kasper