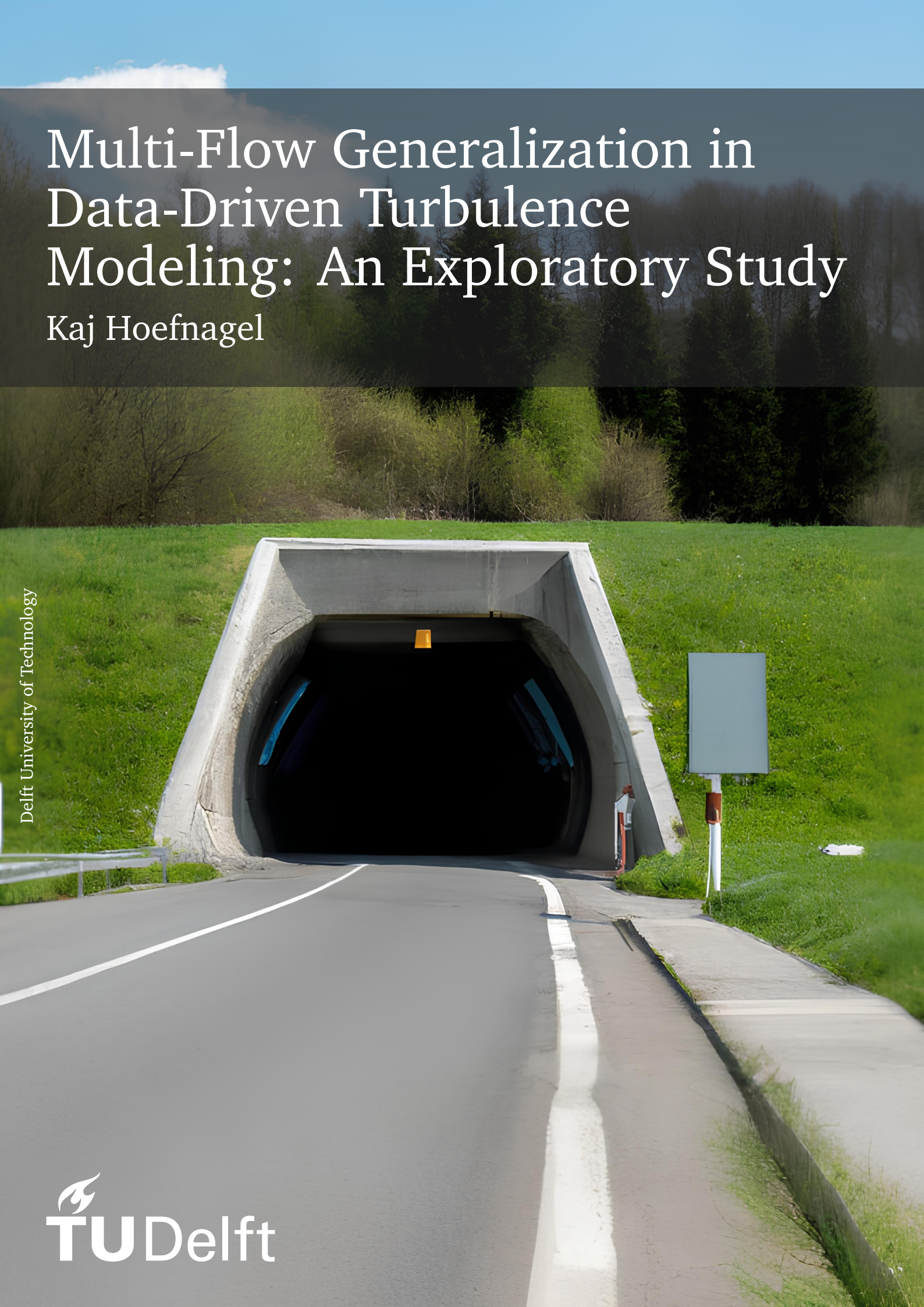
Multi-Flow Generalization in Data-Driven Turbulence Modeling: An Exploratory Study

Kaj Hoefnagel



Delft University of Technology

TU Delft

Multi-Flow Generalization in Data-Driven Turbulence Modeling: An Exploratory Study

by

Kaj Hoefnagel

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on July 20, 2023 at 13:30.

Student number:	4651715
Project Duration:	April, 2022 - July, 2023
Thesis committee:	Dr. Richard Dwight, TU Delft, Committee chair and thesis supervisor
	Dr. Steven Hulshoff, TU Delft, Additional member
	Dr. Frits de Prenter, TU Delft, external member
Faculty:	Faculty of Aerospace Engineering, Delft

TU Delft

Abstract

Computational fluid dynamics (CFD) is an important tool in design involving fluid flow. Scale-resolving CFD methods exist, but they are too computationally expensive for practical design. Instead, the relatively cheap Reynolds-averaged Navier-Stokes (RANS) approach is the industry standard, specifically models based on the Boussinesq hypothesis, which are unable to represent the effect of turbulence anisotropy. Development of RANS models based purely on physical arguments has stagnated; however, data-driven turbulence modeling presents a paradigm shift for improved predictions. Though this technique has produced successful models tailored to specific flows, it has yet to produce a successful general turbulence model, which is the focus of this work.

In this work, models consist of corrections to the classical k-ω SST turbulence model; b_{ij}^{Δ} to correct the Reynolds stress anisotropy and R to correct the turbulent kinetic energy. High fidelity data combined with the k-corrective-frozen technique is used to obtain exact correction fields, which are validated. Then, the SpaRTA framework is used to regress symbolic expressions for the corrections. Using a newly developed solver, models are injected into a full RANS solver to assess *a-posteriori* performance for various test cases. SpaRTA identifies a good R model, but only after *a-posteriori* optimization of coefficients, this model holds promise for generalization. Meanwhile, SpaRTA is unable to find a good b_{ij}^{Δ} model due to its reliance on linear regression. A new framework based on non-linear regression is introduced which identifies a much better b_{ij}^{Δ} model, though this model is Reynolds number dependent.

Preface

This thesis represents the final stage in my pursuit of a master's degree in aerodynamics at Delft University of Technology. I have worked on this thesis from April 2022 till June 2023. Though this exceeds the nominal time of nine months, I believe that I effectively used the extra time. Compared to my findings at the ∼8 month mark, the current work presents a much more well-rounded and clear contribution to the field of data-driven turbulence modeling.

My reason for choosing the topic of data-driven turbulence modeling is two-fold: Firstly, I wanted to learn more about computational fluid dynamics, both the theoretical as well as the practical side. Secondly, I wanted to learn more about the recent advancements in machine learning which have revolutionized other fields. For example, both the title and the cover image were generated by AI (chatGPT[1] and DALL-E[2] respectively). I have learned a great deal about both topics, though I have also learned that my research only represents a fraction of what there is to know.

First and foremost, I want to express my gratitude to my supervisor Dr. Richard Dwight for giving me the opportunity to work on this topic. Also for regularly discussing my progress, making sense of results and suggesting related work. Your ability to quickly understand my intricate ideas and contextualize them makes you an excellent supervisor. Next, at the basis of my thesis is the entry to the challenge posed by NASA as part of their 2022 symposium on turbulence modeling. This was a collaborative effort led by Richard, together with Renzhi and Tyler. I would like to thank all three of you for the great work under significant time pressure, which in the end allowed us to present a solid submission. Next, I want to thank Richard, Tyler, Matthijs, Renzhi and Kherlen for our productive group meetings, which inspired many ideas presented in this work. Furthermore, I am deeply grateful to Ali Amarloo and Dr. Mahdi Abkar for kindly sharing their heterogeneous roughness data.

I want to thank my friends Luuk, Coen, Yannick, Andy and Jan-Willem whom I met in the Bachelor for bringing a sense of fun and camaraderie to the lectures and the occasional evening gatherings. I further want to thank Luuk, with whom I followed the same minor and master, for the seamless collaboration on numerous projects and his valuable insights throughout. Next, I want to thank Luuk, Matthijs, Tom and Siddharth as well as the PhD students for the enjoyable lunches and coffee breaks, which served as a refreshing respite from our work. I further want to thank my (ex-)housemates for the fun dinners and occasional communal unwindings, allowing me to relax and start each day with renewed energy. Moreover, I want to thank my friends in Hilversum for providing similarly refreshing weekends filled with new experiences. Last but not least, I want to thank my family for their unwavering love, guidance, and support throughout my educational journey, including the completion of this master's thesis.

Kaj Hoefnagel
Delft, July 2023

[1] OpenAI (2023). *ChatGPT* (May 24 version) [Large language model]. https://chat.openai.com/
[2] OpenAI (2023). *DALL-E* (version 2) [Text-to-image AI] https://labs.openai.com/

Contents

List of symbols

Greek alphabet

Symbol	Definition	Unit
α	Angle of attack	deg
β	Cell growth ratio	-
γ	Heat capacity ratio	-
δ_i	ith cell height	-
δ_{ij}	Kronecker delta	-
ϵ	Context 1: Dissipation rate	$\mathrm{m^2\,s^{-3}}$
	Context 2: Classifier division threshold	-
ϵ_{ijk}	Levi-Civita symbol	-
η	Kolmogorov length scale	m
θ	Context 1: Input flow features	-
	Context 2: Momentum thickness	m
λ	Elastic net regularization weight	-
μ	Dynamic viscosity	$\mathrm{kg\,m^{-1}\,s^{-1}}$
ν	Kinematic viscosity	$\mathrm{m^2\,s^{-1}}$
ν_t	Kinematic eddy viscosity	$\mathrm{m^2\,s^{-1}}$
ξ	Classifier threshold	-
ρ	Context 1: Density	$\mathrm{kg\,m^{-3}}$
	Context 2: Elastic net mixing parameter	-
σ	Context 1: Classifier function	-
	Context 2: Data uncertainty	-
τ	Characteristic time scale	s
τ_η	Kolmogorov time scale	s
τ_i	Shear stress vector	$\mathrm{kg\,m^{-1}\,s^{-2}}$
τ_{ij} (or τ''_{ij})	Reynolds stress tensor	$\mathrm{m^2\,s^{-2}}$
ϕ	Hump wall panel rotation angle	deg
ω	Specific dissipation rate	$\mathrm{s^{-1}}$
ω_{ij}	Normalized rotation rate tensor	-

Latin alphabet

Symbol	Definition	Unit
AR	Aspect ratio	-
b_{ij}^{Δ}	Normalized Reynolds stress anisotropy correction	-
C	Unknown coefficient to be regressed	-
C_d	Drag coefficient (2D)	-
C_f	Skin friction coefficient	-
C_l	Lift coefficient (2D)	-
C_p	Pressure coefficient	-
c	Chord length	m
D_{jet} (or D_j)	Axisymmetric jet exit diameter	m
F_1, F_2	Blending functions in k-ω SST	-
$f_\lambda(q)$ (or G^λ)	λth scalar function (multiplied by T_{ij}^{λ})	-
G_λ	Production of turbulent kinetic energy modification by $T_{ij}^{(\lambda)}$	$\mathrm{m^2\,s^{-3}}$
h	Height	m
I	Turbulence intensity	-
I_m	Context 1: mth Pope invariant	-
	Context 2: mth tensor invariant	-
k	Turbulent kinetic energy	$\mathrm{m^2\,s^{-2}}$
L	Characteristic length	m
l_{mix}	Mixing length	m
M	Mach number	-
$M_{b^\Delta}^{(i)}$	ith Model for b_{ij}^{Δ}	-
$M_R^{(i)}$	ith Model for R	$\mathrm{m^2\,s^{-3}}$
$N_\cdot$	Number of $\cdot$	-
P_k	Production of turbulent kinetic energy	$\mathrm{m^2\,s^{-3}}$
$P_k^{\lvert\Delta\rvert}$	Absolute modification of P_k due to R and b_{ij}^{Δ}	$\mathrm{m^2\,s^{-3}}$
p	Pressure	$\mathrm{kg\,m^{-1}\,s^{-2}}$
p_t	Total pressure	$\mathrm{kg\,m^{-1}\,s^{-2}}$
Q	2D rotation matrix	-
q	Arbitrary feature, see Tab. 30	-
R	Context 1: Production of turbulent kinetic energy correction	$\mathrm{m^2\,s^{-3}}$
	Context 2: Specific gas constant	$\mathrm{m^2\,s^{-2}\,K^{-1}}$
R^2	Coefficient of determination	-
$Re_\cdot$	Reynolds number based on $\cdot$	-

Continued on next page

Latin alphabet continued

Symbol	Definition	Unit
r	Radius; distance from duct center	m
s_{ij}	Normalized strain rate tensor	-
T	Temperature	K
$T_{ij}^{(\lambda)}$	λth Pope base tensor	-
t	Time	s
U	Characteristic velocity	$\mathrm{m\,s^{-1}}$
$U_\cdot$	Velocity in $\cdot$-direction	$\mathrm{m\,s^{-1}}$
U_0	Mean streamwise velocity averaged at $z = h$ (roughness)	$\mathrm{m\,s^{-1}}$
u_b	Bulk velocity	$\mathrm{m\,s^{-1}}$
u_i	Velocity vector	$\mathrm{m\,s^{-1}}$
u_τ	Friction velocity	$\mathrm{m\,s^{-1}}$
w	Roughness strip width	m
$\hat{w}_{\|\|,i}$	Wall tangential unit vector	-
x_i	Cartesian coordinate vector	m
y	Distance to closest wall	m
y_1^+	Dimensionless first cell height	-
$y_{1/2}^+$	Dimensionless first cell center height	-

Other notations

Notation	Definition
$\bar{\cdot}$	Filtered quantity
$\cdot''$	Context 1: Sub-filtered quantity (LES)
	Context 2: Dispersive velocity (see Eq. 74)
$\langle\cdot\rangle$	Averaged quantity
$\cdot'$	Fluctuating quantity
$\cdot_\infty$	Freestream value of $\cdot$
$\cdot_{ref}$	Reference value of $\cdot$
$\cdot_w$	Wall value of $\cdot$
$\cdot^+$	Nondimensionalized version of $\cdot$ (usually by wall quantities)

List of abbreviations

ASM	Algebraic stress model
CFD	Computational fluid dynamics
CuRTA	Curve-fit regression of turbulent stress anisotropy
DES	Detached eddy simulation
DNS	Direct numerical simulation
EASM	Explicit algebraic stress model
FLOP	Floating point operation
GEP	Gene expression programming
LES	Large eddy simulation
LEVM	Linear eddy viscosity model
LRR	Launder-Reece-Rodi
NaN	Not a number
NASA	National Aeronautics and Space Administration
NLEVM	Nonlinear eddy viscosity model
PDE	Partial differential equation
PIV	Particle image velocimetry
RANS	Reynolds-averaged Navier-Stokes
RMS	Root mean square
RSM	Reynolds stress model
RST	Reynolds stress tensor
SpaRTA	Sparse regression of turbulent stress anisotropy
SSG	Speziale-Sarkar-Gatski
SST	Shear stress transport

1 Introduction

Computational fluid dynamics (CFD) is used in almost all modern design involving fluid flow, ranging from internal combustion engines to full aircraft [5, p. 6-9]. Most CFD methods are based on the Navier-Stokes equations; a set of partial differential equations (PDEs) derived in the first half of the 19th century [63, p. 3]. To this day, no analytical solution has been found for these equations, so CFD relies entirely on modeling them. One approach known as direct numerical simulation (DNS) numerically solves the equations by resolving all relevant scales. Unfortunately, this approach is unfeasible for most practical engineering problems with current computing power. Another approach called large-eddy simulation (LES) resolves only the largest scales and models the smaller ones, leading to a significantly reduced computational cost compared to DNS. Although for a long time the computational cost of LES was also too high for practical engineering problems, it has been gaining popularity in recent years [69, p. 112].

The most popular approach to CFD is solving the Reynolds-averaged Navier-Stokes (RANS) equations. The main idea is to split the flow into a mean- and fluctuating component and then take the average [40]. The resulting system of PDEs contains mostly mean terms, except for the Reynolds stress tensor (RST), which contains the averaged products of fluctuating velocity components [5, p. 321]. The RST requires modeling as it depends on fluctuating components, while only mean components are solved for, it is the main focus of RANS modeling. RANS is much cheaper than LES as it does not discretize time, can be less than 3-dimensional and only needs to accurately resolve mean flow gradients [5, p. 325]. The most popular RANS models rely on the Bousinesq hypothesis to model the RST, which is unable to represent turbulence anisotropy. While alternative models have been developed, these have not proven consistently better than Boussinesq models for general cases, meaning classical RANS modeling has been stagnant over the past thirty years.

Recent developments in machine learning combined with the increasing availability of high fidelity data has opened the possibility of using this data to train more accurate RANS models [12]. This has led to the discipline known as data-driven turbulence modeling. A variety of approaches exist within it, from simple coefficient recalibration to symbolic regression of models beyond Boussinesq's hypothesis. Most models show improvements in an *a-priori* setting; prediction of the RST based on fixed field data (such as from an existing RANS model). The *a-posteriori* setting, in which the velocity and pressure are updated, yielding new predictions of the RST, seems much harder. Many models suffer from stability issues in the *a-posteriori* environment and improvements are less obvious than *a-priori* [23] [46].

The k-corrective-frozen approach and the SpaRTA framework introduced by Schmelzer et al. are of particular focus in the current work, which is written under the same group [46]. They still use a Boussinesq dependent model (k-ω SST), but they introduce corrections to the RST and the production of turbulent kinetic energy. The five main steps of the methodology are summarized below:

1. Establish a baseline case, including a mesh independent mesh, boundary conditions, initial conditions and a converged result.
2. Interpolate the high-fidelity field data onto the RANS mesh and run k-corrective-frozen to obtain correction fields.
3. Validate the correction fields *a-posteriori* (propagation).

4. Train models based on the correction fields using SpaRTA for symbolic regression.
5. Propagate the trained models through a full RANS solver to attain *a-posteriori* results.

With this approach, Schmelzer et al. were able to find a simple symbolic expression model that improved *a-posteriori* predictions of various separating flows, also at a higher Reynolds number. Their approach has been applied to a variety of flows, yielding simple models which improve *a-posteriori* predictions, even for non-training cases of similar topology [21] [67]. However, there is little precedent for training and testing a model on a range of flow topologies, not for the aforementioned approach but also not for the field as a whole. Such a model would prove much more useful to the wider CFD community, as they can more confidently apply it to an unseen, complex case. This is the direct impetus for the collaborative testing challenge as part of NASA's 2022 symposium on turbulence modeling [45]. The goal of the challenge is to train a single RANS turbulence model and test it on five distinct flows. The classifier approach by Steiner et al., which applies corrections only in certain areas, shows promise for generating such a general model [54].

The novelty of this works is in part formed by an entry to the challenge, for which a model and a classifier are trained using the approach by Steiner et al. Unsatisfactory model fits for the RST correction of the challenge cases inspired further efforts in this work. Two cases are added which are dominated by anisotropy, a phenomenon that heavily relies on the RST correction. The functional forms that SpaRTA is able to regress are found to be too limited, especially for RST correction models. Hence, a new symbolic regression framework is proposed which is able to regress a wider range of functional forms. Furthermore, a new solver is developed that automates the propagation of models (step five of the approach outlined above).

The aim of the current study is summarized by the following research question, which is further divided into sub-questions:

How suitable is the *k*-corrective-frozen approach combined with the SpaRTA framework [46] for training a model giving improvements over a range of steady-state flows with respect to k-ω SST?

- Is the SpaRTA framework [46] suitable for the symbolic regression of general models?
- Does a classifier which applies corrections only in certain areas improve generalizability of models?
- How well does the *a-priori* fit to the corrections translate to *a-posteriori* performance of the model?

The thesis is structured as follows: In Sec. 2, classical approaches to CFD are reviewed with an emphasis on RANS models. Then in Sec. 3, relevant literature on data-driven turbulence modeling is discussed, with particular focus on work related to *k*-corrective-frozen and SpaRTA. Sec. 4 delves into the functionality and implementation of the new symbolic regression framework and the solver to automate model propagation. Next, efforts for the NASA challenge entry are summarized in Sec. 5. Then, steps 1-3 of the methodology laid out above are described for five flows. The first two are the anisotropic flows; a rectangular duct and heterogeneous roughness, described in Sec. 6 and 7 respectively. The latter three come from the challenge; a channel, a flat plate and a wall-mounted hump, described in Sec. 8, 9 and 10 respectively. The classifier is further studied in Sec. 11, followed by results of model training and testing in Sec. 12. Finally, conclusions are drawn in Sec. 13 after which recommendations are given for future work in Sec. 14.

2 Classical computational fluid dynamics

When designing a new CFD model, it is important to understand how existing ones were designed. Traditionally, CFD models were designed by making physically motivated assumptions to simplify the Navier-Stokes equations. Models that rely on more simplifications usually have a lower accuracy compared to models with few simplifications. However, models with more simplifications are usually significantly cheaper in terms of computational cost. When designing a new model, the goal is usually to have a cost similar to models of the same class, while improving accuracy. Inaccuracies in CFD models originate from the assumptions made in their derivation, understanding these assumptions is therefore key in improving model accuracy. In this light, the current section aims at exploring existing CFD models and their underlying assumptions.

In Sec. 2.1, the assumptions behind the Navier-Stokes equations are laid out in various sets, each leading to a slightly simpler version of the Navier-Stokes equations. Then, in Sec. 2.2, the Reynolds number is explained along with it implications on the cost of CFD. Also, the most accurate but expensive form of CFD modeling (DNS) is explained. Next, the evolution of turbulent structures is explored in Sec. 2.3 along with a class of CFD models of medium cost and accuracy (LES). The idea behind RANS, the cheapest yet least accurate class of CFD models, is laid out in Sec. 2.4. Since this class of models is the focus of the current work, various RANS models based on Boussinesq's hypothesis are given in Sec. 2.5 along with their underlying assumptions. Finally, in Sec. 2.6, other RANS models are laid out that are not based on Boussinesq's hypothesis.

2.1 Navier-Stokes equations

At the foundation of fluid dynamical theory are the Navier-Stokes equations; a set of partial differential equations (PDEs) describing the velocity and thermodynamic state of fluids through space and time. Though the derivation of these equations is not shown here, the main assumptions and ideas behind the derivation are. A comprehensive derivation of the Navier-Stokes equations can be found in Chapter 2 of Anderson's Fundamentals of Aerodynamics book [4]. The fundamental assumptions used in the derivation of the Navier-Stokes equations are listed below.

Fundamental assumptions in Navier-Stokes:

- Conservation of mass
- Conservation of momentum
- Conservation of energy
- Continuum assumption

The derivation of the Navier-Stokes equations starts by assuming conservation of mass, momentum and energy on an infinitesimal element. The infinitesimal element is assumed to act as a continuum rather than a collection of separate molecules (continuum assumption). As explained by Pope, for a flow at atmospheric conditions, over a non-microscopic object at typical atmospheric velocities, this assumption is valid [37, p. 10]. With these assumptions, a mass, momentum and

energy balance of the infinitesimal fluid element can be made, the structure of which is as follows:

$$\oiint_{S=\text{Surface of element}} (\text{mass/momentum/energy flux})\, dS = \iiint_{\mathcal{V}=\text{Volume of element}} (\text{internal mass/momentum/energy increase})\, d\mathcal{V}. \quad (1)$$

The surface integrals can be converted to volume integrals using the divergence theorem. Then, the integrals can be dropped altogether since no assumption was made on the shape of the infinitesimal element, resulting in the Navier-Stokes equations in differential form (not shown here as this form is simplified further). Whereas mass and energy are scalar quantities, momentum is a vector quantity, meaning the Navier-Stokes equations consist of five PDEs. Since there are five independent unknowns (x-, y-, z-velocity, pressure and temperature), the system is closed.
In the majority of aerodynamic analyses, additional assumptions are used to simplify the Navier-Stokes equations. These assumptions are listed below and they are motivated next.

Further assumptions used in the majority of aerodynamic analyses:

- No chemical reactions
- Uniform composition
- No electromagnetic forces
- Viscous stress linear function of strain rate (Newtonian fluid)
- Fluid is isotropic

Firstly, it is assumed that there are no chemical reactions, such that the composition of the fluid is not changing and there is no source of energy. Secondly, the composition of the fluid is assumed to be uniform, such that there is no diffusion. Thirdly, electromagnetic forces are assumed to be absent to simplify the body force term. For dry air (typical in aerodynamic analyses), these are valid assumptions if the Mach number is not too high [63, p. 60].

Next, one of the terms in the momentum equations is the viscous stress. In order to obtain a solution, this viscous stress should be expressed in terms of the five unknowns (x-, y-, z-velocity, pressure and temperature). For so-called Newtonian fluids, the viscous stress depends linearly on the strain rate (which depends only on the velocity gradient). Linearity implies that the two can be related through a rank-4 tensor, which is symmetric since the viscous and strain rate tensor are symmetric. Symmetry of this rank-4 tensor implies it has 36 independent coefficients, where each coefficient is a thermodynamic property of the fluid which can be expressed in terms of pressure and temperature. In order to reduce the number of independent coefficients, the fluid is also assumed to be isotropic, leaving just two independent coefficients; the dynamic viscosity and the bulk viscosity. Both the Newtonian fluid and the isotropic fluid assumption are valid for dry air [63, p. 65].

The version of the Navier-Stokes equations resulting from the assumptions listed above is often referred to as the compressible Navier-Stokes equations within the field of aerodynamics. In the current work, additional assumptions are made to arrive at the even simpler incompressible Navier-Stokes equations. These assumptions are listed below and explained next. It should be noted that there are many practical aerodynamic scenarios where these next assumptions are invalid.

Further assumptions in the current work:

- Incompressible fluid
- Constant viscosity

The fluid is assumed to be incompressible which implies that the volume of fluid elements does not change as they travel through the flow. Note that this assumption is only valid for air up to Mach 0.3 [4, p. 64]. For many aerodynamic applications (such as commercial jets), this is not a valid assumption as Mach numbers are much higher. Combining incompressibility with the uniform composition assumption made earlier implies that the density of the fluid is constant. This means that the density can be taken out of many derivatives, as it is no longer a function of time and space. Furthermore, the bulk viscosity reduces to zero under this assumption, leaving the dynamic viscosity μ as the only independent coefficient relating the viscous stress to the strain rate [63, p. 68].

The dynamic viscosity is a thermodynamic property that can be found from the pressure and temperature, for instance using Sutherland's law (which only uses temperature). This law also implies that a relatively constant temperature results in a constant viscosity. For aerodynamic problems below Mach 0.3 and without heat sources, constant temperature is a valid assumption and it is made in the current work. This actually leaves no more temperature dependent terms in the mass and momentum equations, meaning they decouple from the energy equation. As noted by Pope, assuming both a constant density and temperature means that the pressure is no longer a thermodynamic variable. Now, only relative differences in the pressure field influence the velocity field, meaning the effect of gravity can be removed by including it in the modified pressure [37, p. 18].

The simplified Navier-Stokes equations resulting from the assumptions listed above are given in Einstein notation in Eq. 2; these will be the foundation of the models discussed in the current work. Note that the ν parameter in the momentum equation is the kinematic viscosity; it is simply the dynamic viscosity divided by the density ($\nu = \mu/\rho$). Finally, note that no general analytical solution exists for these equations. Instead, approximate solutions are found using computers. In the next sections, various methods for obtaining these approximate solutions are discussed.

Incompressible, constant-viscosity Navier-Stokes equations

$$\begin{aligned} &\textit{Conservation of mass} && \frac{\partial u_i}{\partial x_i} = 0 \\ &\textit{Conservation of momentum} && \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} \end{aligned} \tag{2}$$

2.2 The Reynolds number and direct numerical simulation

Before discussing numerical solution methods to the Navier-Stokes equations, the distinction between laminar and turbulent flow shall be laid out. Laminar flow is characterized as organized and predictable, with perpendicular streamlines (no mixing). Turbulent flow, on the other hand, is highly chaotic and unpredictable and much more unsteady than laminar flow. When a flow goes from turbulent to laminar, this is known as relaminarization. When a flow goes from laminar to turbulent, this is known as transition. An example of candle fumes transitioning from laminar to turbulent is shown in Fig. 1.

Figure 1: Schlieren image of candle fumes transitioning from laminar to turbulent, by Gary Settles [48].

The main parameter governing whether a flow is laminar or turbulent is the Reynolds number, defined as:

$$Re = \frac{UL}{\nu}. \tag{3}$$

The Reynolds number is a dimensionless measure of the ratio between inertial and viscous forces. The inertial forces are characterized by a velocity U and a length scale L. The viscous forces are characterized by the kinematic viscosity ν. Low Reynolds numbers are associated with laminar flow, while high Reynolds numbers are associated with turbulent flow. In Fig. 1, the characteristic length scale is the height above the candle; at a certain height the critical Reynolds number is reached and the flow transitions to turbulent.

Another effect of the Reynolds number is visible in Fig. 1 after the point of transition; first the turbulent flow structures are chaotic, but no small structures are visible yet. As the Reynolds number increases further, large structures are still visible, but increasingly smaller structures also appear. Thus, turbulent flow is characterized by structures with a range of sizes, this size range increases with the Reynolds number. The ratio between the length scale of the largest and smallest turbulent structure is given as:

$$\frac{L}{\eta} \approx Re^{3/4}. \tag{4}$$

These smaller structures also live on much smaller timescales than the largest structures, further giving a range of timescales. The ratio between the timescale of the largest and smallest turbulent structure is given as:

$$\frac{\tau}{\tau_\eta} \approx Re^{3/4}. \tag{5}$$

Both ratios are derived based on the equations given in Sec. 9.1.2 in the book by Pope [37].

The concepts of the Reynolds number and the range of turbulent scales are introduced here, as they are highly relevant to the most obvious approach to numerically solving the Navier-Stokes equations. Namely, local approximations of first and second derivatives can be found using Taylor series expansions. Thus, by discretizing the problem in both space and time, a solvable system of equations is obtained (see the book by Van Kan et al. for more information [57]). In order for the discretized solution to be close to the continuous solution, all relevant scales should be resolved. Though the smallest structures in Fig. 1 may seem irrelevant, they are responsible for the majority of the dissipation, meaning they also have to be resolved. This approach of simply solving the discretized Navier-Stokes equations on a grid fine enough to resolve all relevant scales is known as direct numerical simulation (DNS).

Contrary to what one may initially believe, DNS is not used for most practical engineering problems involving turbulent flow. To understand why, consider again the ratio between the largest and smallest length and time scales in Eq. 4 and Eq. 5 respectively. Since turbulence is chaotic, it is inherently 3-dimensional. Even for the simplest geometries, a 3-dimensional computational domain should be used which represents the whole range of turbulent structures. Furthermore, since turbulence is unsteady, time integration has to be used. The total time should be larger than the time scale of the largest structure, while the timestep should be smaller than the timescale of the smallest structure. Representing all scales in three spatial dimensions and one temporal dimension gives a combined scaling with Re^3.

Pope provides an estimate for the total number of domain points times the total number of timesteps in terms of the Reynolds number in Eq. 9.12 [37, p. 348]: $160Re^3$. Assuming an extremely efficient matrix solver which only needs one floating point operation (FLOP) per point per timestep, $160Re^3$ FLOPs would be required to solve one flow case. Consider a group of aerodynamicists simulating a 4.6 m long Toyota Prius [3] driving on the highway at 120 km h^{-1} (33 m s^{-1}), corresponding to a Reynolds number of 6.5 million. A DNS simulation of this case would require 4.4×10^{22} FLOPs. The new DelftBlue supercomputer with a theoretical maximum of 1.05×10^{15} floating point operations per second [4] would take at least 1.3 years to complete this simulation.

Even this extremely conservative runtime estimate of the Toyota Prius illustrates the unfeasibility of using DNS for practical engineering problems. Most engineers do not have access to a computer the size of DelftBlue. Furthermore, many simulations have to be performed in the design process with a runtime of at most a few weeks. Additionally, many aerodynamic applications have an even higher Reynolds number than the Toyota Prius. In practise, DNS is only used to study low-Reynolds number academic cases, where the number of runs is limited and a runtime of months is acceptable. Luckily, there are alternative methods for finding an approximate solution to the Navier-Stokes equations with a lower computational cost than DNS, these are discussed in the next sections.

[3] Toyota (2019). *2023 PRIUS Full Specs*. Toyota. https://www.toyota.com/prius/features/mpg_other_price/1223/1225/1227

[4] Delft High Performance Computing Centre (2022). *DelftBlue: the TU Delft supercomputer*. TU Delft. https://www.tudelft.nl/en/dhpc/system

2.3 The energy cascade and large eddy simulation

Pope lays out Richardson's view on the origin of the range of sizes in turbulent structures (eddies), which was discussed in the previous section. The largest eddies have a Reynolds number comparable to the case Reynolds number. Due to vortex stretching, these eddies break up into smaller and smaller eddies. The size and velocity of these smaller eddies decreases, meaning their Reynolds number also decreases. As mentioned, the Reynolds number is the ratio between inertial and viscous forces, so the lower Reynolds number of these smaller eddies means a larger effect of dissipation. At the smallest scales, the Reynolds number becomes so low that no further break up occurs. Instead, these smallest eddies rapidly disappear due to viscous dissipation [37, p. 183].

The process described above is known as the energy cascade, it can be summarized as the transfer of energy from the largest to the smallest scales, where it is dissipated. The largest eddies are responsible for the production of turbulent kinetic energy and contain the vast majority of this energy. The nature of these largest eddies is also highly dependent on the boundary conditions. Kolmogorov argues that as eddies break down, the turbulence becomes more and more isotropic (also implying universal), since the chaotic nature of turbulence reduces anisotropic bias [24]. This implies that it may be possible to find a universal model for these smallest scales. As mentioned, simulation of these smallest scales is responsible for the majority of the computational cost of DNS, so modelling them could lead to a significant decrease in cost. Resolving only the large eddies and modelling the small eddies is a discipline known as large eddy simulation (LES).

In order to distinguish the large and small eddies, velocity is decomposed into a filtered component $\overline{u}_i$ and a sub-filtered component u_i'':

$$u_i = \overline{u_i} + u_i''. \tag{6}$$

A similar decomposition is done for the pressure:

$$p = \overline{p} + p''. \tag{7}$$

Applying filtering to the incompressible, constant-viscosity Navier-Stokes equations in Eq. 2 results in the filtered Navier-Stokes equations:

$$\begin{aligned} \frac{\partial \overline{u}_i}{\partial x_i} &= 0 \\ \frac{\partial \overline{u}_i}{\partial t} + \overline{u}_j \frac{\partial \overline{u}_i}{\partial x_j} &= -\frac{1}{\rho}\frac{\partial \overline{p}}{\partial x_i} + \nu \frac{\partial^2 \overline{u}_i}{\partial x_j \partial x_j} + \frac{\partial \tau_{ij}^R}{\partial x_j}. \end{aligned} \tag{8}$$

Note that in this derivation, filtering and differentiation are assumed to commute, which is only true for spatially uniform filters [37, p. 581]. The τ_{ij}^R term in Eq. 8 is known as the residual-stress tensor, it is defined as:

$$\tau_{ij}^R = \overline{u}_i \overline{u}_j - \overline{u_i u_j}. \tag{9}$$

The $\overline{u_i u_j}$ component of this tensor cannot be expressed in terms of filtered quantities, but requires knowledge of the unfiltered flow. Finding models for this tensor in terms of filtered flow quantities is the main focus of LES modelling.

Since only the largest eddies are simulated, the mesh requirements of LES are much less strict than those of DNS, as only these largest eddies have to be represented. The question remains how fine to make the mesh; refining the mesh leads to representing a larger part of the energy cascade. Usually the mesh refinement is chosen such that at least 80% of the turbulent kinetic energy is resolved [37, p. 560]. Meeting this requirement near walls results in a rather fine mesh, and the number of mesh cells actually becomes dependent on the Reynolds number again [37, p. 598]. An alternative approach is to drop the 80% requirement near walls and use wall models instead, though this introduces additional uncertainties.

Though LES is much cheaper than DNS for high-Reynolds number cases, it is often still too computationally expensive to be of use in practical aerodynamic design. The 80% requirement still necessitates rather fine grids. Furthermore, the simulated large eddies are still inherently 3-dimensional and unsteady. Thus, even for simple geometries, LES requires a 3-dimensional domain simulated through time. Hence, many aerodynamic design problems require a computationally even cheaper method than LES. The most popular one, Reynolds-averaged Navier-Stokes, is discussed in the next section.

2.4 Reynolds-averaged Navier-Stokes

Reynolds averaged Navier-Stokes (RANS) takes the ideas of LES one step further: Instead of separating the large and small eddies, the velocity is decomposed into a mean component $\langle u_i \rangle$ and a fluctuating component u_i' [40]:

$$u_i = \langle u_i \rangle + u_i'. \tag{10}$$

A similar decomposition is used for the pressure:

$$p = \langle p \rangle + p'. \tag{11}$$

Applying these decompositions to the incompressible, constant-viscosity Navier-Stokes equations in Eq. 2 and averaging results in the Reynolds-averaged Navier-Stokes equations:

$$\begin{aligned} \frac{\partial \langle u_i \rangle}{\partial x_i} &= 0 \\ \frac{\partial \langle u_i \rangle}{\partial t} + \langle u_j \rangle \frac{\partial \langle u_i \rangle}{\partial x_j} &= -\frac{1}{\rho}\frac{\partial \langle p \rangle}{\partial x_i} + \nu \frac{\partial^2 \langle u_i \rangle}{\partial x_j \partial x_j} - \frac{\partial \langle u_i' u_j' \rangle}{\partial x_j}. \end{aligned} \tag{12}$$

More details on this derivation can be found in the book of Pope [37, p. 83].

The structure of the RANS equations (Eq. 12) is similar to the structure of the filtered Navier-Stokes equations used for LES (Eq. 8). The term $\langle u_i' u_j' \rangle$ is known as the Reynolds stress tensor (RST), denoted as τ_{ij}''. Like the τ_{ij}^R term in the filtered Navier-Stokes equations, the RST is the only term that cannot be expressed in terms of mean flow quantities. Finding models for the RST in terms of mean flow quantities is the main focus of RANS modeling. For simplicity, the RST is referred to as τ_{ij} throughout this work.

Simulating only mean flow quantities and modeling all effects of turbulence comes with several advantages in terms of computational cost. Firstly, if the mean flow is constant through time, the whole dimension of time can be removed from the simulation by solving for the steady state. This simply means setting $\partial \langle u_i \rangle / \partial t$ to zero in the momentum equation. Secondly, if the geometry implies a mean flow that is constant in a certain dimension this can be exploited by RANS by only using a single cell in the direction of this dimension. Thirdly, if the geometry implies symmetries in the mean flow, only one side of the symmetry has to be modeled; the mean flow results can simply be mirrored to the other side of the symmetry. In LES and DNS, these simplifications are impossible as (part of) the turbulence is resolved, which is inherently unsteady, chaotic and 3-dimensional. Finally, a coarser mesh can be used for RANS as it only needs to accurately resolve gradients of the mean flow, rather than gradients of the (largest) eddies [5, p. 325].

Due to its low cost, RANS is the most popular approach to practical fluid dynamics problems. However, this lower cost comes at the price of reduced accuracy, as RANS models are based on rather strong assumptions [46]. The most popular class of RANS models is explored in the next section. Another note regarding the cost of RANS models: if the whole wall region is resolved, the grid spacing at the wall is Reynolds number dependent. However, similar to LES, wall models can be used if resolving the wall proves too computationally expensive.

2.5 Linear eddy viscosity modeling in RANS

The most popular class of RANS models, linear eddy viscosity models (LEVMs), rely on Boussinesq's hypothesis to model the Reynolds stress. Boussinesq's hypothesis is laid out in Sec. 2.5.1 along with its shortcomings. LEVMs are classified based on the number of additional PDEs they introduce. The first eddy viscosity models did not introduce any additional PDEs and are thus known as zero-equation models. Though not as popular today, these models are explored in Sec. 2.5.2 to motivate further models. In Sec. 2.5.3, one equation models are explored, which introduce one additional PDE. Finally, two-equation models (introducing two additional PDEs) are explored in Sec. 2.5.4. Specifically, the k-ω SST model is explored in detail as it is the most popular RANS model and the one under investigation in the current work.

2.5.1 Boussinesq's hypothesis

As explained in Sec. 2.4, the RST is the only term in the RANS equations (Eq. 12) that requires modeling. A model for the RST should be in terms of mean quantities as only these are resolved. LEVMs rely on Boussinesq's hypothesis, which models the effect of turbulence in a similar way as molecular viscosity of a Newtonian fluid [37, p. 93]. In the RANS equations in Eq. 12, the viscous term (second term of the right-hand side of the momentum equation) is written in a simplified form, which is allowed by the incompressible, constant-viscosity fluid assumption. The more general viscous term (also valid in case of a non-constant viscosity) is as follows:

$$\frac{\partial}{\partial x_j}\left[\nu\left(\frac{\partial \langle u_i \rangle}{\partial x_j} + \frac{\partial \langle u_j \rangle}{\partial x_i}\right)\right]. \tag{13}$$

Writing the Reynolds stress in an analogous way to the part of the viscous term (Eq. 13) between square brackets results in the Boussinesq hypothesis:

$$\langle u_i' u_j' \rangle = -\nu_t\left(\frac{\partial \langle u_i \rangle}{\partial x_j} + \frac{\partial \langle u_j \rangle}{\partial x_i}\right) + \frac{2}{3}k\delta_{ij}. \tag{14}$$

The $2/3k\delta_{ij}$ term is added in Eq. 14 such that the trace of the RST is equal to $2k$. This k is the turbulent kinetic energy, defined as:

$$k = \frac{1}{2}\left(\langle u_1' u_1' \rangle + \langle u_2' u_2' \rangle + \langle u_3' u_3' \rangle\right), \tag{15}$$

the trace of $2k$ is required for this definition to be consistent. Finally, rather than the kinematic molecular viscosity ν, Eq. 14 uses the kinematic eddy viscosity ν_t. Contrary to the molecular viscosity, the eddy viscosity is a function of space and time. Finding an expression for the eddy viscosity in terms of mean flow quantities is the focus of LEVMs.

Boussinesq's hypothesis assumes that the strain rate tensor and the RST are aligned, which is a rather strong assumption as shown by Schmitt [47]. Schmitt finds the misalignment angle between these two tensors for various flows based on DNS/LES data. Even for simple flows, he finds a significant misalignment between these two tensors, indicating Boussinesq's hypothesis is invalid. He notes that for shear dominated flows, the misalignment does not effect the final result. However, this does not mean that shear dominated flows always produce accurate results. Since Schmitt only measures the misalignment, differences in magnitude between these two tensors are not included in his measure. Thus, an inaccurate prediction of the magnitude can still spoil the solution of shear dominated flows. The magnitude prediction depends on the eddy viscosity which is model dependent. Various eddy viscosity models are explored in the coming sections. Finally, it should be noted that the misalignment angle is independent of eddy viscosity, meaning Schmitt's findings hold for all LEVMs.

2.5.2 Zero-equation models

The idea of momentum transfer due to turbulence being analogous to momentum transfer due to molecular viscosity, used in the Boussinesq hypothesis, was also used by Prandtl to come up with the first eddy viscosity models. Prandtl imagined lumps of particles having a certain mixing length l_{mix}, akin to the mean free path of molecules [38]. This results in the following expression for the eddy viscosity:

$$\nu_t = l_{mix}^2 \left|\left| \frac{\partial u_i}{\partial x_j} \right|\right|. \tag{16}$$

One major drawback of the mixing length model is that the mixing length is not the same for each type of flow, so it has to be chosen beforehand. Also, the mixing length is only approximately constant when it is far from walls. In boundary layers, the mixing length should decrease closer to the wall. Various near-wall corrections for the mixing length proposed through the years are laid out in the book of Wilcox [65, p. 77-79]. These corrections are implemented in later mixing length models such as the Cebeci-Smith model [49].

The mixing length models discussed above are known as zero equation models, because they introduce no extra PDEs. The advantage is that these models have a low computational cost and are relatively easy to implement. One drawback of these models was already mentioned; the need to specify the mixing length beforehand, requiring calibration. Another drawback is the lack of time history in these models, giving rise to bad performance for flows where the time history is relevant such as separation [65, Sec. 3.6]. In an attempt to improve performance for such cases, new models were proposed which included additional PDEs, these are discussed next.

2.5.3 One-equation models

The first eddy viscosity model introducing an extra PDE was proposed by Prandtl [39]. He chooses to solve a PDE for the turbulent kinetic energy k, as the square root of k gives a characteristic turbulent velocity. The PDE solved to obtain k is used in many subsequent models, so it is further examined here. First, the Reynolds stress transport equation is needed:

$$\frac{\partial \langle u_i' u_j' \rangle}{\partial t} + \langle u_k \rangle \frac{\partial \langle u_i' u_j' \rangle}{\partial x_k} = -\langle u_i' u_k' \rangle \frac{\partial \langle u_j \rangle}{\partial x_k} - \langle u_j' u_k' \rangle \frac{\partial \langle u_i \rangle}{\partial x_k} + 2\nu \left\langle \frac{\partial u_i'}{\partial x_k} \frac{\partial u_j'}{\partial x_k} \right\rangle + \dots + \left\langle \frac{u_i'}{\rho} \frac{\partial p'}{\partial x_j} \right\rangle + \left\langle \frac{u_j'}{\rho} \frac{\partial p'}{\partial x_i} \right\rangle + \nu \frac{\partial^2 \langle u_i' u_j' \rangle}{\partial x_k \partial x_k} + \frac{\partial \langle u_i' u_j' u_k' \rangle}{\partial x_k}. \quad (17)$$

This equation is obtained by manipulating the Reynolds-averaged Navier-Stokes equations in Eq. 12, the full derivation can be found in the book of Wilcox [65, p. 41]. Note that this equation is exact, however, many new unknowns are introduced, so the system is not closed (see Sec. 2.6). By taking half the trace of the Reynolds stress equation and reordering some terms, the transport equation for k is obtained:

$$\frac{\partial k}{\partial t} + \langle u_j \rangle \frac{\partial k}{\partial x_j} = -\langle u_i' u_j' \rangle \frac{\partial \langle u_i \rangle}{\partial x_j} - \epsilon + \nu \frac{\partial^2 k}{\partial x_j \partial x_j} - \frac{1}{2} \frac{\partial \langle u_i' u_i' u_j' \rangle}{\partial x_j} - \frac{1}{\rho} \frac{\partial \langle p' u_j' \rangle}{\partial x_j}, \quad (18)$$

where ϵ is defined as:

$$\epsilon = \nu \left\langle \frac{\partial u_i'}{\partial x_k} \frac{\partial u_i'}{\partial x_k} \right\rangle. \quad (19)$$

The full derivation of this equation can also be found in the book of Wilcox [65, p. 108].

The transport equation of k in Eq. 18 is again exact, however, there are more unknowns than equations. The last and second to last term are modeled together using a gradient-diffusion approximation:

$$\frac{1}{2} \langle u_i' u_i' u_j' \rangle + \frac{1}{\rho} \frac{\partial \langle p' u_j' \rangle}{\partial x_j} = -\frac{\nu_t}{\sigma_k} \frac{\partial k}{\partial x_j}, \quad (20)$$

where σ_k is a model constant. This approximation is actually not valid for the pressure term, but the introduced error is small. Prandtl used the following equation to model the dissipation rate ϵ:

$$\epsilon = C_D \frac{k^{3/2}}{l_{mix}}, \quad (21)$$

where C_D is a model constant and l_{mix} is the mixing length, necessary to obtain the correct dimension. Thus, an a priori calibration procedure is still necessary to determine an appropriate mixing length. In practical problems, there are often many different flow regimes, making the specification of an appropriate mixing length rather difficult [65, p. 110-111].

2.5.4 Two-equation models

In order to address the problem of having to define a turbulent length scale, two equation models were introduced, where a second PDE is included. This second equation does not need to solve for the turbulent length scale directly, another quantity with dimensions of length or time that is not a power of length per time works as well. The two most popular choices for this second variable are the dissipation rate ϵ (dimension length^2/time^3) and the specific dissipation rate ω (dimension 1/time). These models are known as $k-\epsilon$ and $k-\omega$ models respectively, many variations exist of both.

By manipulating the Reynolds-averaged Navier-Stokes equations, exact transport equations can be derived for ϵ and ω, as was done for k. However, these are much more complicated than the transport equation for k (Eq. 18). Finding suitable closures for the unknown terms in these exact equations and validating them with experimental data was at the time impossible. Thus, the transport equations for ϵ and ω are postulated, based on the form of the relatively simple k transport equation. The ϵ transport equation of the popular standard k-ϵ model is as follows [26]:

$$\frac{\partial \epsilon}{\partial t} + \langle u_j \rangle \frac{\partial \epsilon}{\partial x_j} = -C_{\epsilon 1} \frac{\epsilon}{k} \left\langle u_i' u_j' \right\rangle \frac{\partial \langle u_i \rangle}{x_j} - C_{\epsilon 2} \frac{\epsilon^2}{k} + \nu \frac{\partial^2 \epsilon}{\partial x_j \partial x_j} + \frac{\partial}{\partial x_j} \left(\frac{\nu_t}{\sigma_\epsilon} \frac{\partial \epsilon}{\partial x_j} \right), \tag{22}$$

where the eddy viscosity ν_t is defined as:

$$\nu_t = C_\mu \frac{k^2}{\epsilon}. \tag{23}$$

In these equations, $C_{\epsilon 1}$, $C_{\epsilon 2}$, σ_ϵ and C_μ are model constants. This model uses the k-transport equation in Eq. 18, where the dissipation rate ϵ is plugged in directly. The ω transport equation of the popular Wilcox k-ω model is as follows [64]:

$$\frac{\partial \omega}{\partial t} + \langle u_j \rangle \frac{\partial \omega}{\partial x_j} = -\gamma \frac{\omega}{k} \left\langle u_i' u_j' \right\rangle \frac{\partial \langle u_i \rangle}{\partial x_j} - \beta \omega^2 + \nu \frac{\partial^2 \omega}{\partial x_j \partial x_j} + \frac{\partial}{\partial x_j} \left(\sigma \nu_t \frac{\partial \omega}{\partial x_j} \right), \tag{24}$$

where the eddy viscosity is defined as:

$$\nu_t = \gamma^* \frac{k}{\omega}. \tag{25}$$

Again, this model uses the k-transport equation in Eq. 18, but obtains ϵ from

$$\epsilon = \beta^* k \omega. \tag{26}$$

In these equations, γ, β, β^*, σ and γ^* are model constants.

The main disadvantage of k-ϵ is its inability to accurately model boundary layers, especially in case of adverse pressure gradients. Also shear layers are not always accurately modeled by k-ϵ [65, p. 228]. The k-ω model is much better at modeling boundary layers, however, it suffers from a sensitivity to the specified inflow ω outside boundary layers. This makes the k-ω model often inferior to the k-ϵ model outside boundary layers. In 1994, Menter addressed the freestream sensitivity by introducing a blending function [31]. Rewriting the k-ϵ model to a k-ω formulation results in the following additional term in the transport equation:

$$2\frac{\sigma_{\omega 2}}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j}. \tag{27}$$

Menter's idea was to switch this term on outside boundary layers such that k-ϵ is used and switch it off inside boundary layers to use k-ω. To achieve this, Menter multiplied this term with $(1-F_1)$, where F_1 is a function that goes to one near walls and to zero away from walls, it is defined as:

$$F_1 = \tanh\left\{\left\{\min\left[\max\left(\frac{\sqrt{k}}{\beta^*\omega y}, \frac{500\nu}{y^2\omega}\right), \frac{4\sigma_{\omega 2}k}{CD_{k\omega}y^2}\right]\right\}^4\right\}. \tag{28}$$

In this expression, y is the distance to the closest wall, β^* and $\sigma_{\omega 2}$ are model constants and $CD_{k\omega}$ is defined as:

$$CD_{k\omega} = \max\left(2\frac{\sigma_{\omega 2}}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j}, 10^{-10}\right). \tag{29}$$

Note that in the original paper $CD_{k\omega}$ was bounded by 10^{-20}, but this was later increased to 10^{-10} [32].

In the same 1994 paper, Menter introduces the following limiter of ν_t:

$$\nu_t = \frac{a_1 k}{\max(a_1\omega, SF_2)}. \tag{30}$$

Here a_1 is a model constant, S is the invariant measure of the strain rate tensor and F_2 is defined as:

$$F_2 = \tanh\left[\left[\max\left(\frac{2\sqrt{k}}{\beta^*\omega y}, \frac{500\nu}{y^2\omega}\right)\right]^2\right]. \tag{31}$$

Similar to the F_1 function, F_2 is one near walls and zero away from walls. This new definition of ν_t limits the production of turbulent shear stress in boundary layers with strong adverse pressure gradients, where F_2 ensures no limiting occurs in shear layers. Note that in the 1994 model, the F_2 function was multiplied with the invariant measure of the rotation rate tensor in the ν_t limiter in Eq. 30, this was changed in the 2003 model [32]. The incompressible, constant-viscosity k-transport equation of the k-ω SST model is:

$$\frac{\partial k}{\partial t} + \langle u_j \rangle \frac{\partial k}{\partial x_j} = P_k - \beta^* k\omega + \nu\frac{\partial^2 k}{\partial x_j \partial x_j} + \frac{\partial}{\partial x_j}\left(\sigma_k \nu_t \frac{\partial k}{\partial x_j}\right). \tag{32}$$

The incompressible, constant-viscosity ω-transport equation of the k-ω SST model is:

$$\frac{\partial \omega}{\partial t} + \langle u_j \rangle \frac{\partial \omega}{\partial x_j} = \frac{\gamma}{\nu_t}P_k - \beta\omega^2 + \nu\frac{\partial^2 \omega}{\partial x_j \partial x_j} + \frac{\partial}{\partial x_j}\left(\sigma_\omega \nu_t \frac{\partial \omega}{\partial x_j}\right) + 2(1-F_1)\frac{\sigma_{\omega 2}}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j}. \tag{33}$$

In these transport equations, the k production term P_k is given as:

$$P_k = \min\left(-\left\langle u_i' u_j' \right\rangle \frac{\partial \langle u_i \rangle}{\partial x_j}, c_1 \beta^* k\omega\right). \tag{34}$$

Note that the limited P_k was initially only used for the k equation, only later was it also added to the ω equation [43].

The model coefficients used in k-ω SST as implemented in OpenFOAM are given below. As can be seen, σ_k, σ_ω, β and γ are not actually constant but dependent on the F_1 blending function. Similar to how the F_1 function blends between the k-ϵ and k-ω model, their model constants are blended as well. These model constants were calibrated by considering academic flows where many terms cancel. The k-ω SST model shows improvements over both the k-ϵ and the k-ω turbulence models. It has grown to become the most used RANS model today, often being referred to as the industry standard. Due to its popularity and relatively good accuracy, the k-ω SST model is used as the baseline RANS model in the current work [31] [32].

$$\begin{aligned}
&\beta^* = 0.09, && a_1 = 0.31, && c_1 = 10,\\
&\sigma_k = F_1\sigma_{k1} + (1-F_1)\sigma_{k2}, && \sigma_{k1} = 0.85, && \sigma_{k2} = 1.0,\\
&\sigma_\omega = F_1\sigma_{\omega 1} + (1-F_1)\sigma_{\omega 2}, && \sigma_{\omega 1} = 0.5, && \sigma_{\omega 2} = 0.856\\
&\beta = F_1\beta_1 + (1-F_1)\beta_2, && \beta_1 = 0.075, && \beta_2 = 0.0828\\
&\gamma = F_1\gamma_1 + (1-F_1)\gamma_2, && \gamma_1 = 5/9, && \gamma_2 = 0.44
\end{aligned}$$

While LEVMs have proven highly successful for many types of flows, they come with some fundamental shortcomings that make them inaccurate for certain kinds of flows. In his book, Wilcox mentions flows with sudden changes in mean strain rate and extra rates of strain as flow regimes where Boussinesq's hypothesis is invalid. Concrete cases where LEVMs fail include: flows with streamline curvature, flows with secondary motions, flows with re-circulation zones and three-dimensional flows [65, p. 303] [22]. Note that some inaccuracies also result from the lack of unsteadiness modeling in RANS. These effects occur in many practical engineering flows, meaning LEVMs may provide unsatisfactory predictions. For such flow cases, RANS models that do not depend on Boussinesq's hypothesis were developed, these are laid out next.

2.6 Other RANS models

In an effort to overcome the inherent shortcomings of Boussinesq's hypothesis laid out by Schmitt [47], various RANS models have been proposed that do not use it. One option is to solve the tensorial Reynolds stress equation (Eq. 17) directly, these models are known as Reynolds stress models (RSMs). Since the RST is symmetric, at least six PDEs have to be solved. As for LEVMs, a length and time scale are needed to close the system, which are usually obtained from either k and ϵ or k and ω. Luckily, k can be straightforwardly attained as the trace of the RST. However, ϵ/ω require the introduction of an additional PDE, bringing the total to seven. The main effort of designing a RSM lies in finding approximations for the higher order correlation terms in Eq. 17, such as $\langle u_i' u_j' u_k' \rangle$. The most famous RSMs are the Launder-Reece-Rodi (LRR) model and the Speziale-Sarkar-Gatski (SSG) model [27] [52].

Whereas LEVMs depend on local mean velocity gradients to find the RST, RSMs do not. This means that by nature, RSMs are able to capture flows with sudden changes in mean strain rate and extra rates of strain. For instance, RSMs are able to take into account streamline curvature and secondary motions. Given these advantages, one would expect RSMs to be the most used RANS models today. However, they come with several disadvantages which have resulted in two-equation LEVMs being the dominant RANS models. One obvious disadvantage is the increased computational cost of solving seven PDEs rather than just two. Also, RSMs are significantly less robust than LEVMs. Robustness is rather important in practical engineering problems, where the complex geometries often result in worse quality meshes compared to academic cases [37, p. 458] [65, p. 372]. Finally, Menter mentions that results often did not systematically improve when using RSMs compared to LEVMs [33].

Nonlinear eddy viscosity models (NLEVMs) are a mix between LEVMs and RMSs. They use more complex relations between the RST and the velocity gradient tensor than Boussinesq's hypothesis. Contrary to RMSs though, they typically only solve two transport equations, making their computational cost similar to LEVMs. A subset of NLEVMs are algebraic stress models (ASMs), which are further split in implicit and explicit versions. ASMs are characterized by the fact that they are derived from full RSMs. Rodi proposes the weak equilibrium assumption, by which the mean substantial derivative of the Reynolds stress is assumed relatively much smaller than the mean substantial derivative of the turbulent kinetic energy [41]. This allows RSMs to be written in a form where the Reynolds stress is determined using five implicit algebraic equations depending on k, ϵ and $\partial \langle u_i \rangle / \partial x_j$. Unfortunately these equations are implicit, which results in numerical stiffness, especially for complex flows encountered in practical engineering problems [15] [37, p. 448-452].

In order to address the numerical stiffness of implicit ASMs, the implicit algebraic equations are rewritten in an explicit form, resulting in what are known as explicit algebraic stress models (EASMs). Pope used an integrity basis to make the equations explicit, his equation for the RST is as follows [36]:

$$\langle u_i' u_j' \rangle = \frac{2}{3} k \delta_{ij} + k \sum_{\lambda} G^{\lambda} T_{ij}^{\lambda}. \tag{35}$$

The tensors T_{ij}^{λ} depend only on the normalized strain rate tensor s_{ij} and the normalized rotation rate tensor ω_{ij}, defined as:

$$s_{ij} = \frac{\tau}{2} \left(\frac{\partial \langle u_i \rangle}{\partial x_j} + \frac{\partial \langle u_j \rangle}{\partial x_i} \right), \tag{36}$$

$$\omega_{ij} = \frac{\tau}{2} \left(\frac{\partial \langle u_i \rangle}{\partial x_j} - \frac{\partial \langle u_j \rangle}{\partial x_i} \right). \tag{37}$$

In Eq. 36 and Eq. 37, τ is a timescale for nondimensionalization; Pope used $\tau = k/\epsilon$. Another option is to use the magnitude of the velocity gradient tensor ($\tau = 1/||\nabla U||$) as proposed by Spalart [50]. In the most general form, there are ten linearly independent tensors T^{λ}_{ij}:

$$
\begin{aligned}
T^1_{ij} &= s_{ij}, & T^6_{ij} &= \omega_{ik}\omega_{kl}s_{lj} + s_{ik}\omega_{kl}\omega_{lj} - \frac{2}{3}\left(s_{kl}\omega_{lm}\omega_{mk}\right)\delta_{ij},\\
T^2_{ij} &= s_{ik}\omega_{kj} - \omega_{ik}s_{kj}, & T^7_{ij} &= \omega_{ik}s_{kl}\omega_{lm}\omega_{mj} - \omega_{ik}\omega_{kl}s_{lm}\omega_{mj},\\
T^3_{ij} &= s_{ik}s_{kj} - \frac{1}{3}\left(s_{kl}s_{lk}\right)\delta_{ij}, & T^8_{ij} &= s_{ik}\omega_{kl}s_{lm}s_{mj} - s_{ik}s_{kl}\omega_{lm}s_{mj},\\
T^4_{ij} &= \omega_{ik}\omega_{kj} - \frac{1}{3}\left(\omega_{kl}\omega_{lk}\right)\delta_{ij}, & T^9_{ij} &= \omega_{ik}\omega_{kl}s_{lm}s_{mj} + s_{ik}s_{kl}\omega_{lm}\omega_{mj} - \ldots\\
& & & \quad - \frac{2}{3}\left(s_{kl}s_{lm}\omega_{mn}\omega_{nk}\right)\delta_{ij},\\
T^5_{ij} &= \omega_{ik}s_{kl}s_{lj} - s_{ik}s_{kl}\omega_{lj}, & T^{10}_{ij} &= \omega_{ik}s_{kl}s_{lm}\omega_{mn}\omega_{nj} - \omega_{ik}\omega_{kl}s_{lm}s_{mn}\omega_{nj}.
\end{aligned}
\tag{38}
$$

Each base tensor has an associated scalar function G^{λ}, which depends on at most five invariants:

$$
\begin{aligned}
I_1 = s_{kl}s_{lk}, \qquad I_2 = \omega_{kl}\omega_{lk}, \qquad I_3 = s_{kl}s_{lm}s_{mk},\\
I_4 = \omega_{kl}\omega_{lm}s_{mk}, \qquad I_5 = \omega_{kl}\omega_{lm}s_{mn}s_{nk}.
\end{aligned}
\tag{39}
$$

Deriving an EASM consists of finding the functions G^{λ} based on an implicit ASM. In his paper, Pope derived an EASM from the implicit ASM based on the LRR RSM, though only for 2D flow as 3D was mathematically too cumbersome [36]. Later, Gatski and Speziale derived the full 3D EASM from the LRR RSM using symbolic manipulation software, their G^{λ} functions are as follows [16]:

$$
\begin{aligned}
G^1 &= -\frac{1}{2}\left(6 - 3I_1 - 21I_2 - 2I_3 + 30I_4\right)/D_{GS}, & G^6 &= -9/D_{GS},\\
G^2 &= -\left(3 + 3I_1 - 6I_2 + 2I_3 + 6I_4\right)/D_{GS}, & G^7 &= 9/D_{GS},\\
G^3 &= \left(6 - 3I_1 - 12I_2 - 2I_3 - 6I_4\right)/D_{GS}, & G^8 &= 9/D_{GS},\\
G^4 &= -3\left(3I_1 + 2I_3 + 6I_4\right)/D_{GS}, & G^9 &= 18/D_{GS},\\
G^5 &= -9/D_{GS}, & G^{10} &= 0,
\end{aligned}
\tag{40}
$$

where

$$
D_{GS} = 3 - \frac{7}{2}I_1 + I_1^2 - \frac{15}{2}I_2 - 8I_1I_2 + 3I_2^2 - I_3 + \frac{2}{3}I_1I_3 - 2I_2I_3 + 21I_4 + 24I_5 + 2I_1I_4 - 6I_2I_4.
$$

Whereas (E)ASMs are derived from RSMs, it is also possible to construct a NLEVM purely from data and physical constraints using the same integrity basis in Eq. 38. Confusingly, these models are often referred to as NLEVMs, while the class of NLEVMs also contains (E)ASMs [15]. Hereafter, NLEVM will only be used to indicate these data calibrated models, while EASM refers to models derived from a RSM. The simplest NLEVMs, known as quadratic eddy viscosity models, only use the first four base tensors in Eq. 38. Craft et al. argue that these models are only slightly more general than LEVMs and instead propose a cubic eddy viscosity model using the first six base tensors [9]. An advantage of NLEVMs is that they are easier to design than (E)ASMs as they do not require a RSM. Though promising, no (E)ASM or NLEVM has shown sufficient improvements over LEVMs to see widespread use.

3 Data-driven turbulence modeling

Recent advancements in data-driven approaches together with the increasing availability of high-fidelity field data from numerical simulations has opened up a new discipline in turbulence modeling; data-driven turbulence modeling. The aim of this section is to give a brief history of data-driven turbulence modeling and cover techniques relevant to the current work. In Sec. 3.1, works using a Bayesian framework to recalibrate turbulence model coefficients are discussed as well as types of uncertainty in RANS simulations. Then, in Sec. 3.2, works that train either a neural network or random forests to predict the full RST are discussed. Arising stability issues in an *a-posteriori* setting are analyzed. Next, the works discussed in Sec. 3.3 aim to prevent instability by modeling only the RST difference between LEVM and high-fidelity. One framework in particular, SpaRTA, is used in the current work, so it is discussed in more detail in Sec. 3.4.

3.1 Coefficient calibration and uncertainty classification

Early efforts to use data to improve the predictive capability of RANS models focused on the recalibration of their coefficients. One of the first to do so were Cheung et al., who used a Bayesian framework to find posterior probability distributions of model coefficients of the Spalart-Allmaras turbulence model [8]. Incompressible boundary layers over a flat plate subjected to various streamwise pressure gradients were used as training data. Only two coefficients could be recalibrated, their optimal values were notably different from their actual values. Using the same framework, Edeling et al. recalibrated the coefficients of five RANS models for a broader set of boundary layers under a streamwise pressure gradient [14]. They calibrated the coefficients for each case separately and found that each case had significantly different optimal coefficients.

Recalibrating model coefficients for certain cases can improve results and it is not uncommon in industry to find custom sets of coefficients, recalibrated to the flow in question [68]. However, recalibration requires high fidelity data which is not always available and improvements may deteriorate with slight changes in geometry. All in all, it does not seem possible to find a universally applicable improved turbulence model simply by recalibrating coefficients. To explain this conclusion, it is useful to consider the uncertainty classification introduced by Duraisamy et al., listing the following four uncertainties in RANS models [12]:

- **L1** : Uncertainties stemming from the loss of information in the averaging process.
- **L2** : Uncertainties from the representation of a microscopic quantity (the Reynolds stress) using macroscopic quantities.
- **L3** : Uncertainties in the functional form of the model to relate the Reynolds stress to macroscopic quantities.
- **L4** : Uncertainties in the model coefficients to relate the Reynolds stress to macroscopic quantities.

Coefficient recalibration at best removes all L4 uncertainties. However, there are many practical flows where the other uncertainties are also important for LEVMs. Hence, in the search for improved turbulence models, later data-driven approaches have mostly focused on L2 and L3 uncertainties [12]. A number of these approaches are explored in the coming sections.

3.2 Reynolds stress tensor modeling

Initial efforts to address L2 and L3 uncertainties modeled the RST directly, removing Boussinesq's hypothesis. Thus these are not NLEVMs but a whole class of RANS models their own. At a bare minimum, new models should be Galilean invariant, meaning results are the same for each inertial reference frame. Ling et al. train a neural network to predict the nondimensional, anisotropic RST, where a special architecture is used to ensure Galilean invariance [28]. They use the Pope tensor basis in Eq. 35, where the outputs of the neural network are the scalars G^{λ}. As inputs, they use the five scalar invariants in Eq. 39. In order to make the model as general as possible, they trained on five distinct flows and tested on two other distinct flows. Results were compared to an existing LEVM and NLEVM, improvements were found compared to both, but results were still far off the DNS data.

In the same paper, Ling et al. attempted to perform *a-posteriori* tests (implementation in a full RANS solver). However, their approach can at best be called *semi-a-posteriori*; they evaluated the RST just once on the converged fields of the LEVM and converged the solution a second time while keeping this new RST fixed. The results again showed improvements over the LEVM and NLEVM, but were still not matching the DNS [28]. Kaandorp and Dwight mention several advantages of random forests over neural networks, such as easier implementation and training [23]. Ling et al. used a neural network as it was easier to embed Galilean invariance. Nonetheless, Kaandorp and Dwight manage to implement a random forests algorithm with embedded Galilean invariance. Like Ling et al., they trained on one set of cases and tested on another set of cases, they found similar performance to the neural network of Ling et al. Interestingly, they found large performance improvements when expanding the input space by introducing additional invariants dependent on ∇k.

Kaandorp and Dwight also attempted to test their trained random forest in an *a-posteriori* setting. They noted instability issues; similar issues likely necessitated the *semi-a-posteriori* strategy employed by Ling et al. Kaandorp and Dwight addressed the stability issues by blending their anisotropy tensor between LEVM (k-ω) and random forest prediction, starting the run fully LEVM and gradually going to random forest. However, they could only achieve a maximum blending of 80% random forest without diverging. With this setup, they attained a better *a-posteriori* match with experimental data than found by Ling et al. This performance increase likely originates from the expanded input space.

To explain the propagation instabilities, consider the work by Thomson et al, who propagated the exact RST fields from DNS data in a RANS solver [55]. They did this for plane channel flow at various Reynolds numbers using DNS datasets from various groups. Surprisingly, the errors in the propagated velocity fields were much bigger than the errors in the DNS RST data, especially for large Reynolds numbers. Note that predicting the exact DNS RST is a best-case scenario; the errors in the RST prediction by Ling et al. and Kaandorp and Dwight were much bigger. Thus, amplification of these errors likely led to the propagation instabilities when injecting the predicted raw RST into the RANS solver.

3.3 Reynolds stress tensor discrepancy modeling

The amplification of RST errors in propagation is not observed in LEVMs, where velocity and pressure are often predicted with a higher accuracy than the RST [47]. Given the instabilities of direct data-driven modeling of the RST, inclusion of a LEVM in the model may improve stability. The group of Moser is accredited as being the first to find models only for the discrepancy of the RST, rather than the full RST [66]. Their idea is given in mathematical form as:

$$\left\langle u_i' u_j' \right\rangle = -2\frac{\nu_t}{\tau} s_{ij} + 2k b_{ij}^{\Delta} + \frac{2}{3} k \delta_{ij}, \tag{41}$$

where the notation of Schmeltzer et al. [46] is used rather than Moser. In short, a LEVM is still used to (largely) calculate the RST, but an additional tensor is included to correct the shortcomings of the LEVM. Here, b_{ij}^{Δ} is a nondimensional, anisotropic tensor, often modeled using Pope's tensor basis [36]. In this section, various approaches to modeling b_{ij}^{Δ} are explored. Note that various LEVMs are used and some authors also modify the LEVM.

In Sec. 3.3.1, early work by the group of Xiao is laid out who train random forests to predict the RST discrepancy. Rather than Pope's tensor basis, they use projections of the RST to ensure Galilean invariance. Then in Sec. 3.3.2, initial work by the group of Sandberg is summarized, who train algebraic expressions as the scalar models in Pope's tensor basis. They use gene expression programming to find these expressions. Furthermore, their frozen approach is discussed which aims at addressing the mismatch between the RANS and high-fidelity time scale. Next, early work of the group of Dwight is laid out in Sec. 3.3.3, who extend the frozen approach to also account for a discrepancy in turbulent kinetic energy between RANS and high-fidelity. Dwight's group also use algebraic expressions as models, but they use sparse regression to find these. Finally, later work by Dwight's group looking at applying their framework to wind turbines is discussed in Sec. 3.3.4.

3.3.1 Random forests by Xiao group

Wang et al. used random forests to predict Reynolds stresses of square ducts and separated flows [60]. Possibly due to the difficulties of combining random forests with Pope's tensor basis noted by Ling et al. [28], they did not use Pope's tensor basis. Instead, they trained discrepancies of six normalized projections of the RST. Most of their inputs came from Ling and Templeton [29], but they also added their own input based on the streamline curvature. A significant disadvantage of this input, however, is the need to specify a characteristic length. Ling and Templeton constructed their input features based on physical intuition, making sure to nondimensionalize and normalize them for generalizability. Wang et al. found a good RST match with DNS when testing their model at a higher Reynolds number [60]. However, when they tested on a slightly different separated flow geometry, there was only a slight improvement over the baseline LEVM.

In a later paper, Wang et al. added 47 more features to the inputs [59]. These were derived in a similar fashion to the invariants of the Pope tensor series in Eq. 39. However, they also included the asymmetric tensors associated with the gradient of k and the gradient of p. In this paper, they again trained a random forest to predict RST discrepancy, using normalized projections of the RST to ensure Galilean invariance. Training took place on a square duct at various Reynolds numbers, while testing was performed on the same geometry at a higher Reynolds number. Again, the predicted RST showed good agreement with the RST in DNS. In this paper they also tested the model *a-posteriori*, where they also observed improvements over the baseline NLEVM. They compared a random forest with only the 10 input features from their original paper and with 57 inputs (using the 47 additional invariants). The random forest with 57 inputs performed significantly better *a-posteriori*, advocating for the use of a large input space.

3.3.2 Gene expression programming and the frozen approach by Sandberg group

In the work laid out so far, authors attempted to train either a neural network or random forest to predict the RST (discrepancy). Weatheritt and Sandberg argue against this approach, noting several fundamental issues with a neural network/random forest and calling it the ultimate black box [61]. Firstly, such a black box is difficult to implement in an actual CFD solver. Secondly, it is impossible to understand the nature of the correction being applied by a black box (though random forests may give some understanding, as illustrated in Fig. 2 in [60]). Thirdly, especially neural networks are infamous for overfitting and bad performance in extrapolation, which makes the models less generalizable. To address these issues, Weatheritt and Sandberg instead propose symbolic expressions as models.

Weatheritt and Sandberg fit the nondimensional, anisotropic discrepancy of the RST (b_{ij}^{Δ} in Eq. 41), noting that they observed divergence when modeling the full RST. They use the first four tensors of Pope's tensor basis in Eq. 38. Their goal is then to find expressions for the first four scalar functions G^{λ}, which depend on the first two invariants in Eq. 39. As laid out in Sec. 2.6, finding expressions for these scalar G^{λ} functions is also the goal of NLEVMs. The functions used in existing NLEVMs are already quite complex, so Weatheritt and Sandberg's framework should be able to find rather complex functions. Fitting a function of arbitrary form to data is known as symbolic regression.

Weatheritt and Sandberg use gene expression programming (GEP) for symbolic regression, adding the concept of separate 'plasmids' to distinguish between tensors and scalars. This modification is necessary to ensure the output is always a nondimensional, Galilean invariant tensor. Training was performed on a backward facing step, while testing took place on the same backward facing step at higher Reynolds number as well as on a periodic hill. They noted that their framework produced simple, stable models. They tested their models both *a-priori* and *a-posteriori* and found improvements over conventional EASMs, though their models were not yet matching the DNS [61].

In later work, Weatheritt and Sandberg noted that in industrial cases, DNS and wall-resolved LES would be too expensive to use as high fidelity training data [62]. Instead, they use detached eddy simulation (DES) data for training so as to test its feasibility. Also, they note that the turbulent time scale used in RANS is different from the turbulent time scale used in scale resolving simulations. This stems from the empirical nature of the RANS closure equations and their cal-

ibration to boundary layers. To address the difference in time scales, they introduce the frozen approach, in which they freeze the high-fidelity velocity and RST fields. They then solve the ω transport equation to obtain the RANS time scale corresponding to the high-fidelity data. Furthermore, they apply an *ad hoc* correction to the production of k, noting nonconvergence with the original production term .

Using this frozen approach, Weatheritt and Sandberg trained on DES data of two ducts of different aspect ratio. Again, they find relatively simple, stable models which they test *a-posteriori* in a duct with higher aspect ratio as well as in a diffuser. They demonstrate significant improvements over both the LEVM as well as over conventional EASMs, even for the diffuser which is rather different than the training data. However, their model is still not matching the DNS results closely. Most importantly though, they show that fully resolved DNS data is not needed for training; DES suffices.

3.3.3 Sparse regression and k-corrective-frozen by Dwight group

Schmelzer et al. extend the frozen approach by addressing the problems with the production of k, noted by Weatheritt and Sandberg, in a more systematic way [46]. They argue that similar to the turbulent time scale, the turbulent kinetic energy in RANS may be different from the scale resolved turbulent kinetic energy. Note that the effect of b_{ij}^{Δ} is already included in the k and ω transport equations (Eq. 32 and Eq. 33 respectively) through the modified RST in Eq. 41. To address the difference in k between RANS and scale resolved, they replace P_k in both transport equations by $P_k + R$, where R is the residual of the k transport equation. In their frozen approach, called k-corrective-frozen-RANS, they again freeze the high fidelity velocity and RST and iteratively solve the k and ω transport equations to find R, ω and b_{ij}^{Δ}.

In order to validate that the found R and b_{ij}^{Δ} fields indeed propagate to the high-fidelity solution in a full RANS solver, Schmelzer et al. introduce an approach that is referred to as propagation in this work. Note that some authors refer to the testing of a trained model in a RANS solver as propagation; this will be referred to as model propagation. Models are trained to predict both R and b_{ij}^{Δ}; propagation is the upper performance limit in the sense that it represents a hypothetical model that exactly predicts R and b_{ij}^{Δ}. For each training case, Schmelzer et al. find an exact match with the high-fidelity data in this propagation step.

Models for R and b_{ij}^{Δ} can be in any form laid out before; neural networks, random forests or symbolic expressions. Schmelzer et al. choose symbolic expressions following the aforementioned arguments of Weatheritt and Sandberg (Sec. 3.3.2). Schmelzer et al. use Pope's tensor basis in Eq. 38 for b_{ij}^{Δ} models. For R models, a new scalar basis is devised based on the P_k modification due to each Pope base tensor:

$$R = 2k \sum_{\lambda} G^{\lambda} T_{ij}^{\lambda} \frac{\partial \langle u_i \rangle}{\partial x_j}. \tag{42}$$

This basis ensures that R has the correct dimension (length^2/time^3), since G^{λ} and T_{ij}^{λ} are nondimensional. Finally, note that finding expressions for G^{λ} happens completely independently for R and b_{ij}^{Δ}.

Schmelzer et al. note that the symbolic regression method used by Weatheritt and Sandberg (GEP) has the disadvantage of being non-deterministic. Furthermore, GEP appears rather difficult to implement. Noting that models with few nonlinear terms showed low errors and were numerically robust, Schmelzer et al. opted for a different approach to symbolic regression; Sparse regression of turbulent stress anisotropy (SpaRTA). In short, SpaRTA constructs a library of all possible single terms for either R or b_{ij}^{Δ}. A linear combination of the terms in the library is then fitted to the data. To prevent overfitting, elastic net regression [70] is used to find sparse models with small coefficients. A more elaborate explanation of SpaRTA is provided in Sec. 3.4.

Using their SpaRTA framework, Schmelzer et al. separately train models on a periodic hill, a converging-diverging channel and a curved backward-facing step, all characterized by a separation bubble. Since all cases are 2D, they only used the 2D subset of Pope's tensor basis (three tensors; two invariants). Each model is propagated for each case and the best model of each case is analyzed in more detail, resulting in a detailed analysis of three models. Interestingly, two of these models only had a model for R ($b_{ij}^{\Delta} = 0$). All three models showed significant improvements over the LEVM for all three cases, however, the match with DNS was still far from the upper limit found in the propagation step. These models were also tested for a periodic hill at significantly higher Reynolds number than the training data. Also for this case, significant improvements were found over the LEVM. Finally, note that algebraic expressions were always trained and tested on similar geometries. However, if a model is to see widespread use in the turbulence community, it should prove itself over a wide range of flows.

3.3.4 SpaRTA applied to wind-turbines by Dwight group

The SpaRTA framework (including k-corrective-frozen-RANS) was later applied to wind turbine modeling by Steiner et al. (same group as Schmelzer et al.) [53]. This is a case with much more industrial relevance than the academic separation bubble cases used by Schmelzer et al. As such, many more difficulties were encountered that had to be addressed. Firstly, there was already a mismatch between the LES and RANS boundary layer development even without a turbine. SpaRTA was used to find correction models for this discrepancy only, so as to separate it from discrepancies in rotor modeling. Secondly, the larger number of cells together with many more input features (discussed next) resulted in a significant increase in data in the library. To alleviate memory requirements and training time, mutual information and cliqueing were used to reduce library size, these are discussed in more detail in Sec. 3.4.

In the work of Wang et al. [59] and Kaandorp and Dwight [23], a significant increase in predictive capability of the random forest was found when using input features based also on ∇k and ∇p. Steiner et al. note this as well and massively expand the space of scalar variables (so-called features) available to SpaRTA [53]. They use the 47 invariants introduced by Wang et al., which are based on s_{ij}, ω_{ij} and the tensorized versions of ∇k and ∇p [59]. Furthermore, they use 11 q-features, most of which come from Wang et al. [60] who in turn got most from Ling and Templeton [29]. One particular feature, vortex stretching, was not included by Wang et al. as their cases were 2D. However, Steiner et al. also did not use this feature even though their case is 3D, possibly due to an oversight. Also, it should be noted that not all q-features are Galilean invariant and a number of features depend on the ∇p. According to Spalart, this automatically disqualifies these features from use in a turbulence model [51]. Finally, Steiner et al. also include the dissipation rate ϵ as a basis for R [53].

Steiner et al. also apply the propagation step introduced by Schmelzer et al. and find excellent agreement with LES. Their trained models are much more complicated than those found by Schmelzer et al., likely due to the flow being more complex and a larger number of features available to the symbolic regressor. A number of their models were numerically unstable, likely due to the more complex algebraic structure. Furthermore, they noted that corrections were small outside the wake region, resulting in terms that cancelled outside the wake. Nonetheless, their models showed significant improvements over the LEVM, though results were still far from the upper limit found in propagation.

To address the complicated models with cancelling terms, Steiner et al. propose the use of a classifier which activates only in regions where corrections are needed [53]. In later work, they develop the framework for such a classifier [54]. The expression for their classifier σ is as follows:

$$\sigma := \begin{cases} 1 \text{ if } \left(\frac{\left| 2k b_{ij}^{\Delta} \left(\partial \langle u_i \rangle / \partial x_j \right) \right|}{\left| P_{k,LES} \right| + \epsilon} > 0.2 \right) \cup \left(\frac{|R|}{\left| P_{k,LES} \right| + \epsilon} > 0.2 \right) \\ 0 \text{ otherwise} , \end{cases} \tag{43}$$

where $P_{k,LES}$ is the LES production of k and $\epsilon = 0.01$ to prevent division by zero. The classifier is a binary function, being one in regions where the additional production of k due to either b_{ij}^{Δ} or R is more than 20% of the production in the high fidelity data (in this case LES) and zero outside these regions. A symbolic expression is trained to approximate σ using SpaRTA. Training of R and b_{ij}^{Δ} only occurs on points for which this classifier is active. Steiner et al. indeed found simpler models with this classifier approach, however, the models were numerically unstable and performed slightly worse than the more complicated models found without a classifier.

3.4 Detailed description of SpaRTA

After the promising paper by Schmelzer et al. on the use of the SpaRTA framework to train symbolic expressions for the correction fields found from k-corrective-frozen-RANS [46], the framework has been further developed. SpaRTA will also be used in the current work, which is conducted in the same research group. Hence, the SpaRTA framework including its recent developments is discussed in more detail in the current section. The technical flow diagram of the first version of SpaRTA is shown in Fig. 2. Furthermore, the affected k-ω SST equations are repeated here, the new Reynolds stress tensor equation is:

$$\left\langle u_i' u_j' \right\rangle = -2 \frac{\nu_t}{\tau} s_{ij} + 2k \boldsymbol{b}_{\boldsymbol{ij}}^{\boldsymbol{\Delta}} + \frac{2}{3} k \delta_{ij}. \tag{44}$$

The new k-transport equation is:

$$\frac{\partial k}{\partial t} + u_j \frac{\partial k}{\partial x_j} = P_k + \boldsymbol{R} - \beta^* k \omega + \frac{\partial}{\partial x_j} \left[\left(\nu + \sigma_k \nu_t \right) \frac{\partial k}{\partial x_j} \right]. \tag{45}$$

The new ω-transport equation is:

$$\frac{\partial \omega}{\partial t} + u_j \frac{\partial \omega}{\partial x_j} = \frac{\gamma}{\nu_t} \left(P_k + \boldsymbol{R} \right) - \beta \omega^2 + \frac{\partial}{\partial x_j} \left[\left(\nu + \sigma_\omega \nu_t \right) \frac{\partial \omega}{\partial x_j} \right] + 2 \left(1 - F_1 \right) \frac{\sigma_{\omega 2}}{\omega} \frac{\partial k}{\partial x_j} \frac{\partial \omega}{\partial x_j}. \tag{46}$$

The new production of k equation is:

$$P_k = min\left(2\left(\frac{\nu_t}{\tau}s_{ij} - k\boldsymbol{b}_{ij}^{\Delta}\right)\frac{\partial u_i}{\partial x_j}, 10\beta^*\omega k\right). \tag{47}$$

The equations for ν_t, F_1 and F_2 remain the same, they are found in Sec. 2.5.4.

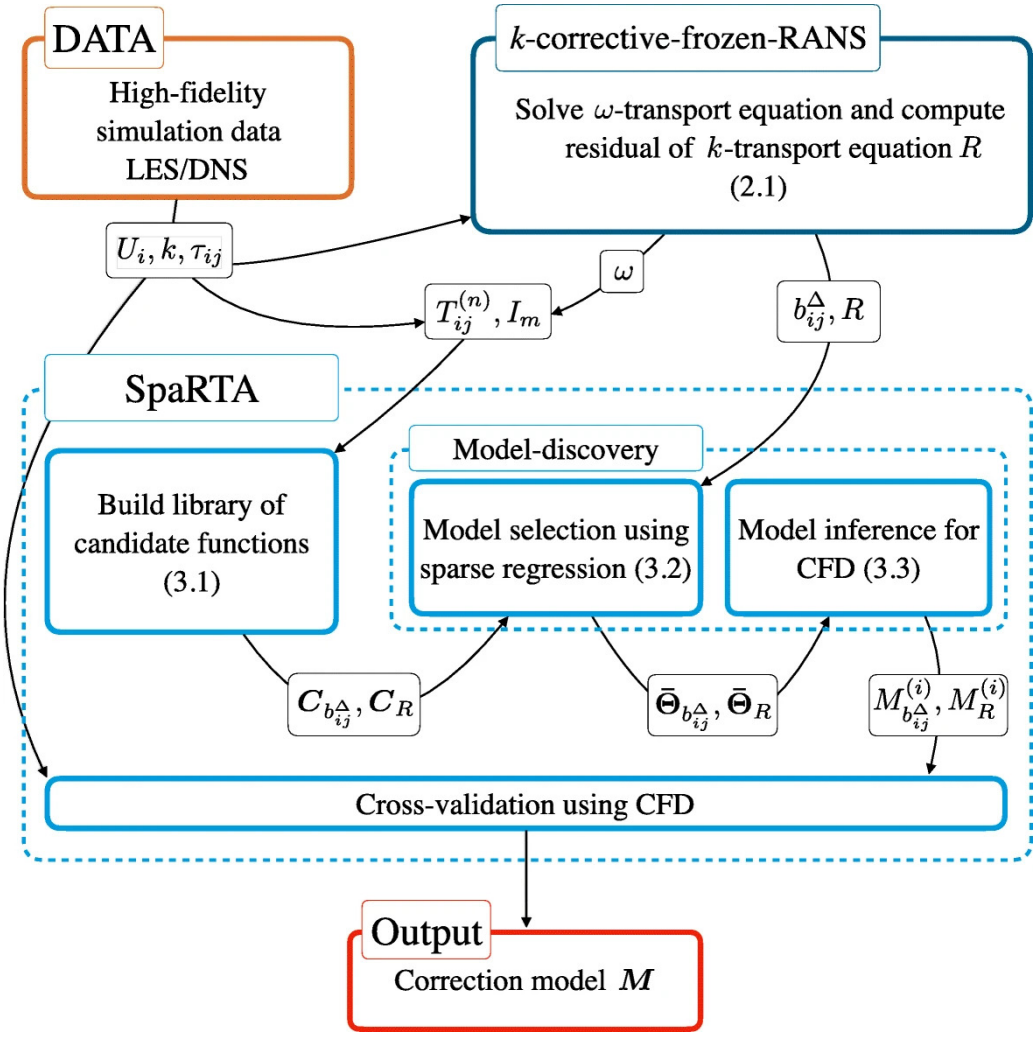


Figure 2: Technical flow diagram of SpaRTA (taken from [46]).

The first step of SpaRTA is the construction of a library of candidate functions. The inputs for this step are the base tensors (T_{ij}^n in Fig. 2) and the scalar invariants (I_m in Fig. 2). For R models, the base tensors are converted to base scalars using Eq. 42. Furthermore, the dissipation rate $\epsilon = k\omega$ was later added as a basis for R [53]. Also, for I_m Schmelzer et al. only used Pope's first two invariants I_1 and I_2 [46] [36]. However, as explained in Sec. 3.3.4, the number of input features was later expanded by also including invariants based on ∇k and ∇p as well as q-features from Ling et al. [29]. For the construction of the library, a sub-library $\boldsymbol{\mathcal{B}}$ of the scalar invariants is constructed first. This sub-library contains combinations of (functions of) input features, the $\boldsymbol{\mathcal{B}}$ used by Schmelzer et al. contains 16 terms, it is given as:

$$\begin{aligned}\boldsymbol{\mathcal{B}} = \big[&1, I_1, I_2, I_1^2, I_2^2, I_1^2I_2^3, I_1^4I_2^2, I_1I_2^2, I_1I_2^3 \\ &I_1I_2^4, I_1^3I_2, I_1^2I_2^4, I_1^2I_2, I_1I_2, I_1^3I_2^2, I_1^2I_2^2\big]^T.\end{aligned} \tag{48}$$

In order to keep the size of the library manageable, the number of input features per term should not exceed a certain global degree (2 for Schmelzer et al.). Within this global degree, each combination of (functions) of input features is added to $\mathcal{B}$. Thus, $\mathcal{B}$ rapidly grows with the number of input features and the number of functions used. Finally, $\mathcal{B}$ is multiplied with each b_{ij}^{Δ} basis to obtain the full library $\mathcal{C}_{b_{ij}^{\Delta}}$ for b_{ij}^{Δ} models:

$$\mathcal{C}_{b_{ij}^{\Delta}} = \left[\mathcal{B}T_{ij}^{1}, \mathcal{B}T_{ij}^{2}, \ldots, \mathcal{B}T_{ij}^{10}\right]^{T}. \tag{49}$$

Also $\mathcal{B}$ is multiplied with each R basis to obtain the full library the library $\mathcal{C}_R$ for R models:

$$\mathcal{C}_R = \left[\mathcal{B}\epsilon, \mathcal{B}\left(2kT_{ij}^{1}\frac{\partial\langle u_i\rangle}{\partial x_j}\right), \mathcal{B}\left(2kT_{ij}^{2}\frac{\partial\langle u_i\rangle}{\partial x_j}\right), \ldots, \mathcal{B}\left(2kT_{ij}^{10}\frac{\partial\langle u_i\rangle}{\partial x_j}\right)\right]^{T}. \tag{50}$$

Although this 'brute force' library approach worked for Schmelzer et al., it was not feasible for Steiner et al. due to the considerably larger number of input features and bigger meshes used in their study. Two techniques are used to reduce the size of the library; mutual information and cliqueing. Mutual information is a measure of how much information one has about a variable by observing a different variable. The benefit of mutual information is that it works for any functional relation between the variables. In SpaRTA, the k-nearest neighbour method is used to estimate the mutual information between input features and the correction term [17, p. 24-29]. Steiner et al. only used the ten features with the highest mutual information in their library [53].

Cliqueing is performed after the library of candidate functions has been constructed. In this step, the correlation coefficient between all candidates in the library is computed based on the training data. Cliques of candidate functions with a mutual correlation correlation coefficient of at least 0.99 are formed. Each clique is then reduced to a single candidate function in the library; the algebraically simplest one in the clique. For this step, use of the linear correlation coefficient is allowed as models are linear combinations of candidate functions [53].

After a (reduced) library of candidate functions has been established for R and b_{ij}^{Δ}, models are discovered as linear combinations of the candidate functions. The corrections predicted by the model are calculated as $R = \mathcal{C}_R\boldsymbol{\Theta}_R$ and $b_{ij}^{\Delta} = \mathcal{C}_{b_{ij}^{\Delta}}\boldsymbol{\Theta}_{b_{ij}^{\Delta}}$, where $\boldsymbol{\Theta}$ is a vector of coefficients (one coefficient for each library candidate function). Finding a model thus consists of finding the coefficients in $\boldsymbol{\Theta}$. A simple least-squares fit would result in many large, nonzero coefficients and possibly overfitting of the data. To promote sparsity and small coefficients in $\boldsymbol{\Theta}$, elastic-net regression is used instead [46]. Elastic-net regression was proposed by Zou and Hastie, it blends Ridge and Lasso regression to take advantage from both their benefits [70]. The equation for elastic net regression of $\boldsymbol{\Theta}$ is given as:

$$\boldsymbol{\Theta} = \arg\min_{\hat{\boldsymbol{\Theta}}} \left\|\mathcal{C}_{\Delta}\hat{\boldsymbol{\Theta}} - \boldsymbol{\Delta}\right\|_2^2 + \lambda\rho\|\hat{\boldsymbol{\Theta}}\|_1 + 0.5\lambda(1-\rho)\|\hat{\boldsymbol{\Theta}}\|_2^2 \tag{51}$$

In Eq. 51, $\|\boldsymbol{\Theta}\|_1$ corresponds to Lasso-regression while $\|\boldsymbol{\Theta}\|_2$ corresponds to Ridge regression. Lasso-regression promotes sparsity while Ridge-regression promotes small (though nonzero) coefficients. The two are blended via the hyperparameter ρ; $\rho = 1$ corresponds to pure Lasso-regression, $\rho = 0$ to pure Ridge-regression. The relative importance of these regularization terms compared to the least-squares term is controlled through the hyperparameter λ. A large value of λ results in a sparser $\boldsymbol{\Theta}$ with smaller coefficients at the cost of a worse least-squares fit. To prevent bias towards higher magnitude candidate functions, they are all standardised. Schmelzer et al. go over an approximately logarithmically spaced range of ρ and λ to find a variety of models [46].

As mentioned, all candidates were standardised when finding models using the elastic net. In the model inference step (see Fig. 2), the nonzero coefficients of these models are refit for unstandardised candidate functions. The number of model terms remains the same in this step, but the refit coefficients should still be small to promote stability. Hence, pure Ridge-regression is used for this refit, corresponding to Eq. 51 with $\rho = 0$ and only using the columns with nonzero coefficients. The refitted models for R and b_{ij}^{Δ}, $M_R^{(i)}$ and $M_{b^{\Delta}}^{(i)}$ respectively, are then tested a-posteriori in a CFD solver.

Though SpaRTA has been successfully applied to many cases and has shown significant improvements over LEVMs, it has never come close to the best-case scenario of propagation. This discrepancy is speculated to originate mostly from a misfit in b_{ij}^{Δ} models. The model prediction error was always larger for b_{ij}^{Δ} models compared to R models. Furthermore, the best model found by Schmelzer et al. used $b_{ij}^{\Delta} = 0$ [53] [46]. This difficulty is understandable considering that b_{ij}^{Δ} is a traceless, symmetric tensor while R is a scalar; b_{ij}^{Δ} essentially has to fit five times more data. Hemmes notes that SpaRTA is rather inflexible and uses a different framework for symbolic regression [20]. Likely, Hemmes was referring to the fact that SpaRTA only fits coefficients of a linear combination of terms, severely limiting the functional forms SpaRTA can regress. For instance, SpaRTA would not be able to regress the EASM by Gatski laid out in Sec. 2.6 [16].

4 Expansion of SpaRTA methodology

Consider again the steps laid out in the introduction for training and testing a model for a given case. Custom solvers and turbulence models are already available for running the first three steps (baseline, frozen, propagation). Also, the SpaRTA framework is available for the symbolic regression in the fourth step. However, no general solver was yet available for the fifth step, such a solver is developed in the current work, it is laid out in Sec. 4.1. Furthermore, the linear nature of the coefficient fitting in SpaRTA is found to be rather limiting in terms of model accuracy. Hence, a new symbolic regression framework is developed in the current work which is able to fit coefficients in nonlinear functions. This new framework named CuRTA is explained in Sec. 4.2.

4.1 Model propagation infrastructure

The current section covers the custom OpenFOAM solver and turbulence model created for propagation of correction models, developed as part of the current work. In Sec. 4.1.1, the motivation for creating this solver and turbulence model is given. Then, the algorithm and further implementation details are given in Sec. 4.1.2 and a brief usage guide is provided in Sec. 4.1.3.

4.1.1 Problem statement

The symbolic regression step generates a plethora of potential models, stemming from the trade-off between accuracy and number of terms. Model accuracy and stability cannot be exactly determined *a-priori*, meaning a significant number of models has to be tested *a-posteriori* to determine the most suitable one. Testing models *a-posteriori* may reveal flaws, leading to new model generation approaches. All in all, the *a-posteriori* model testing process needs to be as flexible, optimized and automated as possible.

In previous work, model testing was performed by compiling the models into a custom OpenFOAM turbulence model, coding the necessary features into OpenFOAM. This approach is rather inflexible, as OpenFOAM has to be recompiled for each new model. Furthermore, models/features have to be written in the C++ language, which has less library functionality compared to languages such as Python. Also, the approach is not optimized, as a recompile of a custom turbulence model takes around one minute. Once compiled though, the model is evaluated efficiently. Finally, the approach is hard to automate, owing to the manipulation of source code. Running two instances of a source code modifying script could lead to problems when they are concurrently modifying the custom turbulence model.

In the current work, a new approach is introduced which uses a universal model propagation turbulence model that only needs to be compiled once. This is done by removing the model dependent part (computation of required features followed by computation of correction fields) from the compiled code. Instead, this part of the computation is outsourced to an interpreted language (a language that is not pre-compiled). When correction fields are needed, the new turbulence model sends relevant flow fields such as $\nabla_i u_j$ to the interpreted language script, which sends back the correction fields. Given that the SpaRTA infrastructure is already written in Python, Python is used as the interpreted language. To circumvent the slower speed of Python compared to C++, the NumPy library is used for computations, as NumPy does these at roughly C++ speed. Outsourcing the model evaluation to Python was not implemented before, as such an interface

was deemed to difficult to implement. However, with the paper by Maulik et al. [30], implementation becomes feasible as they provide all code for such an interface in OpenFOAM-8.

4.1.2 Implementation details

The custom turbulence model `modelPropagationkOmegaSST` contains the OpenFOAM-Python interface, its code is contained in a header and a source file given in Appendix D.1.1 and Appendix D.1.2 respectively. The header file only contains declarations and initializations of functions and variables, while the source file contains the actual code. The algorithm of this modified turbulence model is summarized in Algorithm 1, the algorithm of the associated Python script 'model_definiton.py' is summarized in Algorithm 2. The implementation of these algorithms into actual C++ and Python code is explained next.

Algorithm 1 OpenFOAM side of model propagation.

Start Python interpreter and load Python file ▷ Executed at startup
procedure TURBULENCEMODEL ▷ Executed each iteration
    $R,\ b_{ij}^{\Delta},\ \sigma = \texttt{model}(\nabla_i u_j,\ k,\ \omega,\ \nabla_i p,\ \nabla_i k,\ \nu_t,\ u_i,\ y,\ \nu,\ \epsilon_{ijk}\nabla_i u_j)$ ▷ Python model evaluation
    $\omega = \texttt{solve}(\text{Eq. 46})$ ▷ Solve modified ω transport equation
    $k = \texttt{solve}(\text{Eq. 45})$ ▷ Solve modified k transport equation
    $\nu_t = a_1 k / min(a_1\omega, SF_2)$ ▷ Calculate eddy viscosity (Eq. 30)
    $\langle u_i' u_j' \rangle - 2/3 k\delta_{ij} = -(\nu_t + \nu)\left(\nabla_i u_j + \nabla_j u_i\right) + 2k b_{ij}^{\Delta}\sigma$ ▷ Calculate modified RST

Algorithm 2 Python side of model propagation (called by Algorithm 1).

$M_R,\ M_{b^{\Delta}},\ M_{\sigma} = \texttt{readModelFiles}(\text{casePath})$ ▷ Read model files, executed at startup
procedure MODEL(Γ) ▷ Model evaluation function, input is flow variables
    $\Psi = \texttt{setupFlow}(\Gamma)$ ▷ Calculate features and bases based on flow variables
    $R,\ b_{ij}^{\Delta},\ \sigma = M_R(\Psi),\ M_{b^{\Delta}}(\Psi),\ M_{\sigma}(\Psi)$ ▷ Evaluate models based on features and bases
    return $R,\ b_{ij}^{\Delta},\ \sigma$

The OpenFOAM source file can be split into two parts; the startup part which is run once at the start of the program and the iterative part which is run each solver iteration. In the startup part, a Python interpreter is started and the 'model_definition.py' Python script in the case directory is loaded and executed. In this initial execution of the Python script, global variables are defined/-calculated and the equation files are read in. Furthermore, all functions are loaded in, including the `model` function which is later called by OpenFOAM. During the run, the Python interpreter is kept alive along with all variables/functions loaded in this step, such that the associated overhead is avoided in the iterative part. Also, the array that is converted and send to Python each iteration is initialized at startup. Dynamic allocation is used rather than static allocation used by Maulik et al., because of the large number of cells present in some cases.

In the iterative part, the latest value of a number of variables is loaded into the pre-initialized array which has size $(N_{cells} \times N_{scalars})$. Here N_{cells} is the number of mesh cells and $N_{scalars}$ is the number of scalars send to Python; a vector for example has three scalars. The following variables are send to Python: k (scalar), ω (scalar), ν_t (scalar), ν (scalar), wall distance (scalar), u_i (vector), $\nabla_i k$ (vector), $\nabla_i p$ (vector), $\epsilon_{ijk}\nabla_i u_j$ (vector) and $\nabla_i u_j$ (tensor), resulting in $N_{scalars} = 26$. This array is converted from a C++ array to a NumPy array, such that it can be interpreted by

Python. The `model` function loaded at startup is called with this converted array as the first argument. The second argument is a dictionary of the following user specified booleans: useSigma, modelkDeficit, modelRST and modelSigma. Here, useSigma toggles the use of a classifier; modelkDeficit/modelRST/modelSigma toggles between the use of a $R/b_{ij}^{\Delta}/\sigma$ model or the exact frozen field, allowing for isolated model testing. Note that kDeficit in variable names refers to the R correction, while bijDelta/RST refers to the b_{ij}^{Δ} correction.

Within the `model` function, the SpaRTA infrastructure is called for the evaluation of the features. This is another advantage of the OpenFOAM-Python interface; features are only defined in one place, decreasing the likelihood of programming errors. Evaluating all ~ 60 features each iteration would result in a significant amount of wasted computational effort. This is because models typically contain less than 10 features and many features contain multiple expensive matrix-matrix products. Hence, features are stored in a custom data type named a lazy dictionary which is derived from the dictionary data type. The lazy dictionary initially only stores a function for each feature, containing the steps to calculate that feature. When accessed, the function is called to actually calculate the feature and the function is replaced with its evaluated result. With this format, only features which appear in the model are calculated; if they are present multiple times the calculated result is reused.

The `model` function evaluates the R/b_{ij}^{Δ} model if modelkDeficit/modelRST is true, it evaluates the σ model if useSigma and modelSigma are true. Results are returned in an array of size $(N_{cells} \times 8)$ (two scalars and one symmetric tensor results in 8 scalars). NaN fields are send back if no model was evaluated for a variable, as the returned result is not used anyways (OpenFOAM then reads in the appropriate fields). After OpenFOAM receives this array, it first checks whether it has the expected shape to prevent out-of-memory reading. Then, for each variable that is modeled, the appropriate column(s) are extracted from the array and converted to a C++ array. The corrections are multiplied with the classifier, the ramping function and usekDeficit/useRST before they are added to the equations of the k-ω SST turbulence model. The ramping function, usekDeficit and useRST are optional, user specified parameters further explained in the next section.

All code for the OpenFOAM-Python interface is contained within the custom turbulence model, so one would expect the default `simpleFoam` solver to suffice for running the case. However, using this solver results in an error at startup, which is also encountered by Maulik et al. [30]. They identify that the error originates from the inclusion of `setRootCaseLists.H` in `simpleFoam.C` and they devise a fix for it. This fix is implemented in a modified version of `simpleFoam.C` named `modelPropagationFoam.C`, it is listed in Appendix D.3.1. The same error is encountered when running the `postProcess` function of the solver, stemming from the inclusion of `setRootCase.H` in the `postProcess.H` file. A similar fix is implemented in a modified version of `postProcess.H` named `modifiedPostProcess.H`, it is listed in Appendix D.3.2. In `modelPropagationFoam.C`, this `modifiedPostProcess.H` file is included rather than `postProcess.H`. The custom `model-PropagationFoam` solver is then compiled from these modified files and other required solver files copied straight from `simpleFoam`.

4.1.3 Usage of the model propagation infrastructure

The model propagation requires use of both a custom solver and a custom turbulence model, they are called `modelPropagationFoam` and `modelPropagationkOmegaSST` respectively. Their associated files are given in Appendix D.3 and Appendix D.1 respectively. Compilation of the turbulence model requires linking to Python and NumPy, the procedure is further laid out by Maulik et al. [30]. Upon compilation, a number of old-style-cast warnings appear, these can be ignored. In order to perform model propagation for a case, the `RASModel` must be set to `modelPropagation-kOmegaSST` in the constant/turbulenceProperties file. A number of other variables can be defined in the turbulenceProperties file as well to modify the behaviour of the custom turbulence model. These variables are listed below along with their default value and a description of their effect. Note that kDeficit in variable names refers to the R correction, while bijDelta/RST refers to the b_{ij}^{Δ} correction.

- **modelkDeficit [true]:** Boolean to switch between using a predefined R field (false) or re-evaluate R each iteration using a model (true). If false, a file named 'kDeficit' should be present in the time directory the run starts from. If true, a file named 'kDeficitEq' should be present in the case directory, containing the model equation to calculate R. This variable is useful for isolated model testing; using the frozen R field (modelkDeficit false) in combination with a b_{ij}^{Δ} model yields the isolated performance of the bijDelta model.
- **modelRST [true]:** Same functionality as modelkDeficit, but for b_{ij}^{Δ}.
- **modelSigma [true]:** Same functionality as modelkDeficit, but for the classifier σ.
- **useSigma [false]:** Boolean to switch between using (true) or not using (false) a classifier. If no classifier is used, σ is 1 everywhere (corrections active in all cells).
- **usekDeficit [1.]:** Scalar factor by which the R correction is multiplied, should be in range $[0, 1]$. No R correction is used if usekDeficit is 0, while the full R correction is used if usekDeficit is 1. A case which is unstable with usekDeficit $= 1$ may stabilize when using for example usekDeficit $= 0.5$.
- **useRST [1.]:** Same functionality as usekDeficit, but for b_{ij}^{Δ}.
- **rampStartTime [-1.]** Scalar start time of the ramping function, see rampEndTime for a description of ramping.
- **rampEndTime [0.]** Scalar end time of the ramping function. The ramping function is defined as follows: before rampStartTime it is 0, then it linearly grows to 1 between rampStartTime and rampEndTime, after which it remains 1. The correction fields are multiplied by the ramping function such that they are slowly introduced to the case. This may stabilize otherwise unstable models/correction fields at the cost of an increased total number of iterations. By default, ramping is disabled by starting the run after rampEndTime.

As explained in the modelkDeficit description above, if modelkDeficit is true, a 'kDeficitEq' file should be present in the case directory. If it is set to false, a 'kDeficit' file with field data should be present in the time directory the run starts from. As an example, if modelkDeficit is false, modelRST is true and the case is started from 0, a 'bijDeltaEq' file should be present in the case directory and a 'kDeficit' file should be present in the 0 directory. Furthermore, the 'model_definition.py' Python script, given in Appendix D.2 and further described in Sec. 4.1.2, should be present in the case directory. Finally, the case has to be run and post-processed using the custom `modelPropagationFoam` solver, which is derived from the `simpleFoam` solver.

4.2 Non-linear symbolic regression framework (CuRTA)

This section covers the CuRTA symbolic regression framework, developed as part of this work. CuRTA stands for Curve-fit Regression of Turbulence stress Anisotropy; SpaRTA's Spa (sparse) is replaced with Cu. This is because CuRTA uses the non-linear least squares regression implemented in Scipy's `curve_fit` function. In Sec. 4.2.1, the problem statement is provided which motivates the development of this new framework. Then, the algorithm and further implementation details are given in Sec. 4.2.2. Finally, in Sec. 4.2.3, the inputs to CuRTA's main function and their effect on the program are laid out.

4.2.1 Problem statement

The goal of SpaRTA is to find simple models with small coefficients that match the target as well as possible. For simplicity, consider the regression of a model containing only one feature q. In this case, the match of the model with the data depends on two factors: the spread of the data at a given value q and the ability of the model's functional form to match the shape of the data. The first factor is addressed in SpaRTA by using a large number of input features in the library and selecting the best one. The second factor is addressed by also including various functions and exponents of features in the library.

Just adding functions of features to the library, for example tanh, does not increase the available functional forms as much as one may expect. This is because SpaRTA performs a linear regression, meaning it can only regress coefficients outside the function, giving the following regression form:

$$y = C \tanh(q) + C. \tag{52}$$

Even this form is is usually not regressed by SpaRTA as the constant term (the second coefficient) rarely appears in models. A more general form of the tanh function is:

$$y = C \tanh(C \cdot q + C) + C, \tag{53}$$

where two additional coefficients are present inside. Although it is impossible to linearly regress these coefficients, an informed guess can be made. To this end, all features in SpaRTA are standardized by their standard deviation, giving the following regression form:

$$y = C \tanh(q/\mathrm{std}(q)) + C. \tag{54}$$

While this proves a good guess in practice, it can never be as accurate as a regression of the coefficient. Furthermore, the lack of the second coefficient inside the tanh also limits accuracy.

To further study the effect of the functional form, the following function is randomly sampled on the domain $0 \leq q \leq 1$ and Gaussian noise with $\sigma = 0.04$ is added:

$$y = -0.4\,(\tanh\,(5 \cdot q - 2) - 1.2)\,. \qquad (55)$$

This sampled data is shown in Fig. 3, together with the SpaRTA fit of Eq. 54 and a non-linear fit of Eq. 53 (CuRTA fit). Clearly, SpaRTA's functional form is much too limited to properly regress this data, while non-linear regression performs optimally. The latter is not surprising, considering the data is generated directly from the non-linear functional form. Nonetheless, a similar shape is observed for the q_v feature and the data fit is significantly improved with non-linear regression, as is laid out in Sec. 12.2.3. This proves that SpaRTA's functional form indeed prevents it from attaining the optimal fit. As explained in Sec. 12.2.2, this bad fit results in unsatisfactory SpaRTA b_{ij}^{Δ} models when tested *a-posteriori*. Hence, a new framework is needed that is able to train models based on non-linear regression, this is the focus of this section.

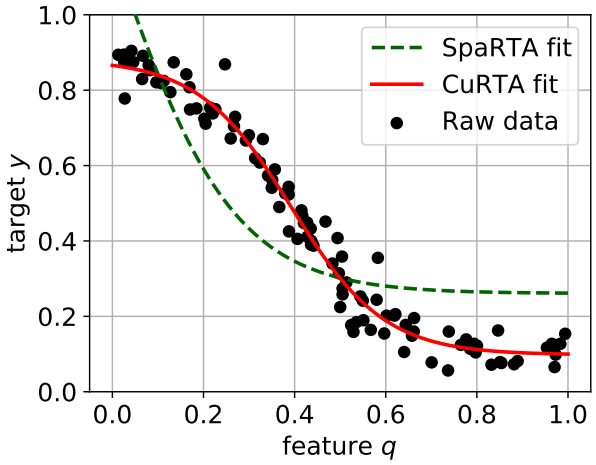


Figure 3: Comparison of data regression between SpaRTA (Eq. 54) and non-linear CuRTA (Eq. 53). Data comes from randomly sampling Eq. 55 on $0 \leq q \leq 1$ and adding Gaussian noise with $\sigma = 0.04$.

4.2.2 Implementation details

The flow of CuRTA's Python program given in Appendix E is laid out next, starting at the call of the main `regressModel` function. For a description of the inputs to this function, the reader is referred to Sec. 4.2.3. A simplified version of CuRTA's algorithm is given in Algorithm 3, where no filtering, bounds, initial guess and targetFunc are present. The implementation of this algorithm into Python as well as the inclusion of these additional effects is elaborated next.

Algorithm 3 Simplified CuRTA algorithm (no filtering, bounds, initial guess and targetFunc).

$\Psi = \texttt{setupFlow}(\text{casePath})$ ▷ Calculate features and bases for training case
$\mathcal{T}_{func,i} = f_{i_1}(q_{i_2}) \cdot \ldots \cdot f_{i_{2gd-1}}(q_{i_{2gd}})$ ▷ Products of global degree (gd) functions of features q
$\mathcal{T} = \mathcal{T}_{func,i}\, b_j$ ▷ Term library, combinations of scalar functions and bases b, indices omitted
$\Delta = y$ ▷ Initialize deficit Δ as the target y
for $i = 0$ to N_{terms} **do** ▷ Loop until the expression has N_{terms} terms
    $\mathcal{T}_{fit} = \texttt{nonLinearLSQ}(\mathcal{T}, \Delta, w)$ ▷ Fit each term in $\mathcal{T}$ to match Δ, using weights w
    $t_{opt,i} = \texttt{min}(\texttt{RMSE}(\mathcal{T}_{fit}))$ ▷ Find optimal term (lowest RSME)
    $M = t_{opt,0} + \ldots + t_{opt,i}$ ▷ Full model is the sum of all optimal terms fitted so far
    $M_{refit} = \texttt{nonLinearLSQ}(M, y, w)$ ▷ Refit coefficients of full model
    $\Delta = y - M_{refit}(\Psi)$ ▷ Update deficit to target-model deficit

First, all variables in the latest time directory of the specified case directory are read in and bases and features are calculated. This is done using the already established sparta.features library. Next, cells are filtered based on allInds followed by random selection of NAll cells. A new dictionary is created of all features/bases/other variables at the cells remaining after these two filtering steps. Then, further filtering is performed by randomly selecting NTrain cells. A second dictionary is created of all features/bases/other variables at these further filtered cells.

Next, the functions are loaded in, a list of implemented functions is provided in Tab. 4 along with their equation. Each function depends on a feature q and most functions also contain one or more unknown coefficients C. These coefficients C are later regressed using non-linear least squares. The non-linear least squares algorithm (discussed in a bit) is sensitive to the initial guess of these coefficients. The default initial guess of 1 is prone to blowup, especially for exponential functions combined with large features. A better initial guess requires knowledge of the characteristics of the feature, so various statistical functions are used for the initial guesses in Tab. 4: The standard deviation of q (std(q)), the mean of q (mean(q)) and the sign of q (sign(q)). Some coefficients also require bounds to prevent blow up, these are again based on feature statistics, they are also given in Tab. 4. Not all coefficients require bounds, in this case they are set to $\pm\infty$. Functions are stored in a custom class which calculates the initial guess and bounds once a function is combined with a feature.

Table 4: Function names, their equation with feature q and coefficients to be regressed C, their bounds and their initial guess.

Name	**Equation**	**Lower bound**	**Upper bound**	**Initial guess**
linear	$y = q$	-	-	-
tanh	$y = \tanh(C \cdot q + C) + C$	$(-\infty, -\infty, -\infty)$	(∞, ∞, ∞)	$\left(\frac{1}{50\text{std}(q)}, \text{std}(q), \text{std}(q)\right)$
std	$y = \exp\left(-C(x - C)^2\right)$	$(0, -\infty)$	$\left(\frac{10}{(\text{std}(q))^2}, \infty\right)$	$\left(\frac{1}{2(\text{std}(q))^2}, \text{mean}(q)\right)$
rlog	$y = \ln(C\lvert q\rvert + 1)$	0	$\frac{10}{\text{std}(q)}$	$\frac{1}{\text{std}(q)}$
rdiv	$y = \frac{q}{C \cdot q^2 + 1}$	$-\frac{10}{\text{std}(q)}$	$\frac{10}{\text{std}(q)}$	$\frac{\text{sign}(\text{mean}(q))}{\text{std}(q)}$
sqrtabs	$y = \sqrt{\lvert q\rvert}$	-	-	-
rdivsqrt	$y = \frac{q}{C\lvert q\rvert^{3/2} + 1}$	$-\frac{10}{\text{std}(q)}$	$\frac{10}{\text{std}(q)}$	$\frac{\text{sign}(\text{mean}(q))}{\text{std}(q)}$
rdivquart	$y = \frac{q}{C\lvert q\rvert^{5/4} + 1}$	$-\frac{10}{\text{std}(q)}$	$\frac{10}{\text{std}(q)}$	$\frac{\text{sign}(\text{mean}(q))}{\text{std}(q)}$
pow	$y = \lvert q\rvert^C$	1.01	4	1.2

A target function is used to reshape tensorial models and targets to 1D. The default 'all' target function simply adds a .flatten() to the model equation and target. Three other target functions are implemented that calculate a scalar from the components of the symmetric tensor, they are the three principle tensor invariants: 'I1' (corresponding to the trace), 'I2' and 'I3' (corresponding to the determinant). The equations of these invariants are as follows:

$$I_1 = A_{11} + A_{22} + A_{33}, \tag{56}$$

$$I_2 = A_{11}A_{22} + A_{11}A_{33} + A_{22}A_{33} - A_{12}^2 - A_{13}^2 - A_{23}^2, \tag{57}$$

$$I_3 = A_{11}A_{22}A_{33} + 2A_{12}A_{13}A_{23} - A_{11}A_{23}^2 - A_{22}A_{13}^2 - A_{33}A_{12}^2, \tag{58}$$

where A is an arbitrary symmetric tensor. Next, the specified weight expression is evaluated for each mesh cell to attain the weights. Special care is taken when the 'all' target function is used to correctly copy the weights to all six components.

The library of terms is constructed in three steps: First a subsub-library is made by combining each specified feature with each specified function. To allow lower order models, the const variable (constant value of one) is added to this subsub-library. Then, the outer product of this subsub-library is taken with itself global_degree times, giving a sub-library where each term contains global_degree functions of features (or const). In case of global_degree zero, this sub-library contains a single empty string. Finally, the outer product of the sub-library is taken with the specified bases to generate the full library. This is also where the leading unknown coefficient of each term is added. If global_degree is zero, the full library just has bases multiplied by this unknown coefficient. Even if global_degree is higher (for example two), terms such as $C \cdot \text{const} \cdot \text{const} \cdot \text{basis}$ are in the library to include models of lower global_degree.

The main iteration starts with regression of the model's first term; coefficients of each term in the library are regressed using NTrain points to match the target. The term that gives the highest R^2 is selected and its coefficients are refit using NAll points. Then, the difference between this term's prediction and the actual target is calculated to find the prediction deficit. For regression of the model's second term, coefficients of each term in the library are again regressed using NTrain points, but now to match this deficit. The highest R^2 term is now added to the first term to form a two-term model. All coefficients of this model are refit using NAll points and the prediction deficit of the model is found, serving as the target for the third term. This process is continued until the model has NTerms terms.

Non-linear regression is performed using SciPy's `optimize.curve_fit` function[5], which uses a different optimization algorithm depending on the presence of finite bounds. The Levenberg-Marquardt (lm) algorithm is used for unbounded problems, while the Trust Region Reflective algorithm (trf) is used for bounded problems. This default behaviour is maintained, such that the significantly faster lm algorithm is used for unbounded terms. Next, weights cannot be directly specified for this function, only the data uncertainty σ. The two are related, as a point with more uncertainty weighs less in the non-linear least squares regression, they are related as:

$$\sigma = 1/\sqrt{\text{weights}}$$

[5] SciPy community (2023). *scipy.optimize.curve_fit*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html

Finally, some limitations and possible improvements of the CuRTA framework are listed. Firstly, the deficit fitting only seems to work for models with just a few terms. After these first terms, the deficit seems to be mostly noise in the training data, resulting in nonsensical extra terms. Secondly, the functional forms in Tab. 4 are still somewhat limited; adding more functional forms potentially improves data fits. Thirdly, a number of optimizations that are in SpaRTA have potential to speed up CuRTA. For example, removing features and bases with small variance. Also the cliqueing procedure in SpaRTA is promising, though it cannot be directly adapted due to CuRTA's nonlinear coefficients.

4.2.3 Usage of CuRTA

Symbolic regression using the CuRTA library is performed by calling its `regressModel` function, which has two required arguments and eleven optional arguments. These arguments are listed next, along with their accepted input forms and their effect on the program. The default value of optional arguments is given in square brackets.

- **targetVar:** The name of the target variable to find a model for, either 'kDeficit' to find a model for R or 'bijDelta' to find a model for b_{ij}^{Δ}.
- **casePath:** Path (string) to the case directory of the case on which to train a model. This can be either the frozen case or the propagation case, in this work it is argued that propagation should be used for training (see introduction to Sec. 12). Within this case directory, the last time directory should contain all variables needed to calculate the features, bases and weights. For example, gradU is needed for most bases and V is needed to apply volume weighing.
- **global_degree [1]:** Number of (functions of) features to multiply per term, also possible to specify zero to only use a linear combination of bases. Examples of terms at various global degrees are given below, where each C indicates an unknown coefficient to be regressed:
  - $0: C \cdot T_{ij}^{(2)}$
  - $1: C \cdot \sqrt{|S_2|} \cdot T_{ij}^{(5)}$
  - $2: C \cdot (\tanh(C \cdot q_\nu + C) + C) \cdot |S_3|^C \cdot T_{ij}^{(1)}$
  - $3: C \cdot q_\gamma \cdot \exp\left(-C\left(q_Q - C\right)^2\right) \cdot \ln(C\,|W_2| + 1) \cdot T_{ij}^{(9)}$

  Note that terms with a lower global degree than the one specified are also considered in the symbolic regression.
- **NTerms [None]:** Total number of terms to symbolically regress. For example, setting global_degree to zero and NTerms to 3 would regress a b_{ij}^{Δ} model such as $b_{ij}^{\Delta} = 1.4 \cdot T_{ij}^{(2)} + 0.3 \cdot T_{ij}^{(3)} + 0.5 \cdot T_{ij}^{(6)}$. Note that the program adds terms one-by-one, meaning that the optimal 1-term model is always printed first, followed by the optimal 2-term model and so on. If NTerms is set to None, it is internally set to infinity such that the program keeps adding terms.
- **allInds [None]:** List/array of indices of cells that qualify for use in the symbolic regression. These indices should be unique and the maximum index should not exceed the number of mesh cells minus one (due to zero relative indexing in Python). Alternatively, allInds can be set to None to qualify all cells for use in the symbolic regression. Excluding some cells from qualifying is useful for boxing of certain mesh areas or excluding bad-quality cells. Finally, note that not all qualifying cells are used, see NAll/NTrain.

- **NAll [None]:** Number of cells to use for computationally cheap operations (field statistics and final model refits). If None is specified, NAll is set equal to the number of qualifying cells, so either the length of allInds or the number of mesh cells if allInds is None. If a number is specified for NAll, it should not exceed the number of qualifying cells. A subset of cells of size NAll is then randomly chosen from the qualifying cells to use for the aforementioned computationally cheap operations. Note that identification of the best term from the library takes the most computational effort; this process uses NTrain cells (see next item). Hence, NAll should only be used if memory problems are encountered or when using an excessively large mesh.
- **NTrain [None]:** Number of cells to use for the identification of the best term from the library. This process is the most computationally expensive, as it does a non-linear least squares fit for each term in the library. If NTrain is set to None, NAll cells are used in this process. If NTrain is specified as a number, a second subset of cells of size NTrain is randomly selected from the subset of NAll cells. Use of NTrain can significantly speed up computations, however, one must ensure that the selected cells are still an accurate representation of all qualifying cells.
- **funcs [None]:** List of function names to use, for example ['linear', 'tanh', 'pow']. A complete overview of function names is given in Tab. 4, together with their equation, bounds and initial guess. The use of initial guesses and bounds is needed for stability, as discussed in Sec. 4.2.2. Each library term consists of a coefficient, global_degree functions of features and a basis. If funcs is specified as None, all functions in Tab. 4 are used.
- **bases [None]:** List of bases to use, for example ['T1', 'T3', 'T9']. These bases must correspond to the specified targetVar. If bases is set to None, all bases corresponding to the targetVar are used, so ['epsilon', 'G1', ..., 'G10'] for 'kDeficit' and ['T1', ..., 'T10'] for 'bijDelta'.
- **fitFeatures [None]:** List of features to use, for example ['q_nu', 'S2', 'q_gamma']. All variables required to calculate these features must be present in the last time directory in casePath. If fitFeatures is set to None, all available features are used.
- **targetFuncName ['all']:** Name of the function to use to attain a 1-dimensional target, only relevant for b_{ij}^{Δ} model training. The default 'all' target function flattens both the tensor equation and the tensor target such that each component is fit, resulting in six times more fitting points than mesh cells. Target functions that instead calculate a scalar from the components of b_{ij}^{Δ} (such as taking its trace) are also possible. To this end, the three principal tensor invariants are each available as a target function ('I1', 'I2' and 'I3'), they are given in Eqs. 56-58.
- **weights [None]:** Expression of how to calculate the weights as a string, for example 'V*k' to weigh by both volume and k. Alternatively, if weights is set to None, each cell is weighed the same.
- **meanFlowTimeScale [False]:** Which timescale to use to nondimensionalize the strain rate and rotation rate tensors s_{ij} and ω_{ij} (see Sec. 2.6). If True, these tensors are nondimensionalized by the magnitude of the mean velocity gradient tensor. If False, they are nondimensionalized by the specific dissipation rate ω.

5 NASA challenge submission

A collaborative testing challenge is set up by NASA as part of their 2022 symposium on turbulence modeling. The direct motivation for this challenge is the fact that efforts in data-driven turbulence modeling have focused on training models tailored to one class of flows, while the wider community desires general models [45]. Hence, five flows of different classes are included in the challenge; a zero pressure gradient flat plate, a channel, an axisymmetric subsonic jet, a wall-mounted hump with separated flow and a NACA0012 airfoil at various angles of attack (including stall). Meshes, boundary conditions and results for various conventional RANS turbulence models are provided on the symposium website [42]. The goal of the challenge is to train a RANS turbulence model for all cases, giving improvements over conventional turbulence models where these fail while retaining performance where they work well.

Prominent groups in the field of data-driven turbulence modeling from all over the world participated in this challenge. One of the participants was the group of Dr. Richard Dwight, known for the introduction of the k-corrective-frozen approach and the SpaRTA framework for symbolic regression [46]. Their submission is based on these two as well as a classifier introduced in later work by Steiner et al. [54]. The present work is written under the same group and this section aims to present the efforts for the challenge submission of this group. The current author is only one of the contributors, the others are Richard Dwight, Renzhi Tian and Tyler Buchanan. A paper on the submission is currently being prepared, but is unpublished at the time of writing, the presentation given at the symposium is available though [13].

The section is structured as follows: In Sec. 5.1, the case setup of the axisymmetric jet and airfoil are presented; other cases merit their own separate sections. The setup includes a baseline k-ω SST run as well as a frozen run to find correction fields and a propagation run to validate them. Next, in Sec. 5.2 the creation of training data of the classifier is laid out, followed by symbolic regression of the classifier. Then, a correction model is trained in Sec. 5.3, followed by its propagation with the classifier to yield *a-posteriori* results for each case. Finally, conclusions are drawn in Sec. 5.4 followed by recommendations.

5.1 Case setup

Since meshes and boundary conditions are provided by NASA, the task at hand is to implement these in an OpenFOAM case and verify the results against NASA's RANS results. Furthermore, k-corrective-frozen is applied to cases with a significant RANS model error where high-fidelity field data is available. The resulting correction fields are then validated using the propagation approach. Three challenge cases are further used in the present work (the channel, plate and hump) and thus merit their own detailed section, these are Sec. 8, Sec. 9 and Sec. 10 respectively. The other two challenge cases, the jet and the airfoil, are not further used in this work. Hence, their setup is discussed briefly in this section, in Sec. 5.1.1 and Sec. 5.1.2 respectively.

5.1.1 Axisymmetric jet

The axisymmetric jet case consists of a high pressure chamber connected to lower pressure, non-moving air via a contracting, axisymmetric nozzle. As a result of the pressure difference, air exits the nozzle at high speed, forming an axisymmetric jet. The schematic of this axisymmetric nozzle is shown in Fig. 4, where the main flow is in the $+x$-direction. The exit diameter of the nozzle, D_{jet}, is set to the experiment value of 0.0254 m. High fidelity data is available in the form of particle image velocimetry (PIV) measurements by Bridges and Wernet [6]. The exit Mach number is 0.51, meaning that while the case is subsonic, it cannot be considered incompressible.

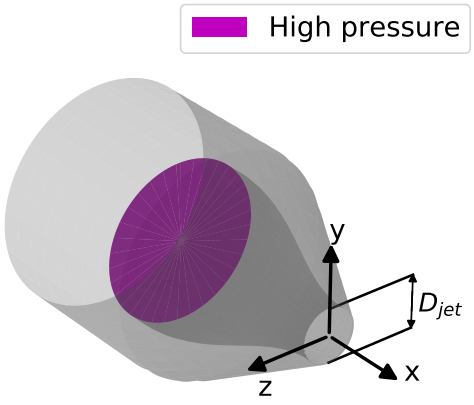


Figure 4: Schematic of the axisymmetric jet case, main flow in $+x$-direction.

For meshing, the axisymmetric nature of the case can be exploited to reduce the number of cells; only a small wedge needs to be meshed with a tangential thickness of one cell. On the symposium site, NASA already provides such a mesh that is verified to be mesh and domain size independent (their 97×97; 61×97; 257×225 mesh) [42]. This mesh is converted from plot3d to OpenFOAM format using `plot3dToFoam`, after which patches are assigned using `autoPatch` and `createPatch`. The compressible, transient solver `rhoPimpleFoam` is used, as the flow exits at Mach 0.51 and the case only converges to quasi-steady-state (this was also observed by NASA). After reaching quasi-steady-state, a few additional periods are run which are averaged to attain final results (also accounting for sampling bias). Boundary conditions are adapted from NASA and initial conditions and relaxation factors are chosen to give stable yet fast convergence.

Profiles of the baseline k-ω SST run on NASA's mesh (CFD domain) are shown in Fig. 5 together with the PIV data. The x-velocity is predicted well just behind the nozzle, but is underpredicted further downstream (especially near the centerline). This can be explained by considering profiles of k; a large k is introduced at the nozzle wall which spreads throughout the jet further downstream. The spreading rate of k is significantly underpredicted by k-ω SST, leading to excessive dissipation, resulting in the underprediction of x-velocity downstream. Given this mismatch, correction terms are found for the jet case using the k-corrective-frozen approach outlined in Sec. 3.3.3. This approach requires knowing high-fidelity velocity and Reynolds stress throughout the CFD domain. However, the PIV data is only available in the wake of the jet, close to the centerline, meaning a CFD case has to be set up of just this PIV portion.

A new wedge shaped mesh is made of this PIV portion, where the cell sizes are approximately matched with the NASA mesh. An incompressible, steady-state case is set up for this mesh, even though the jet is neither. This is necessary as the k-corrective-frozen infrastructure is only implemented for incompressible, steady-state cases, the resulting error is discussed later. The boundary conditions are based on the PIV data, such that discrepancies come only from the model being solved in the domain. For the inflow, the velocity and k are taken directly from the PIV, while ω is calculated based on the assumption that production of k equals dissipation of k. At the outlet, the static pressure is set to the freestream pressure. The profiles resulting from this PIV domain

run are shown in Fig. 5, where a significant discrepancy is visible with respect to the CFD domain. Likely causes of the mismatch are lack of compressibility modeling and a too small domain, however, no resources are available to address either, so the current setup will have to make do.

The k-corrective-frozen approach takes the high-fidelity data in each cell as an input and calculates first and second derivatives from this. Simple interpolation of the noisy PIV data would result in large, fluctuating derivatives. To address this, Scipy's `SmoothBivariateSpline` function is used to regress a smooth, second order bivariate spline of each variable, which is then sampled at each cell center. Next, the correction fields R and b_{ij}^{Δ} are found using the k-corrective-frozen approach described in Sec. 3.3.3. These fields are then propagated in a full RANS solver, the resulting propagation profiles are shown in Fig. 5. Significant improvements are observed for the U_x profiles, while only slight improvements are observed for k. This is contrary to other cases, where usually an almost exact match is observed between high-fidelity and propagation for U and k [46]. Likely the aforementioned lack of compressibility modeling and small domain spoil the propagation performance of the current case.

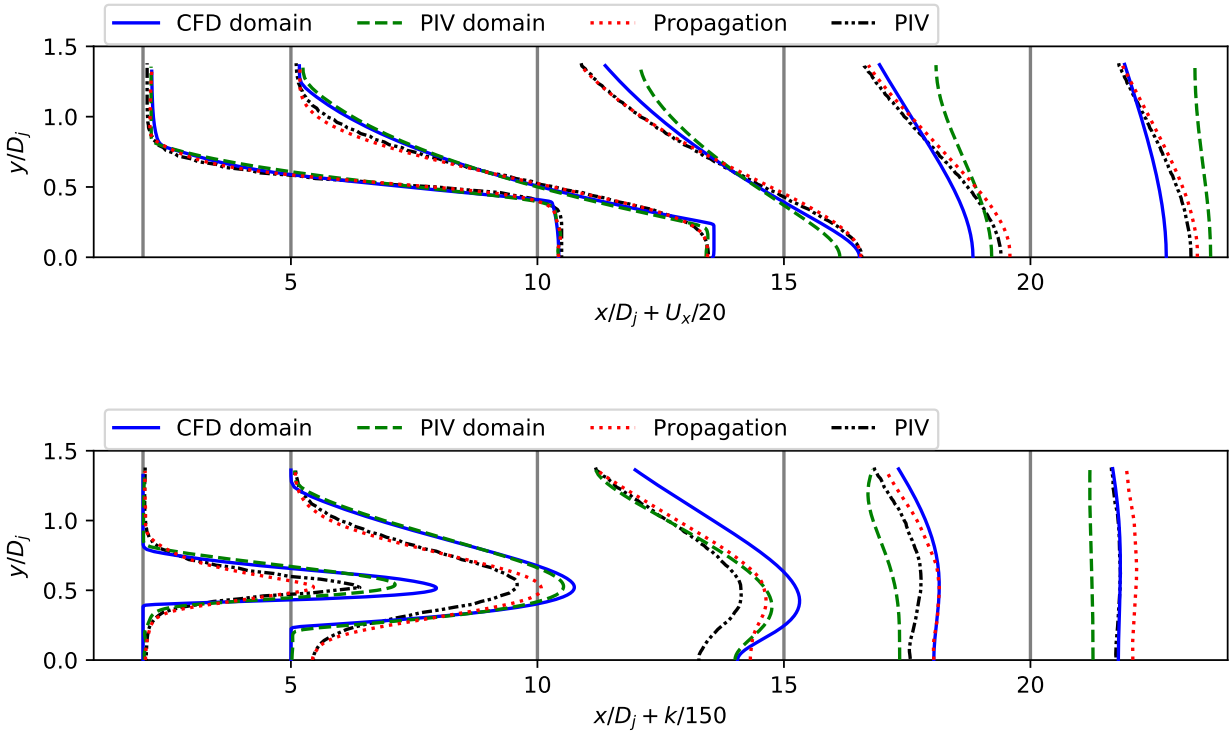


Figure 5: Profiles at various x-stations of the axisymmetric jet, comparing baseline k-ω SST in the large CFD domain and the reduced PIV domain, propagation in the reduced PIV domain and PIV measurements by Bridges and Wernet [6].

5.1.2 NACA0012 airfoil

The NACA0012 airfoil is an iconic airfoil that has been subject to a plethora of studies. Many of these focus on the case of 2D flow over this airfoil, which is also the focus of the present study. The schematic of the 2D NACA0012 case is shown in Fig. 6, where the main flow is in the x-z-plane. The coordinate system is fixed to the airfoil, so the angle of attack α is varied by changing the angle of the incoming freestream velocity U_∞. The airfoil's length in x-direction is its chord length c, while it has an infinite span (length in y-direction). This infinite span is only theoretical to indicate 2D flow with no flow/gradients in y-direction. Next, the case actually consists of multiple CFD runs, each at a different angle of attack. In particular, angles of attack around stall are studied as stall is associated with many complex, hard to model flow phenomena.

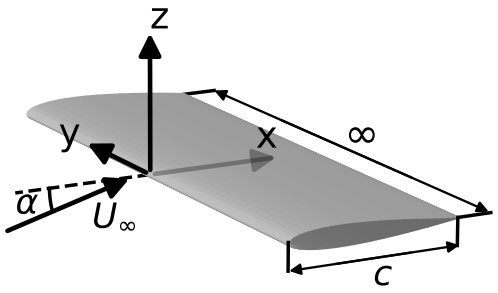


Figure 6: Schematic of the NACA0012 airfoil, main flow in the x-z-plane (direction dependent on α).

Since the case is 2D, the mesh only needs single cell thickness in y-direction, NASA provides such meshes at various levels of refinement. While NASA uses their second to finest mesh, this proves too difficult to converge and comes with extremely long run times, so their third to finest mesh is used here. NASA mentions that no mesh independence study was performed, the validity of using the third to finest mesh is evaluated in a bit. The mesh is converted from plot3d format to OpenFOAM using `plot3dToFoam`, after which patches are assigned using `autoPatch` and `createPatch`. Next, NASA mentions that the case is essentially incompressible, as the freestream Mach number is 0.15. Initial efforts with an incompressible solver gave large discrepancies with respect to NASA's RANS results, these are greatly reduced when using a compressible solver. Upon further investigation, Mach numbers of 0.6 are found near the leading edge, so a compressible solver is indeed necessary. The `rhoSimpleFoam` solver is used, boundary conditions are adapted from NASA and initial conditions and relaxation factors are chosen to give stable yet fast convergence.

The case is run at $\alpha = 10°, 15°, 17°, 18°$, the lift coefficient C_l and the drag coefficient C_d are calculated for each α using the following equations:

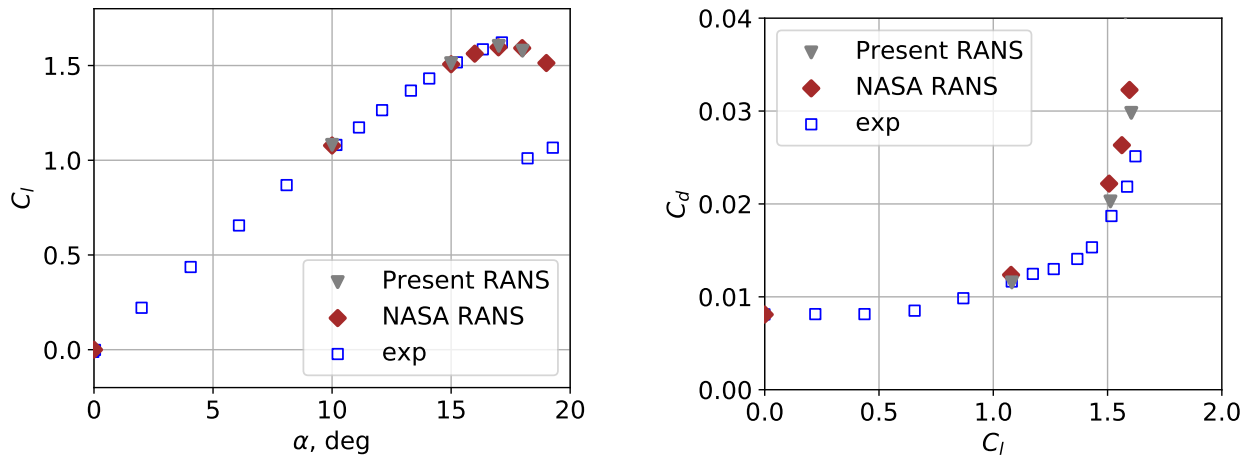
$$C_l = \frac{2L'}{\rho U_\infty^2 c}, \tag{59}$$

$$C_d = \frac{2D'}{\rho U_\infty^2 c}. \tag{60}$$

Here L' and D' are the total lift and drag force per unit span respectively, which are found from the total x and z forces outputted by OpenFOAM. The resulting C_l-α and C_l-C_d curves are shown in Fig. 7, where NASA's RANS data and experimental data by Ladson [25] are added for comparison. For the C_l-α curve, the present RANS matches the NASA RANS extremely well. The experiment is matched well up till $\alpha = 17°$, however, the large drop in C_l at $\alpha = 18°$ is not predicted by either RANS. Nonetheless, both correctly predict the stall angle and the magnitude of $C_{l,max}$. For the C_l-C_d curve, there is a ∼10% difference between the present RANS and the NASA RANS. Also there is a larger mismatch between RANS and experiment even at $\alpha \leq 17°$. At $\alpha = 18°$, the experimental C_d is so much larger than the RANS C_d that it falls outside the plot range (it is ∼0.2).

For further analysis, the pressure coefficient C_p and the skin friction coefficient C_f, further explained in Sec. 10.3.2, are plotted along the airfoil surface. Though an analysis at $\alpha = 18°$ would be most informative, NASA RANS and experimental data are only available at at most $\alpha = 15°$. The surface plots are shown in Fig. 8, where the NASA RANS data has been added for C_p and C_f and experimental data by Gregory and O'Reilly [19] only for C_p. An excellent match is found for both RANS runs and the experiment in terms of C_p, which is not surprising as C_p largely determines C_l. A good match is found between the RANS runs for C_f, though a small discrepancy is found at $x/c \approx 0.1$, likely leading to the small difference in C_d.

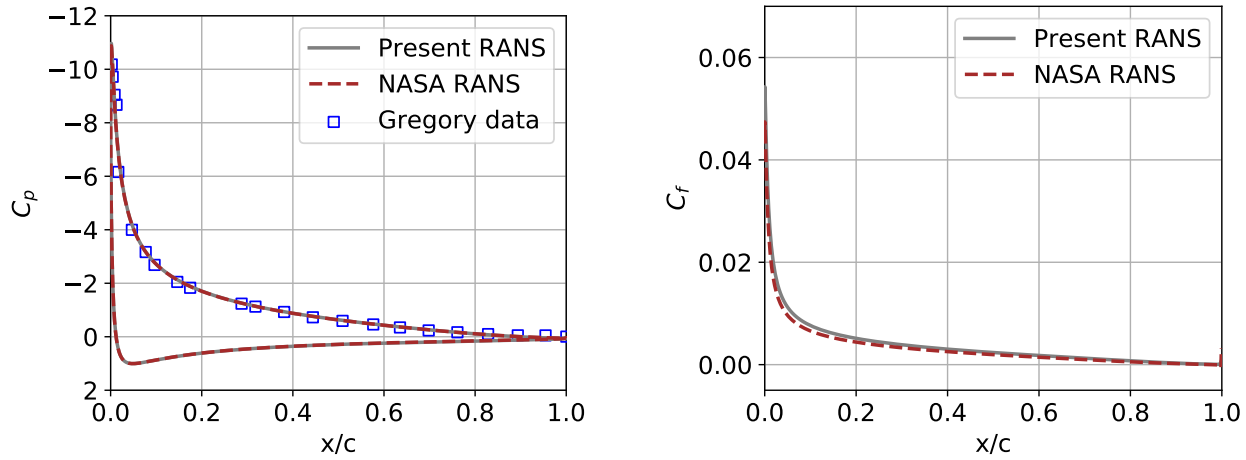
A large discrepancy exists between experimental data and RANS at $\alpha \geq 18°$, making the $\alpha = 18°$ case suitable for the k-corrective-frozen approach. However, this approach requires full-field high-fidelity data to be available, while the provided high-fidelity data only consists of integrated force coefficients. No high-fidelity full-field data could be acquired of this (or a similar) case, so training cannot occur on an airfoil. Hence, the airfoil will only serve as a test case, testing models trained on other geometries and comparing the resulting integrated force coefficients to the high-fidelity data. Finally, regarding the use of the coarse mesh; differences between the present RANS and the NASA RANS were much smaller than differences with respect to experiment at $\alpha = 18°$. Furthermore, since the case is not used for training, discretization errors are not as much of a problem, so use of the coarse grid is deemed acceptable.



(a) C_l vs α

(b) C_d vs C_l

Figure 7: Lift and drag coefficient curves at various angles of attack, comparing the present RANS, NASA's RANS and experiment by Ladson [25].



(a) C_p vs x/c (upper and lower surface)

(b) C_f vs x/c (upper surface)

Figure 8: Pressure and skin friction coefficient over the airfoil surface at $\alpha = 15°$, comparing the present RANS, NASA's RANS and the experiment by Gregory and O'Reilly [19] for C_p.

5.2 Classifier training

The classifier is a boolean function of space and time, both correction terms (R and b_{ij}^{Δ}) are multiplied by it. Thus, no corrections are applied if the classifier is deactivated (zero), while full corrections are applied when it is active (one). The goal is to train a symbolic expression for the classifier which takes local RANS flow variables as inputs and outputs the local activation status [54]. In order to train the classifier, training data has to be generated first, several approaches are possible. The training data generation approach used in the present work is laid out in Sec. 5.2.1. Next, the training procedure used to regress a symbolic expression based on this training data is explained in Sec. 5.2.2.

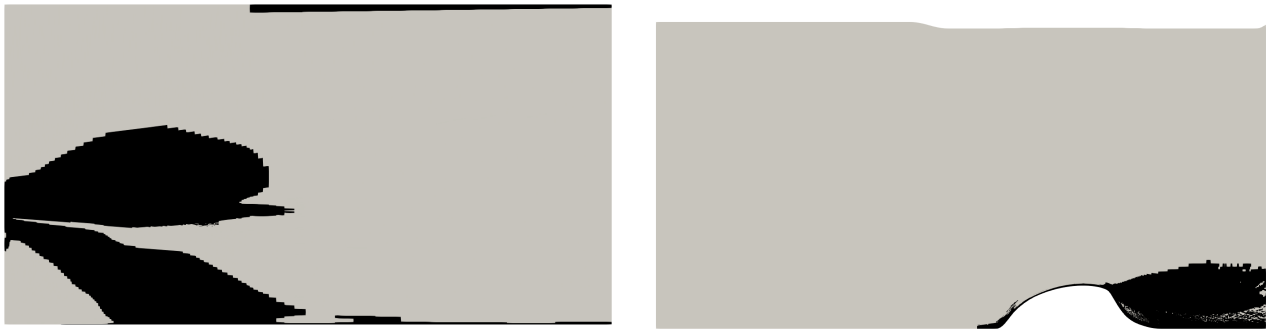
5.2.1 Generating training data

Training data for the classifier model is based on four out of the five cases. For the plate and the channel, k-ω SST already matches the high-fidelity well, so the classifier should be zero everywhere to retain this performance. For the jet and hump, there are regions where the classifier should be one to provide a correction to k-ω SST. However, there are also regions where the corrections are small and not relevant to the case (such as the far field) where the classifier should ideally be zero. Finally, the airfoil case will also have regions where the classifier should be active and regions where it should be inactive. However, no correction fields are available due to a lack of high-fidelity data, meaning no training data can be generated, so the airfoil is excluded from the classifier training.

For the jet and hump, the classifier training criterion is derived from the criterion used by Steiner et al., given in Eq. 43 [54]. Direct implementation of their criterion results in too much classifier activation. This is addressed by changing the denominators, the new criterion is as follows:

$$\sigma := \begin{cases} 1 \text{ if } \left(\frac{\left| 2k b_{ij}^{\Delta} \left(\partial \langle u_i \rangle / \partial x_j \right) \right|}{\overline{P_{k,LES}}} > 0.2 \right) \cup \left(\frac{|R|}{\overline{P_{k,LES}}} > 0.2 \right) \\ 0 \text{ otherwise,} \end{cases} \tag{61}$$

where $\overline{P_{k,LES}}$ is the domain averaged $P_{k,LES}$. Though this criterion gives better activation for now, it lacks theoretical justification. For example, expanding the farfield of a case should not affect classifier activation, but with this criterion $\overline{P_{k,LES}}$ would decrease, resulting in more activation. Furthermore, a domain with as much negative as positive $P_{k,LES}$ would have a mean of zero; it would be better to use $\overline{|P_{k,LES}|}$. Next, for the hump there is erroneous activation in the incoming boundary layer, so cells before $x = -0.05$ are manually forced to be inactive. The resulting classifier activation contours of the jet and hump are shown in Fig. 9. Though there are some small unexpected spots of classifier activation, most activation appears in areas of corrections as intended.



(a) Axisymmetric jet positive x-y-plane (contracted factor 8 in x), PIV domain.

(b) Hump x-z-plane (contracted factor 2 in x), LES domain.

Figure 9: Classifier activation training data for the jet and hump, black indicates $\sigma = 1$; grey indicates $\sigma = 0$.

5.2.2 Classifier training

Training of the classifier largely follows the procedure outlined by Steiner et al. [54], it is briefly laid out next. The classifier is fitted as a sigmoid function, which has a negative asymptote at 0 and a positive asymptote at 1, this behaviour is desirable for the classifier. The classifier is made boolean by setting any values above 0.2 to 1 and any below to 0, the formal mathematical expression is:

$$\sigma(\theta) = \begin{cases} 1 \text{ if } \frac{1}{1+\exp(-f(\theta))} > 0.2 \\ 0 \text{ otherwise.} \end{cases} \tag{62}$$

Here, θ represents a number of flow features which are largely adapted from Steiner et al., though some such as actuator forcing are left out. The function $f(\theta)$ in Eq. 62 represents an arbitrary function of these features; this function is symbolically regressed to fit the classifier training data. The SpaRTA framework laid out in Sec. 3.4 is used to regress a sparse symbolic expression for $f(\theta)$. Since σ is a dimensionless scalar, no bases are used, the output is simply a function of

the features. Training on all four cases with classifier training data does not produce well fitting classifier models. Hence, the jet is excluded from the training data as its correction fields come with many uncertainties (see Sec. 5.1.1). The simplest acceptable model regressed based on the three remaining cases (plate, channel and hump) is still rather complex with twelve terms, it is as follows:

$$\begin{aligned} f(\theta) = 0.02941 &+ 24.07\text{rdiv}\left(\frac{W^2}{2.964}\right) - 3.815\text{rdiv}\left(\frac{q_{pS}}{0.1333}\right) - \dots \\ &- 0.7596\text{rdiv}\left(\sqrt{\frac{q_{pS}}{0.1333}}\right) - 2.869\text{rdiv}\left(\frac{q_\gamma}{1.847}\right) - \dots \\ &- 0.02062\tanh\left(\frac{q_\gamma}{1.847}\right) - 0.935\text{rdiv}\left(\left[\frac{q_\nu}{92.16}\right]^2\right) - \dots \\ &- 0.9397\tanh\left(\left[\frac{q_\nu}{92.16}\right]^2\right) + 3.541\text{rdiv}\left(\sqrt{\frac{q_\nu}{92.16}}\right) + \dots \\ &+ 0.1161\text{rdiv}\left(\left[\frac{q_{Re}}{0.5425}\right]^2\right) + 26.34\text{rdiv}\left(\sqrt{\frac{q_{TI}}{156.1}}\right) - \dots \\ &- 1.995\tanh\left(\left[\frac{q_{\tau k}}{0.8177}\right]^2\right) \end{aligned} \tag{63}$$

where

$$\text{rdiv}(q) = \frac{q}{1+q^2}. \tag{64}$$

In fact, Steiner et al.'s best classifier model only had two terms, so the current model is likely overfitting the data. Luckily, the boolean nature of the classifier makes this overfitting less of a problem, as it cannot 'blow up' for unseen cases.

The trained classifier expression is now applied to the plate, channel, jet and hump to assess how well the training data is matched. The plate and channel training data (no activation anywhere) is matched perfectly by the trained classifier. For the jet and hump, the trained classifier is evaluated on the high-fidelity data to generate *a-priori* classifier activation contours, they are shown in Fig. 10. Comparing the jet activation to its training activation in Fig. 9a, the activation bubble slightly further from the centerline is not predicted. Furthermore, erroneous activation occurs in the top left region, which consists mostly of unperturbed freestream flow. Next, comparing the hump activation to its training activation in Fig. 9b, a good match is found. In fact, the regressed classifier produces a smoother activation region, which will likely give better convergence and a smoother *a-posteriori* flow solution. Finally, it should be noted that when this classifier is implemented in a full RANS solver, the features θ update each iteration, leading to constantly updating activation contours. Hence, the *a-posteriori* activation contours are likely different from the *a-priori* ones shown here.

(a) Axisymmetric jet positive x-y-plane (contracted factor 8 in x), PIV domain.

(b) Hump x-z-plane (contracted factor 2 in x), LES domain

Figure 10: Classifier activation based on Eq. 62 with regressed $f(\theta)$ given in Eq. 63, for the jet and hump, black indicates $\sigma = 1$; grey indicates $\sigma = 0$.

5.3 Model training

The next step is to find symbolic expressions for the corrections R and b_{ij}^{Δ}, which is also done with the SpaRTA framework described in Sec. 3.4. Training takes place only on cells in which the classifier is active, directly excluding the plate and the channel from the training. Furthermore, following the aforementioned uncertainties in the correction fields of the jet, it is excluded from the training data as well, leaving only the hump. Encouragingly, a simple model with a high coefficient of determination ($R^2 = 0.98$) is found for R:

$$R = 0.079\epsilon. \tag{65}$$

For b_{ij}^{Δ}, most models do not have a much higher R^2 than the R^2 of $b_{ij}^{\Delta} = 0$, which is also a valid model of course. In the end $b_{ij}^{\Delta} = 0$ is actually chosen, as *a-posteriori* performance of nonzero models is not any better, they only reduce solver stability. For later reference, the zero b_{ij}^{Δ} is formalized here:

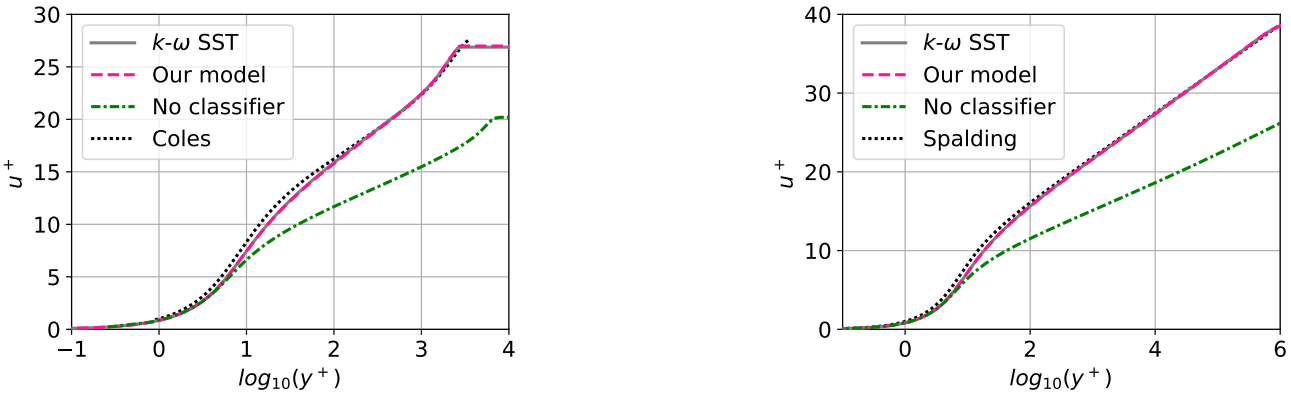
$$b_{ij}^{\Delta} = 0. \tag{66}$$

In order to test a model *a-posteriori*, it needs to be integrated in a RANS solver, such that it can be reevaluated with the latest flow fields and feed correction fields back to the solver. Unfortunately, the modelPropagationFoam solver laid out in Sec. 4.1, which does exactly that, did not exist at the time of the challenge. Hence, a workaround approach was devised using the existing infrastructure for propagation of cases. Specifically, the solver is stopped and restarted repeatedly, writing its fields when it is stopped and starting from these when restarted. Before the solver is restarted, a Python script is called to reevaluate the classifier and correction fields, updating the ones in the latest write. Since restarting comes with significant overhead, the solver is run 100 iterations between restarts. After a number of restarts, the solution has reached a constant value (that is to say, it has converged) at which point the solver is no longer restarted and the run reaches its end.

Using the restarting approach, the regressed model (R from Eq. 65 and b_{ij}^{Δ} from Eq. 66) is propagated under the classifier (defined by Eq. 62 and Eq. 63) to give *a-posteriori* results for each case. Since high-fidelity data is no longer required for the model propagation, all cases are run on their NASA mesh. Furthermore, any solver can be used as model propagation only affects the turbulence model, so a compressible solver is used for the jet and airfoil. Also, an unsteady solver is

used for the jet and final results are averaged. For the plate and channel, boundary layer profiles of u^+ vs y^+ (both defined in Sec. 8.2) are shown in Fig. 11a and Fig. 11b respectively, where baseline k-ω SST and theoretical results are added for comparison. Furthermore, an additional run without classifier is performed for these cases; its results are added as well. The model with the classifier perfectly matches k-ω SST for both cases, which is due to the classifier being 0 in both domains. The use of the classifier is necessary, as the no classifier results deviate significantly from baseline k-ω SST.

The *a-posteriori* results of the model on the hump are shown as profiles of x-velocity and k behind the hump in Fig. 12, where baseline k-ω SST results and LES results by Uzun et al. [56] are added for comparison. For the x-velocity profiles, PIV results by Greenblatt et al. [18] are also added. The regressed model performs extremely well in terms of x-velocity; it is in the range of error between LES and PIV. For k, no obvious improvement is observed, however, k in RANS can be different from k in LES and it is usually not of engineering interest. Now consider the model profiles of x-velocity and k of the jet in Fig. 13, where baseline k-ω SST and PIV profiles are added for comparison. A deterioration is observed with respect to baseline for both x-velocity and k, however, it is small compared to the difference with PIV. Finally, consider the model results for the airfoil shown in Fig. 14, where baseline k-ω SST and experiment are added for comparison. Before $\alpha = 17°$, predictions are good, but the model then predicts a much higher stall angle and $C_{l,max}$ than experiment, while baseline did predict these correctly. Prediction of C_d is similar to baseline (significant underprediction for $\alpha = 18°$).



(a) Plate

(b) Channel

Figure 11: Boundary layer profiles of the propagated regressed model, with and without classifier, together with baseline k-ω SST and theoretical results.

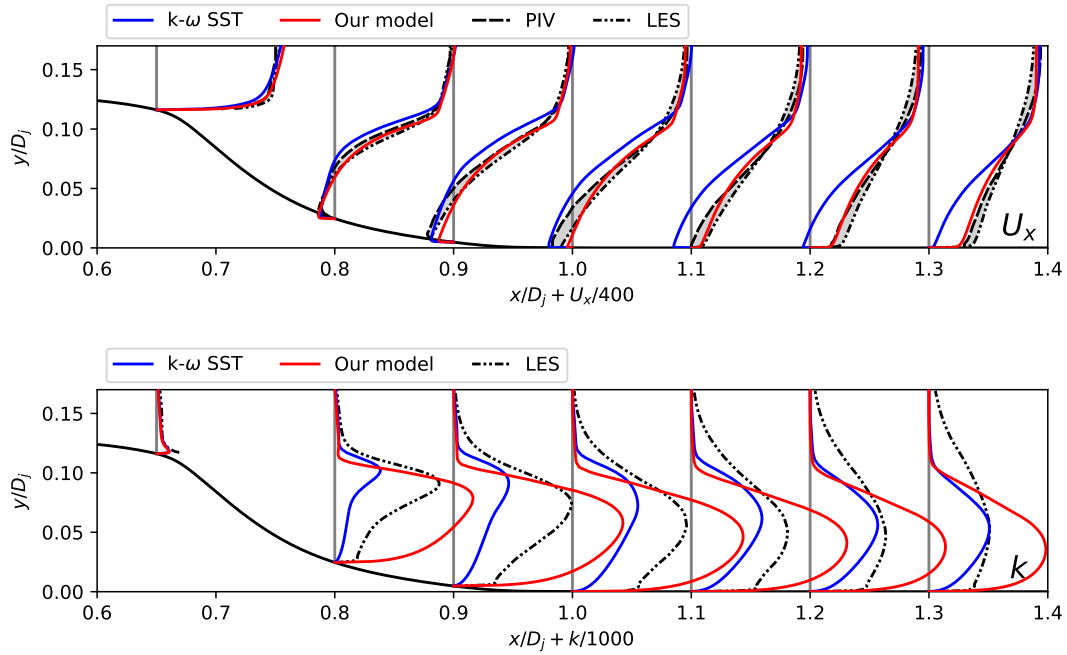


Figure 12: Profiles of x-velocity and k behind the hump, using the 817 point RANS domain mesh, comparing the propagated regressed model, baseline k-ω SST, LES results by Uzun et al. [56] and PIV results by Greenblatt et al. [18].

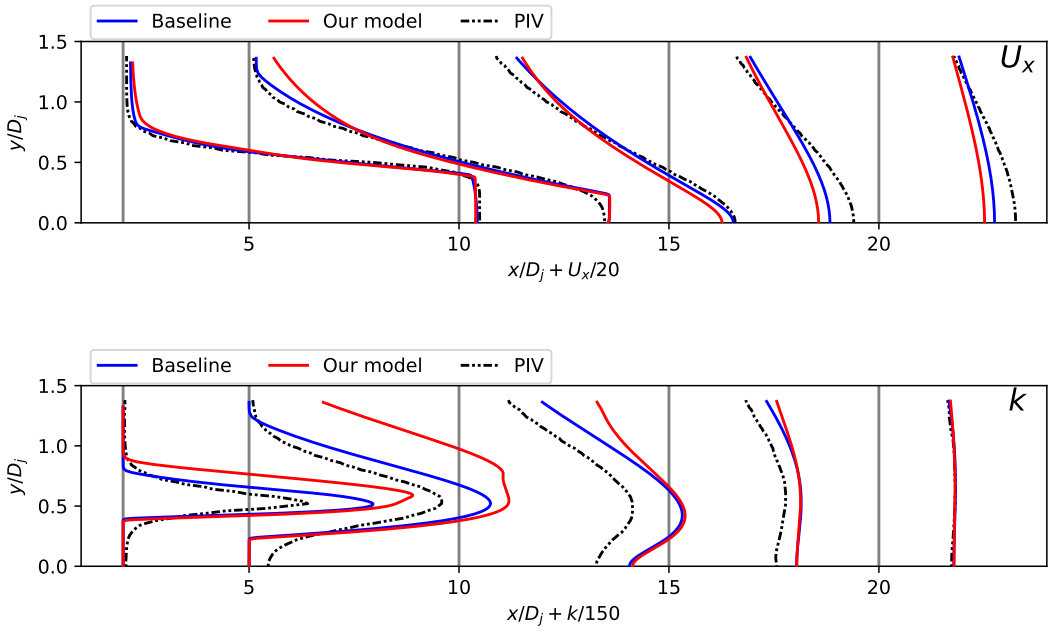


Figure 13: Profiles of x-velocity and k in the axisymmetric jet, using the NASA mesh, comparing the propagated regressed model, baseline k-ω SST and PIV results by Bridges and Wernet [6].

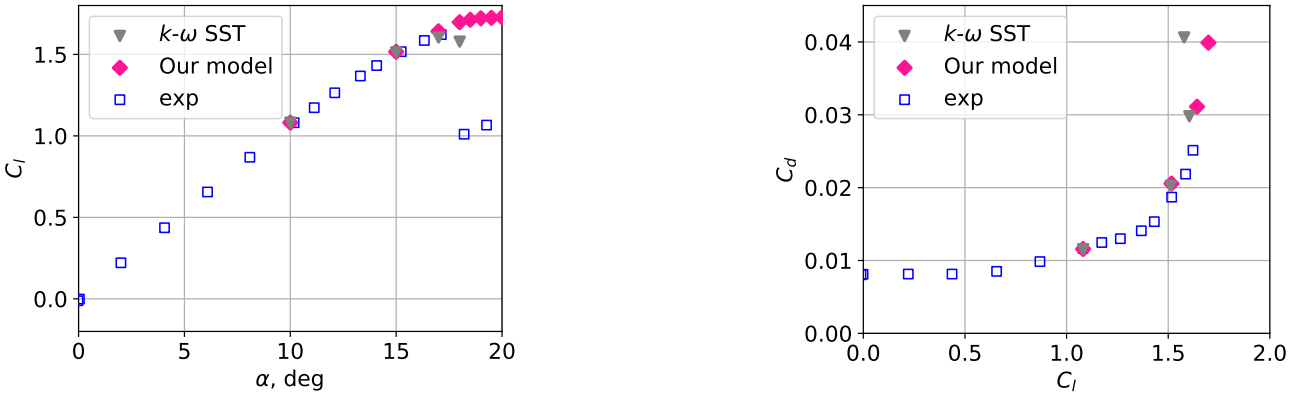
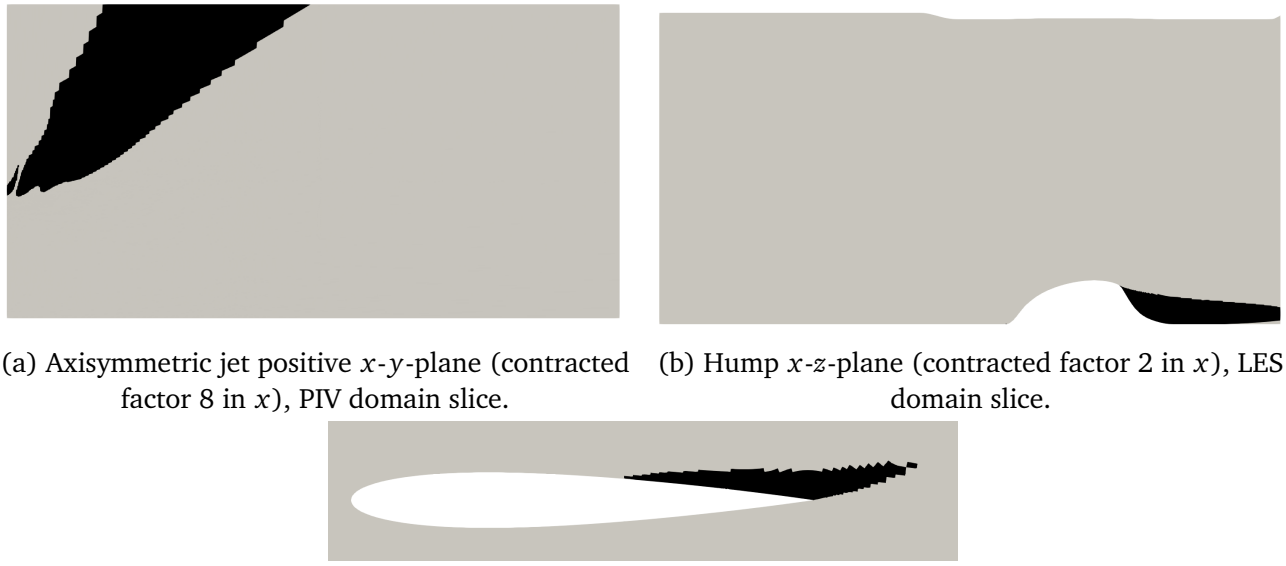


Figure 14: Lift and drag coefficient curves at various angles of attack, comparing the propagated model, baseline k-ω SST and experiment by Ladson [25].

As mentioned, the classifier activation is updated during the runs as well, the final *a-posteriori* activation contours are shown for the jet, hump and airfoil at $\alpha = 18°$ in Fig. 15. The activation contours of the plate and channel are not shown, as they are simply zero everywhere. For easier comparison with the *a-priori* activation in Fig. 10, only the PIV/LES domain slice is shown for the jet/hump. All cases have less activation *a-posteriori* than *a-priori*, this is a desirable property as models may destabilize when encountering cells outside the training region. For the hump, the smaller activation region does not seem to adversely affect the trained model following its good *a-posteriori* agreement with high-fidelity data. For the jet and airfoil, the small activation region already gives a noticeable spoilage of results; this is likely worse without classifier. Of course activation in other regions may also improve results, further research is needed.



(a) Axisymmetric jet positive x-y-plane (contracted factor 8 in x), PIV domain slice.

(b) Hump x-z-plane (contracted factor 2 in x), LES domain slice.

(c) Closeup of airfoil at $\alpha = 18°$, x-z-plane.

Figure 15: Model propagated (*a-posteriori*) classifier activation based on Eq. 62 with regressed $f(\theta)$ in Eq. 63, black indicates $\sigma = 1$; grey indicates $\sigma = 0$.

5.4 Challenge conclusions and recommendations

A collaborative testing challenge is set up by NASA as part of their 2022 symposium on turbulence modeling. The goal is to train and test a turbulence model on five distinct cases; a flat plate, a channel, an axisymmetric jet, a wall mounted hump and a NACA0012 airfoil at various angles of attack. Symbolic expressions are trained for correction terms R and b_{ij}^{Δ}, which are added to P_k and the RST of the k-ω SST turbulence model respectively. Correction fields are found using the k-corrective-frozen approach by Schmelzer et al. and their SpaRTA framework is used for symbolic regression [46]. Training only on the hump, the optimal model is found to be $R = 0.079\epsilon$ and $b_{ij}^{\Delta} = 0$. Additionally a classifier is used to force corrections to zero in most cells following Steiner et al. [54]. Classifier training data is generated using a newly proposed criterion, which has the drawback of being domain dependent. The classifier takes the form of a sigmoid function of a 12-term expression trained on the hump, channel and plate using the SpaRTA framework.

The regressed model is tested *a-posteriori* in combination with the regressed classifier. An important reason for including the classifier is to satisfy the 'do no harm' principle; cases predicted well by k-ω SST (the plate and channel) should not be spoiled by the corrections. Indeed, the plate and channel are found to be severely adversely affected by the raw model, but since the classifier switches off corrections for both, the performance of k-ω SST is retained. For the hump, the classifier is only active behind it, but the model manages to significantly improve predictions over k-ω SST. For the jet, much less activation is observed compared to the training data. This may be for the best, as the model gives slightly worse results than k-ω SST. For the airfoil, the region of classifier activation is also small and the model again adversely affects results compared to k-ω SST. Before stall, the model is in good agreement with k-ω SST and experiment, however, the stall angle and maximum lift coefficient are severely overpredicted compared to both.

Based on the results of the challenge, a number of recommendations are proposed for future research. Firstly, a new classifier criterion should be devised based only on local properties and a simpler model should be trained based on it. Secondly, the regressed model contains no b_{ij}^{Δ} correction, making the model still reliant on the inherently flawed Boussinesq hypothesis. It is recommended to add cases dominated by anisotropic effects to find nonzero b_{ij}^{Δ} models. Thirdly, more cases should be added in general as the model is only trained on the hump. Large domain DNS/LES data should be available for these to prevent problems encountered with the small PIV training domain of the jet. Fourthly, given that the regressed model worked well for the hump, but not for the jet and airfoil suggests different flows may need corrections of a different nature. These could be selectively switched on/off using a multi-class classifier. Fifthly, cases should be steady and incompressible, as the effects of unsteadiness and compressibility on k-corrective-frozen are not fully understood yet. The challenge efforts laid out in the current section preceded subsequent efforts presented later in this work, where the second, third and fifth recommendations are followed.

6 Rectangular duct case setup

The rectangular duct case is based on DNS data by Vinuesa et al. [58], in which flow through a rectangular duct is simulated at duct aspect ratios 1, 3, 5, 7, 10 and 14.4. Each aspect ratio is run at a friction Reynolds number Re_τ of approximately 165. Additionally, the cases with aspect ratio 1 and 3 are run at a friction Reynolds number of approximately 350. Each case is assigned a unique identifier, they are given in Tab. 5. Furthermore, the schematic of the rectangular duct is depicted in Fig. 16, where the main flow is in the $+x$-direction. Note that the indicated infinite domain length in x-direction is only theoretical; it is there to indicate that the case is to be solved for the fully developed turbulent solution. For the DNS data, this is achieved by tripping the initially laminar flow and discarding the solution during the first hundred convective time units. This gives a flow in which the influence of the initial condition is negligible.

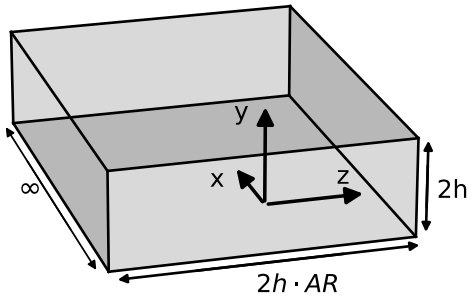


Figure 16: Schematic of the rectangular duct, main flow in $+x$-direction.

The section is organised as follows: In Sec. 6.1, the domain is simplified as much as possible and a mesh is made of this subdomain. In particular, relations are used to generate the mesh based on the dimensionless first cell height and the cell growth ratio. Next, in Sec. 6.2, the setup for a baseline k-ω SST run is given and results of this run are compared with the DNS data to identify where the k-ω SST turbulence model fails. Furthermore, a mesh independence study is performed on both the dimensionless first cell height and the cell growth ratio. Then, in Sec. 6.3, the setup for the frozen run is laid out, where correction terms are found for each cell in the mesh. This gives correction fields based on the frozen DNS data. To validate whether these correction fields also work in a complete RANS solver, the correction fields are injected into the k-ω SST turbulence model in Sec. 6.4.

Table 5: Case identifiers and corresponding duct aspect ratio and friction Reynolds number.

Case identifier	Aspect ratio	Re_τ
rd1L	1	165
rd3L	3	164
rd5L	5	164
rd7L	7	164
rd10L	10	162
rd14L	14.4	166
rd1H	1	342
rd3H	3	363

6.1 Rectangular duct mesh

The duct is symmetric through the y- and z-axis, as can be seen in Fig. 16, this symmetry also holds for the statistical quantities solved for in steady-state RANS. Hence, the domain that needs to be simulated is only one quarter of the actual domain; here the upper-right quarter is chosen. The boundary types that are assigned to the resulting six faces are shown in Fig. 17. The flow is solved for steady-state, meaning x-gradients of flow statistics are zero. Thus, only a single cell is needed in x-direction, with arbitrary thickness. To achieve the steady state solution (solution at $x = \infty$), the cyclic condition is used in x-direction. To still drive the flow, a bulk velocity is specified and the solver automatically applies a pressure gradient to maintain this bulk velocity. Next, due to the presence of both a side- and top-wall, y- and z-gradients are nonzero, meaning a mesh has to be made in the y-z plane (which is later extruded to one single cell in x-direction).

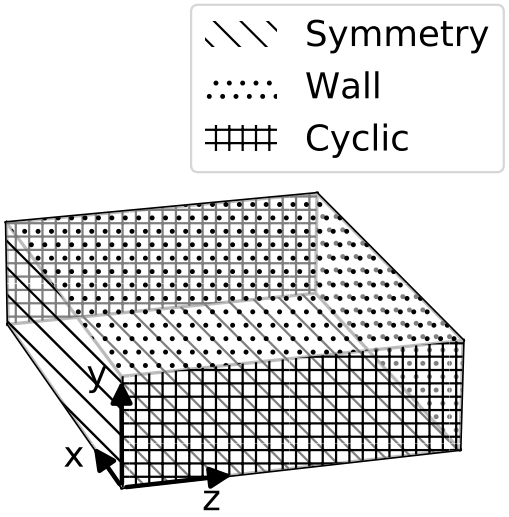


Figure 17: Boundary types used in the rectangular duct mesh.

OpenFOAM's native `blockMesh` utility is used to generate the mesh, which requires the number of cells (N) and the ratio between the first- and last cell (δ_N/δ_1) to be specified along each edge. The optimal value of these parameters is dependent on the shape of the domain and the flow Reynolds number. Two parameters for which the optimal value is less dependent on the domain and the flow Reynolds number are the dimensionless first cell height (y_1^+) and the cell growth ratio (β). Thus, the goal is to find expressions for N and δ_N/δ_1 in terms of y_1^+, β and a-priori known flow quantities.

Consider the equation for y_1^+:

$$y_1^+ = \frac{\delta_1 u_\tau}{\nu} \qquad \Longrightarrow \qquad \delta_1 = \frac{y_1^+ \nu}{u_\tau}, \tag{67}$$

where δ_1 is the first cell height, u_τ is the friction velocity and ν is the laminar kinematic viscosity. This δ_1 appears in the `blockMesh` input parameter δ_N/δ_1, so the expression is rewritten explicitly for δ_1. The only remaining unknown is u_τ, which is found from the equation for the friction Reynolds number Re_τ:

$$Re_\tau = \frac{\overline{u}_\tau h}{\nu} \qquad \Longrightarrow \qquad \overline{u}_\tau = \frac{Re_\tau \nu}{h}, \tag{68}$$

where h is the duct half-height. Note that $\overline{u}_\tau$ in this equation is the average friction velocity, meaning its use in Eq. 67 results in the average y_1^+ being the specified value, with a possibly higher maximum. The maximum y_1^+ is checked after each baseline, frozen and propagation run (see Tab. 10 in Sec. 6.4). Normally, Re_τ is calculated by approximating the skin friction coefficient (such as Eq. 6.1 in [10]), however, in this case Re_τ is directly available from the DNS data [58]. Combining Eq. 67 and Eq. 68 gives the expression for δ_1:

$$\delta_1 = \frac{y_1^+ h}{Re_\tau}. \tag{69}$$

Note that in literature and in OpenFOAM, the y^+ height used is that of the first cell center rather than the full cell, this will be referred to as $y^+_{1/2}$, simply related as $y^+_1 = 2y^+_{1/2}$.

Attaining the required N based on a given y^+_1 and β is rather complicated; one is essentially dividing the domain height by a sum of unequal cell heights. Part of this procedure is laid out in the forum post of the user Tobermory[6]. Below, the math of this procedure is independently verified and the equation for N is derived. First consider the cell height for the second and third cell (δ_2 and δ_3 respectively):

$$\begin{aligned}\delta_2 &= \beta\delta_1,\\ \delta_3 &= \beta\delta_2 = \beta^2\delta_1.\end{aligned}$$

Clearly, the cell height of a certain cell is simply some power of β multiplied with δ_1. The equation is generalized for δ_i ($1 \leq i \leq N$):

$$\delta_i = \beta^{i-1}\delta_1. \tag{70}$$

Now consider the sum of cell heights up to and including cell i, which is given for $i = 1, 2, 3$:

$$\begin{aligned}\sum_{j=1}^{1}\delta_j &= \delta_1,\\ \sum_{j=1}^{2}\delta_j &= \delta_2 + \delta_1 = (\beta+1)\,\delta_1,\\ \sum_{j=1}^{3}\delta_j &= \delta_3 + \delta_2 + \delta_1 = \left(\beta^2+\beta+1\right)\delta_1.\end{aligned}$$

Again, this is generalized for arbitrary i as:

$$\sum_{j=1}^{i}\delta_j = \left(\beta^{i-1} + \beta^{i-2} + \ldots + \beta^2 + \beta + 1\right)\delta_1 = \frac{1-\beta^i}{1-\beta}\delta_1. \tag{71}$$

Note that in the last step of this general expression, the identity of a geometric series is used [1, p. 10, 3.1.10]. Taking this sum in y-direction up to the last cell, which has index $i = N_y$ (where N_y is the number of cells in y-direction), should give a sum equal to the duct half height h. Since h is known beforehand, the sum for $i = N_y$ can be rewritten to give an explicit expression for N_y:

$$\sum_{j=1}^{N_y}\delta_j = h = \frac{1-\beta_y^{N_y}}{\beta_y - 1}\delta_1 \qquad \Longrightarrow \qquad N_y = \ln\left(\frac{h(\beta_y-1)}{\delta_1} + 1\right)/\ln(\beta_y). \tag{72}$$

Note that for general β_y, the N_y attained from Eq. 72 will not be integer. To address this, N_y is rounded up to the nearest integer, which means the actual β_y is slightly lower than the specified one.

[6] Post by Tobermory in thread *[blockMesh] About Simple Grading in blockMesh*. (May, 2020). Tobermory. Archived 21-09-2022, at https://archive.ph/IrpXV

A similar expression as in Eq. 72 could be derived for the number of cells in z-direction (N_z) by replacing h with $AR \cdot h$. This is done for case rd10L, using $y^+_{1/2} = 10$ and $\beta = 1.3$ (tuned for the best visualization), the resulting mesh is shown in Fig. 18. Even though there is ten times more domain to cover in z-direction, there are not ten times more cells in z-direction, arising from the logarithmic nature of Eq. 72. This results in high aspect ratio cells near the origin, even though gradients in y and z are expected to be similar at this location [58]. Thus, it would be more desirable to have square cells near the origin. Since the first cell height is the same in y- and z-direction, square cells at the origin can be achieved by setting δ_N/δ_1 the same in y and z-direction. Note that if $AR > 1$, this results in $\beta_z < \beta_y$. The expression for δ_N/δ_1 is derived next.

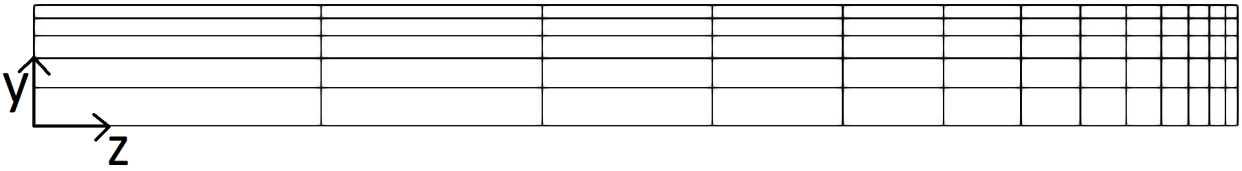


Figure 18: Rectangular duct mesh if Eq. 72 is used in both y- and z-direction, case rd10L with $y^+_{1/2} = 10$ and $\beta_y = 1.3$ for clear visualization.

Finding the expression for δ_N/δ_1 is rather trivial using the already established expression for δ_i in Eq. 70. Plugging in $i = N$ and dividing by δ_1 already gives the desired expression:

$$\frac{\delta_N}{\delta_1} = \beta^{N-1}. \tag{73}$$

With this, the system is closed and a mesh can be generated with a specified $y^+_{1/2}$ and β_y, the resulting `blockMeshDict` is given in Appendix A. The mesh produced by this new `blockMeshDict` is shown for case rd10L in Fig. 19, where $y^+_{1/2} = 10$ and $\beta_y = 1.3$ for clear visualization.

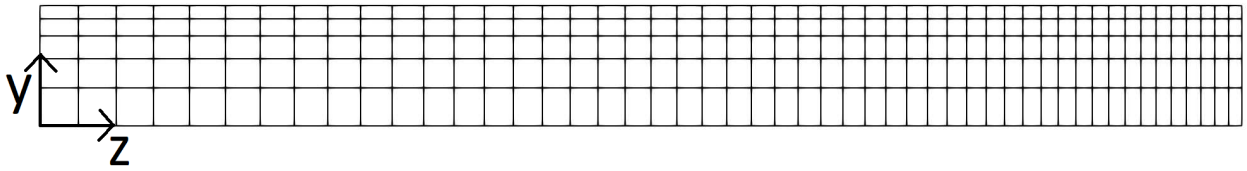


Figure 19: Rectangular duct mesh with $(\delta_N/\delta_1)_z = (\delta_N/\delta_1)_y = \beta_y^{N_y-1}$, case rd10L with $y^+_{1/2} = 10$ and $\beta_y = 1.3$ for clear visualization.

6.2 Baseline setup and convergence study

OpenFOAM's `simpleFoam` solver is used to solve each case. It employs the SIMPLE algorithm to solve the incompressible Navier-Stokes equations, using a turbulence model for the Reynolds stress [7]. The turbulence model used throughout this study is the k-ω SST model as implemented in OpenFOAM [32]. The boundary and initial conditions used for the case are tabulated in Tab. 7. For the walls, low Reynolds number wall functions are used, since the bulk Reynolds number is below 10,000. In the initial condition of U, U_b is the bulk velocity which is calculated from the bulk Reynolds number ($Re_b = U_b h/\nu$). The channel half height is set to $h = 0.001$ and the kinematic viscosity is set to 1.5×10^{-5}; since the solver is incompressible, these can be set to any value. The bulk Reynolds number and resulting bulk velocity are given for each case in Tab. 6. Next, the initial conditions in Tab. 7 are chosen based on similar cases. However, since the case goes to steady state, the choice of initial conditions only effects the rate of convergence. Next, note that the pressure is set to $0\,\mathrm{m^2\,s^{-2}}$ since the relative pressure is solved for rather than the atmospheric pressure. Finally, the flow is driven by a pressure gradient which the solver automatically calculates and applies to give the correct bulk velocity.

Table 6: Given bulk Reynolds number and resulting bulk velocity for each case.

Case	Re_b [-]	U_b [$\mathrm{m\,s^{-1}}$]
rd1L	2500	37.5
rd3L	2581	38.7
rd5L	2592	38.9
rd7L	2605	39.1
rd10L	2580	38.7
rd14L	2665	40.0
rd1H	5693	85.4
rd3H	5817	87.3

Table 7: Boundary- and initial conditions for the square duct case.

Mesh part	U [$\mathrm{m\,s^{-1}}$]	p [$\mathrm{m^2\,s^{-2}}$]	k [$\mathrm{m^2\,s^{-2}}$]	ω [$\mathrm{s^{-1}}$]	ν_t [$\mathrm{m^2\,s^{-1}}$]
Inflow	cyclic				
Outflow					
WallTop	noSlip	zeroGradient	fixedValue 0	omegaWall-Function	nutLowRe-WallFunction
WallSide					
Symmetry-Bottom	symmetry				
Symmetry-Side					
Initial condition	[U_b 0 0]	0	0.02	10	0

The inner and outer residuals, as well as the relaxation factors for the baseline k-ω SST runs are given in Tab. 8. The residuals of ω are much lower since its values varies over many orders of magnitude. The relaxation factors only influence the rate of convergence and stability of the case, not the final solution. The relaxation factors in Tab. 8 give good stability and reasonable convergence for each case. Next, for the outer residuals, consider their convergence for case rd14L shown in Fig. 20; only k, ω and U_x converge, while p, U_y and U_z do not. This is because the residuals in OpenFOAM are normalized, meaning they are divided by their initial value. In the final solution of the case, p, U_y and U_z are zero everywhere, which was also their initial value. Thus the absolute residuals of these parameters were already around machine epsilon at the start of the run, meaning the normalized residuals cannot go down.

Table 8: Residuals and relaxation factors for the baseline rectangular duct case.

Parameter	Inner residual	Outer residual	Relaxation factor
U	10^{-8}	-	0.9
p			0.3
k		5×10^{-6}	0.8
ω	10^{-15}	10^{-10}	

To determine when the case is converged, one would ideally specify a maximum residual for U_x, k and ω. However, in the OpenFOAM version used (OpenFOAM 7), specifying residuals for the component of a vector is not possible. Thus, residuals can only be specified for k and ω. Usually, the case is terminated when all outer residuals are below 10^{-5}. It is observed that the residual of U_x is usually close to that of k. Thus, the outer residuals of k is set to 5×10^{-6} such that the residual of U_x is at least below 10^{-5}, which is confirmed for each case.

As an additional convergence check, probes are placed at various locations in the domain and the evolution of local flow parameters is monitored. The evolution of k is plotted for the baseline k-ω SST case with aspect ratio 14.4 in Fig. 21; at the end of the run, k has converged at each probe. Convergence of outer residuals and flow evolution at these probes is automatically plotted for each case and for each flow parameter. All outer residuals of U_x, k and ω are confirmed to be below 1e-5. Furthermore, all flow parameters are confirmed to have reached a constant value at the end of the run.

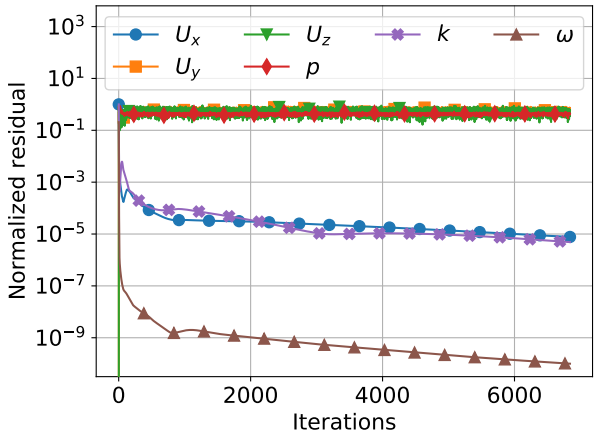


Figure 20: Outer residual versus iteration for various flow parameters, case rd14L, $\beta = 1.1$, $y^+_{1/2} = 0.1$.

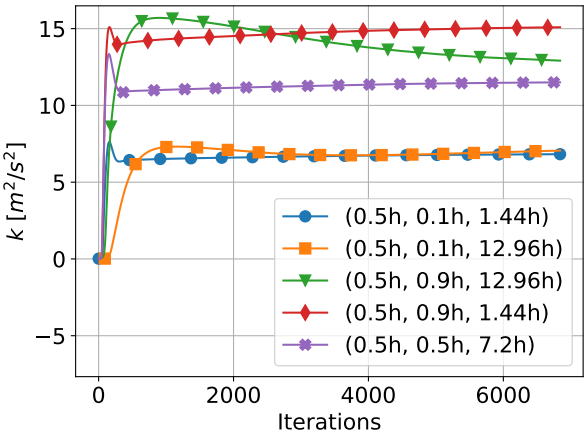
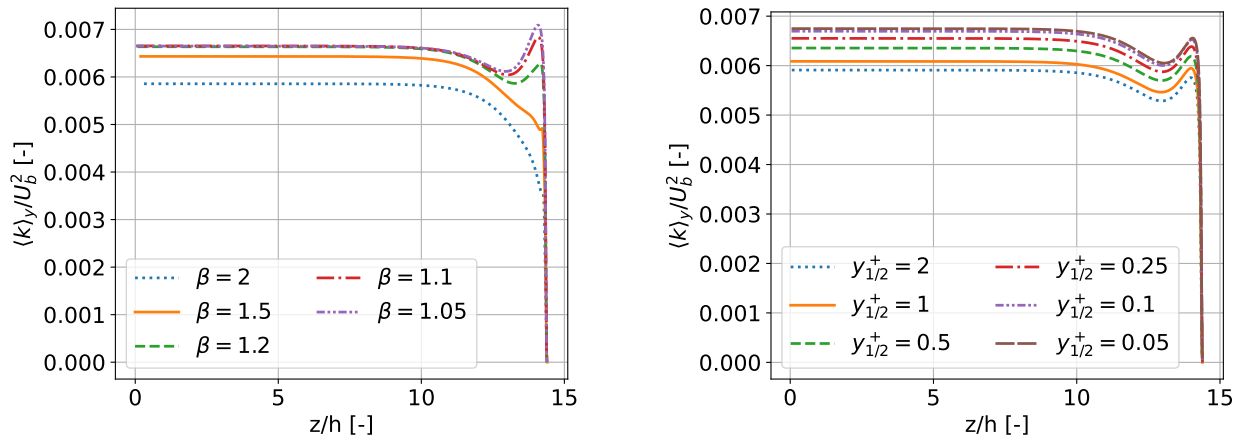


Figure 21: Turbulent kinetic energy versus iteration at various probe locations, case rd14L, $\beta = 1.1$, $y^+_{1/2} = 0.1$.

Now it has been proven that the system of equations being solved by OpenFOAM indeed converges to a stable solution. However, this system of equations is a discretization of an underlying system of nonlinear partial differential equations (PDEs). It must also be proven that this discretized solution is sufficiently close to the continuous solution of the system of nonlinear PDEs. The mesh discussed in Sec. 6.1 represents this discretization. The influence of the discretization is usually tested in a mesh independence study, where various meshes at various levels of refinement are run. At a certain point, further refinement has little effect on the final solution, meaning the result is no longer dependent on the mesh.

As explained in Sec. 6.1, the mesh is defined by two parameters; the cell growth ratio β and the dimensionless first cell center height $y^+_{1/2}$. Mesh independence is to be confirmed with respect to both parameters. Also, it would be best to perform a mesh independence study for each duct case separately. However, this is often unfeasible as it requires running a finer mesh than the actual mesh used, requiring unreasonable runtimes. Thus, the most critical case is usually used for the mesh independence study. In this case, case rd14L is considered critical, as it is expected to have the finest structures. The most critical parameter for mesh independence is found to be y-averaged turbulent kinetic energy. In Fig. 22, it is plotted against z, where in Fig. 22a, β is varied while $y^+_{1/2}$ is kept constant at its converged value. In Fig. 22b, $y^+_{1/2}$ is varied while β is kept constant at its converged value. The final mesh independent parameters are $\beta = 1.1$ and $y^+_{1/2} = 0.1$.



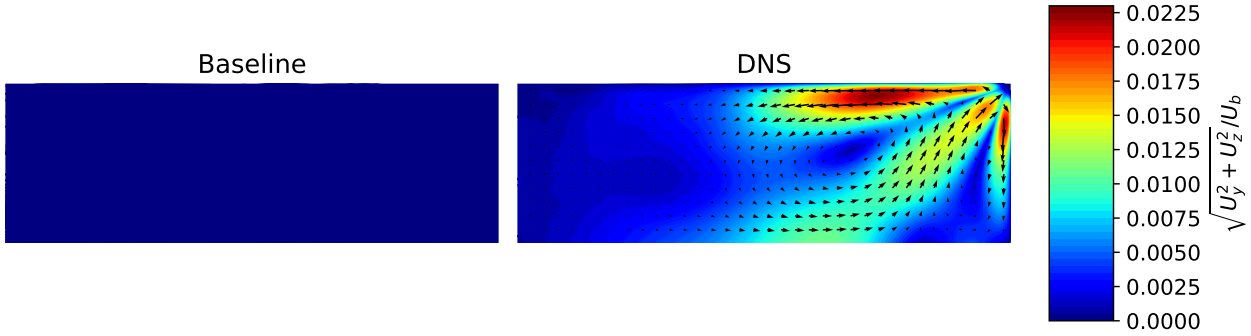
(a) Varying β, $y^+_{1/2} = 0.1$

(b) Varying $y^+_{1/2}$, $\beta = 1.1$

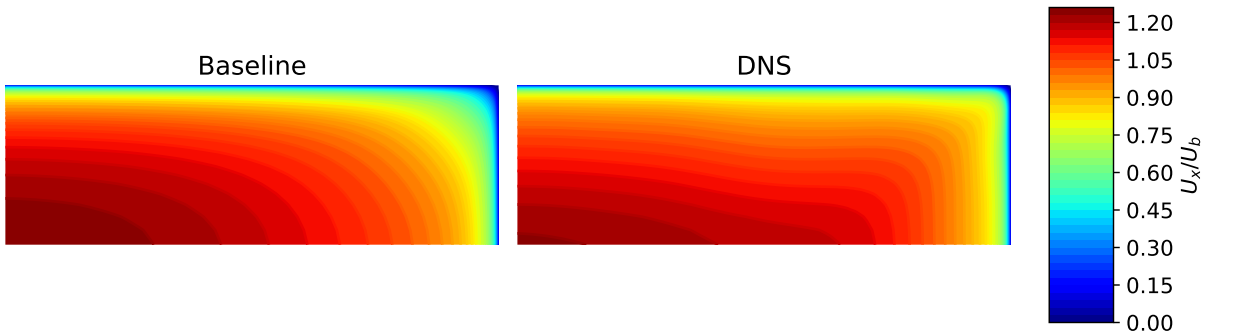
Figure 22: Turbulent kinetic energy averaged over y versus z, for meshes with various β and $y^+_{1/2}$, case rd14L.

A baseline run using the k-ω SST turbulence model is performed to assess its performance and to verify the solver settings and boundary conditions laid out before. Convergence of the baseline run of case rd14L was already shown in Fig. 20 and Fig. 21. Now, a lower aspect ratio case is chosen for clearer visualization. The results of case rd3H are compared with DNS data; contours of the in-plane velocity magnitude and the streamwise velocity are shown in Fig. 23. In Fig. 23a, two in-plane vortices are visible for the DNS data, which generate flow into the corner over the diagonal and out of the corner along the walls. These secondary motions are not predicted by baseline k-ω SST. Though the magnitude of these motions is small compared to the streamwise velocity, their impact is significant. As seen in Fig. 23b, they transport momentum to the corner, giving a much higher U_x there. Similar secondary motions are observed for the other duct cases, none of them are predicted by k-ω SST.

Perkins explains the origin of these secondary motions by analysing the production terms in the mean streamwise vorticity equation [35]. Vorticity turns out to be produced by gradients of Reynolds stresses, which are large in the boundary layer near the duct corner. As explained in Sec. 2.5.4, k-ω SST is based on Boussinesq's hypothesis, which assumes the Reynolds stress tensor to be proportional to the mean strain rate tensor. In the absence of secondary motions, this yields uniform diagonal Reynolds stresses and $\overline{v'w'} = 0$, resulting in zero streamwise vorticity production, explaining the lack of secondary motions.



(a) Normalized in-plane velocity magnitude contours with in-plane velocity vector field overlaid.



(b) Normalized streamwise velocity contours.

Figure 23: Velocity contours of the rd3H case, comparing between baseline (k-ω SST turbulence model) and DNS results.

6.3 Finding correction fields

In order to address the discrepancy described in the prior section, corrections are found for each mesh cell. This is done using the k-corrective-frozen approach proposed by Schmeltzer et al. [46], which is explained in more detail in Sec. 3.3.3. In short, the symmetric tensor correction term b_{ij}^{Δ} is found which is added to the dimensionless anisotropy tensor b_{ij}. Also, the scalar correction term R is found, which is added to the production term of turbulent kinetic energy. The infrastructure to find these terms has already been implemented in the form of a modified solver `frozenSimpleFoam` together with a modified turbulence model `frozenkOmegaSST`.

The modified solver requires the velocity vector field, the Reynolds stress tensor field and the turbulent kinetic energy field from the DNS. The solver needs these for each cell of the RANS mesh, however, they are only available on the DNS mesh. Thus, these quantities need to be interpolated from the DNS mesh onto the RANS mesh. Luckily, the DNS mesh has a slightly larger extent than the RANS mesh and is finer ($\sim$ 2.5$\times$ more cells). Hence, interpolation using a 2-dimensional cubic spline is possible and sufficiently accurate.

During the frozen run, U and k are kept fixed at their DNS value and p is not relevant as only the momentum equation is being solved. Thus, the only parameter that requires iteration is ω. This also makes the frozen run much faster than the baseline k-ω SST run. This lower runtime is exploited to find as accurate correction fields as possible by setting the outer residual of ω to 10^{-15} (same as inner). A relaxation factor of 0.9 is observed to give a stable though fast convergence for all cases. At then end of the run, one outer iteration is performed to solve for pressure; since U is fixed, no further iteration is needed.

The boundary conditions are the same as those listed in Tab. 7, also for the parameters obtained from the DNS, as interpolation is only performed for cells and not for faces. The newly added Reynolds stress tensor field τ_{ij} uses the same boundary conditions as k (fixedValue 0_{ij} at the wall). For p, ω and ν_t, the initial conditions are the same as in Tab. 7; for U, k and τ_{ij}, they are simply the interpolated DNS value.

To assess the convergence during the frozen run, the residual of ω is stored at each iteration and the resulting convergence plot is shown in Fig. 24. As expected, after startup ω converges linearly in the logarithmic plot until it reaches the specified tolerance of 10^{-15}. As an additional convergence check, the value of ω and the value of the correction terms are probed at various locations and stored at each iteration. All were at a constant value at the end of the run for each case, confirming each case converged. The b_{13}^{Δ} term showed the slowest convergence so its probe convergence is shown in Fig. 25. Visually, b_{13}^{Δ} seems to converge only after $\sim$ 1000 iterations. At this point, the outer residual of ω is $\sim 10^{-9}$, which is a much lower tolerance than one would use in a classical RANS run. Thus, future frozen runs should use an outer tolerance for ω of at most 10^{-10} and probes should be placed in the flow to confirm convergence.

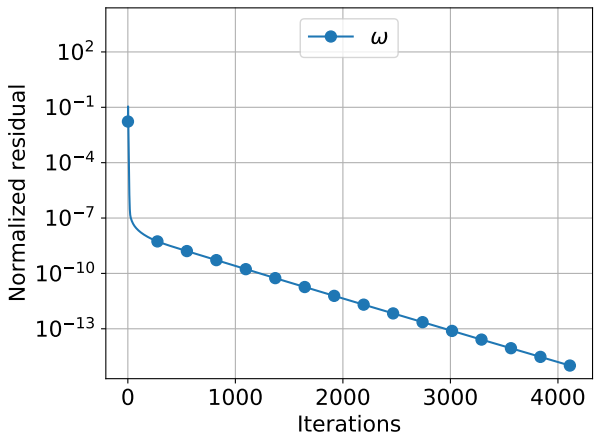


Figure 24: Outer residual of ω versus iteration for frozen run of case rd14L.

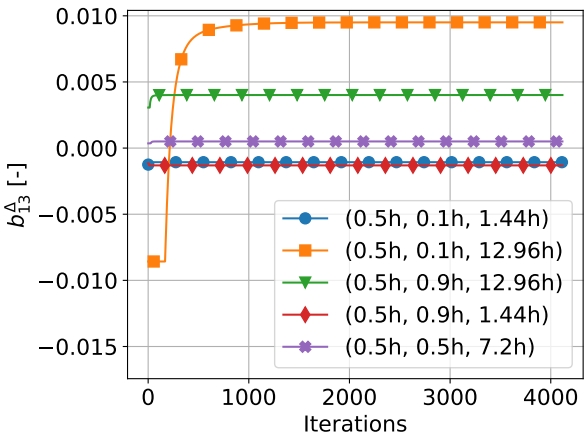


Figure 25: b_{13}^{Δ} versus iteration at various probe locations for frozen run of case rd14L.

6.4 Validating the correction fields

In the previous section, correction fields were found for each case based on the velocity and Reynolds stress of the DNS data. However, these were kept frozen, while in a full RANS simulation, they are updated each iteration. To test whether the correction fields also yield good improvements in a full RANS run, they are propagated in the modified turbulence model `propagationkOmegaSST`. This turbulence model is run with the native `simpleFoam` solver and modifies the k-ω SST turbulence model to also include the corrections.

The boundary conditions of the propagation run are the same as those listed in Tab. 7. However, the initial conditions are different, as the generic initial conditions in Tab. 7 may lead to divergence. Best would be to use the DNS fields as initial conditions, since these are closest to the supposed propagation solution. However, this is potentially misleading as the propagated solution may appear closer to the DNS than it actually is if it is not iterated sufficiently. Thus, the baseline solution fields are used as initial conditions, such that any improvement of the propagated solution over the baseline solution is achieved through iteration. Next, the case is somewhat less stable than the baseline case, so lower relaxation factors are used, they are given in Tab. 9. For the baseline, no outer residual is specified for U and p as (components of) these remain zero the whole run, leading to normalized residuals of order one. With the correction fields, U and p are nonzero such that all their residuals converge. As U_y and U_z are much smaller than U_x, rather low residuals are required for convergence, the outer residuals of U, p and k are set to 10^{-6}.

Table 9: Residuals and relaxation factors for the propagation rectangular duct case.

Parameter	Inner residual	Outer residual	Relaxation factor
U	10^{-8}	10^{-6}	0.8
p			0.5
k			0.4
ω	10^{-15}	10^{-10}	

In order to assess whether the run indeed converges, outer residuals are again plotted against iteration in Fig. 26. As expected, all outer residuals go to their specified tolerance in a straight line in the log-plot after some start-up. Additionally, the evolution of flow parameters at certain probe locations is again measured. U_y and U_z turn out to be most critical for convergence, the probe convergence of U_y is plotted in Fig. 27. At the end of the run, U_y has reached a constant value, indicating the run has converged. Using the same analysis, convergence is confirmed for the other cases as well.

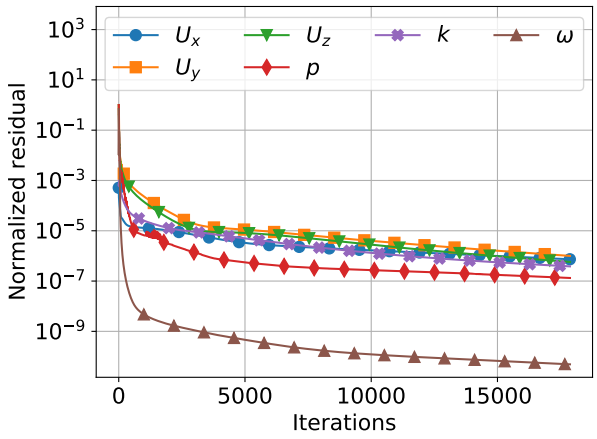


Figure 26: Outer residual versus iteration for various flow parameters, propagation of case rd14L.

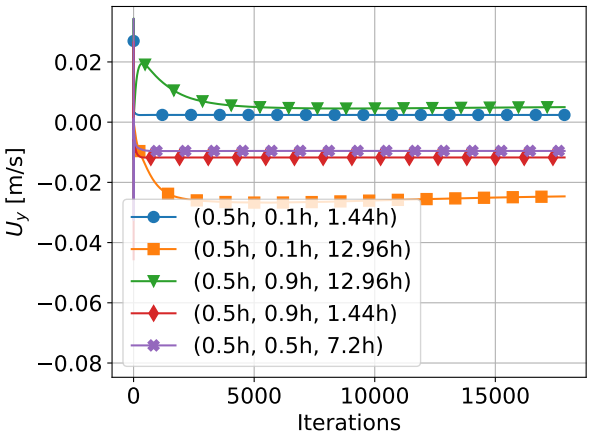
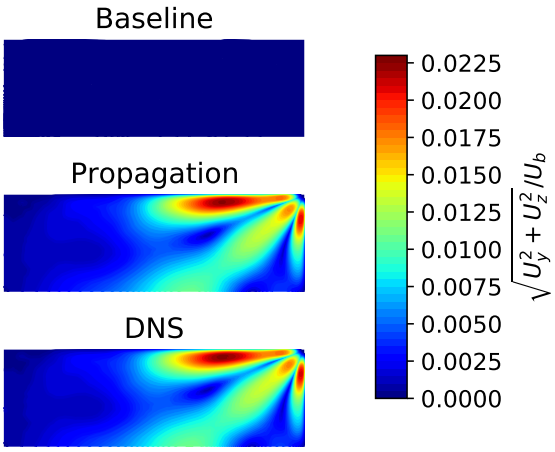
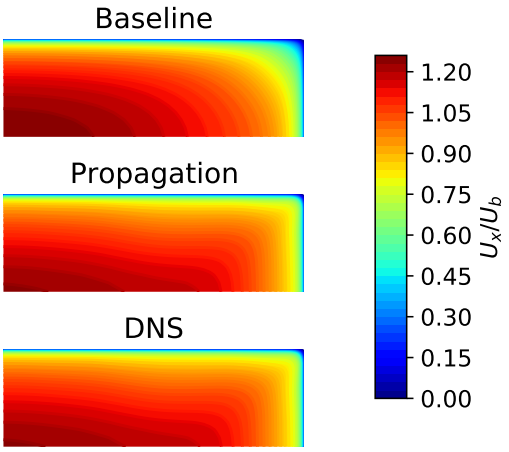


Figure 27: y-velocity versus iteration at various probe locations, propagation of case rd14L.

Contours of both in-plane velocity magnitude and streamwise velocity are compared between the baseline (k-ω SST), propagation (k-ω SST with corrections) and the DNS, they are shown in Fig. 28. No notable differences are observed between the propagation and the DNS in this figure. In order to check whether propagation is really the same as DNS, the in-plane velocity magnitude is plotted along the duct diagonal in Fig. 29, where r is the distance from the duct center. The complex shape of the DNS data is followed extremely well by the propagation, with only a small deviation at $r/h \approx 2.7$. Additionally, the y-averaged turbulent kinetic energy is plotted against z in Fig. 30. Again, the propagation follows the DNS alsmost exactly. Similar contour and line plots are made for each case and the propagation has an almost exact match with the DNS for each one. Hence, the correction fields found during the frozen runs are considered valid.



(a) Normalized in-plane velocity magnitude contours



(b) Normalized streamwise velocity contours

Figure 28: Velocity contours of the rd3H case, comparing between Baseline (k-ω SST turbulence model), propagation (k-ω SST with correction terms b_{ij}^{Δ} and R) and DNS results.

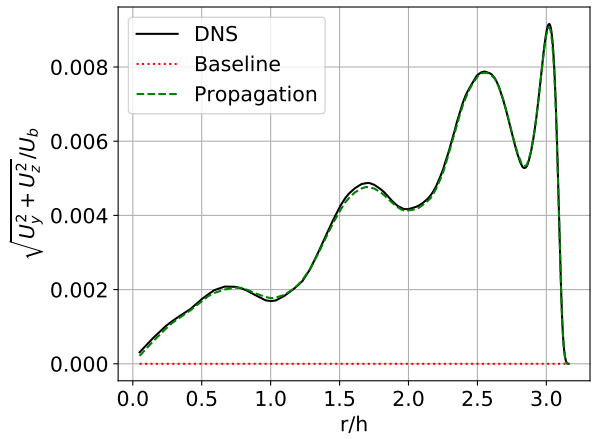


Figure 29: In-plane velocity magnitude along the diagonal of the rectangular duct, case rd3H.

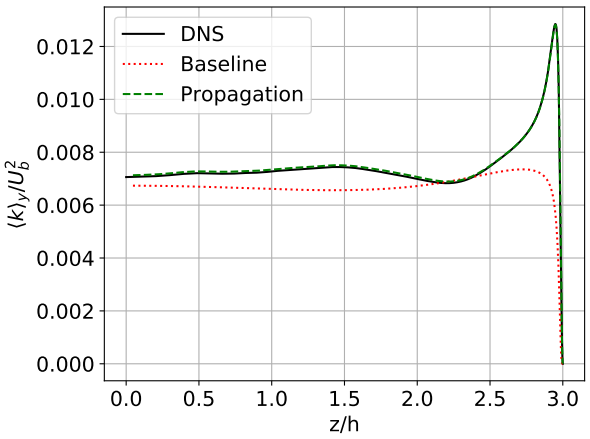


Figure 30: Turbulent kinetic energy averaged over y versus z, case rd3H.

One point left unaddressed is the use of the average Re_τ in Eq. 69, which is used to find the mesh spacing required to get the specified $y^+_{1/2}$ of 0.1. One is usually interested in the maximum value of $y^+_{1/2}$, which depends on the maximum value of Re_τ. To quantify the difference between the specified $y^+_{1/2}$ and the maximum $y^+_{1/2}$ in the case, $y^+_{1/2,max}$ is computed for each case, for the baseline, frozen and propagation run. The maximum $y^+_{1/2}$ can be easily found by postProcessing using OpenFOAM's `yPlus` function, the results are given in Tab. 10. Clearly, the maximum $y^+_{1/2}$ is close to the specified one (maximum deviation of 10%). Hence, the $y^+_{1/2}$ estimation based on the average Re_τ is sufficiently accurate for the mesh design of the current case.

Table 10: Maximum $y^+_{1/2}$ for each baseline, frozen and propagation rectangular duct case.

Case	Baseline $y^+_{1/2,max}$	Frozen $y^+_{1/2,max}$	Propagation $y^+_{1/2,max}$
rd1L	0.109	0.102	0.102
rd3L	0.108	0.102	0.102
rd5L	0.105	0.102	0.102
rd7L	0.104	0.102	0.102
rd10L	0.101	0.099	0.099
rd14L	0.103	0.102	0.101
rd1H	0.100	0.096	0.096
rd3H	0.110	0.106	0.106

7 Heterogeneous roughness case setup

The heterogeneous roughness case is based on wall modeled LES data by Amarloo et al. [3], in which flow over a surface with periodic streamwise roughness strips is studied. In their paper, surface roughness and strip width are kept constant, while the unstable stratification strength is varied. Additional simulations were performed by Amarloo and co-workers with a neutral boundary layer (no stratification), where the width of the roughness strips and the roughness heights are varied. Each case is assigned a unique identifier, given in Tab. 11, together with the roughness heights and strip width. At the time of writing, results of these additional simulations have not been published, but they were generously shared with the author and they will be used in the current work. The schematic of the case is shown in Fig. 31, where the main flow is in the -x-direction. The inifinite domain length in x is only theoretical; it is to indicate that the case should be solved for the fully developed turbulent solution. For the LES, this is attained by discarding the timesteps before the flow reaches steady-state and then averaging twenty steady-state flow-throughs. Finally, the domain height should also be infinite in theory; the height h was chosen such that the influence of the top boundary on the rest of the domain is negligible.

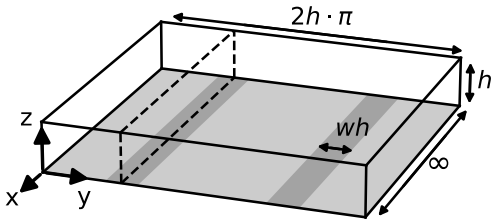


Figure 31: Schematic of the heterogeneous roughness, main flow in -x-direction.

The section is organised as follows: In Sec. 7.1, the boundary types of the domain are given together with the mesh used (which was adapted from Amarloo et al.). Then, in Sec. 7.2, the setup of the baseline k-ω SST run is given, followed by the case convergence through the run. Furthermore, a mesh independence study is performed and finally results are compared with the LES data. Then, in Sec. 7.3, the setup is given for the frozen run to find correction fields which are to address the discrepancy between RANS and LES. Finally, in Sec. 7.4, these correction fields are injected into the k-ω SST turbulence model and run in a complete RANS run for validation.

Table 11: Case identifiers with the corresponding wall roughnesses and the roughness strip width.

Case identifier	Rough roughness height [m]	Smooth roughness height [m]	Roughness strip width [m]
hr00	0.5	0.005	$0.64\pi h$
hr03	0.5	0.05	$0.64\pi h$
hr04	0.5	0.0005	$0.64\pi h$
hr05	0.05	0.005	$0.64\pi h$
hr06	0.05	0.00005	$0.64\pi h$
hr07	0.005	0.00005	$0.64\pi h$
hr08	0.5	0.005	$0.32\pi h$
hr09	0.5	0.005	$0.96\pi h$
hr10	0.5	0.005	$1.28\pi h$
hr13	0.05	0.00005	$0.64\pi h$
hr14	0.005	0.00005	$0.32\pi h$
hr15	0.005	0.00005	$0.96\pi h$
hr16	0.005	0.00005	$1.28\pi h$

7.1 Heterogeneous roughness mesh

The case has three symmetry planes, one of which is indicated with a dashed line in Fig. 31. For the RANS simulation, only the part to the left of this symmetry plane is simulated. The resulting boundary types assigned to each of the six domain faces are shown in Fig. 32. For the bottom wall patch, different roughness heights are specified for faces inside and outside the roughness strip. The cyclic condition is used in x-direction to solve for steady-state ($x = \infty$). At $x = \infty$, gradients with respect to x will be zero, meaning the mesh only needs one cell in x-direction. A bulk velocity is specified in x-direction and a pressure gradient is automatically applied to maintain this bulk velocity. Finally, the symmetry condition is applied to both sides as they are geometrically symmetric. Symmetry is also applied to the top to act as a farfield condition.

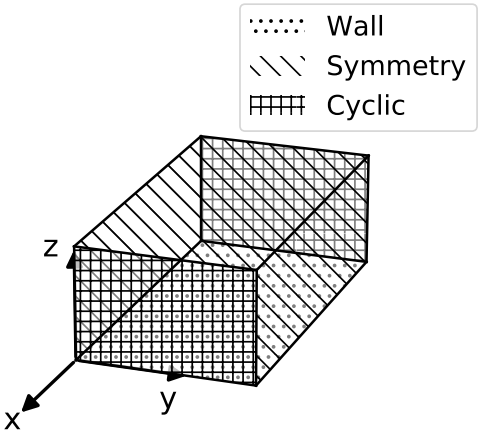


Figure 32: Boundary types used in the heterogeneous roughness mesh.

The RANS mesh employed by Amarloo et al. is also used in the present work, the lower left corner is shown in Fig. 33. The mesh is equally spaced in y- and z-direction, except the leftmost and rightmost column of cells, which have half the width of the other cell columns. There are 33 cells in y-direction and 127 cells in z-direction. The mesh is 500 m in z-direction and 250π m in y-direction. The case Reynolds number is intentionally large ($Re_\tau \approx 1.3 \times 10^7$), since this increases the error propagation [2]. The first cell height is also intentionally large (3.9 m), as it should be at least twice the largest roughness height [34]. Together, this makes for a maximum $y^+_{1/2}$ of around 30000, meaning wall functions have to be used, these are discussed in the next section.

Figure 33: Lower left corner of the heterogeneous roughness mesh.

7.2 Baseline setup and convergence study

OpenFOAM's `simpleFoam` solver (SIMPLE algorithm [7]) is again used to solve the incompressible Navier-Stokes equations, with the $k-\omega$ SST turbulence model. The boundary and initial conditions are tabulated in Tab. 12. Atmospheric wall functions are used for ω and ν_t, however, these are only implemented in newer versions of OpenFOAM, not in the version of OpenFOAM used in the current work (OpenFOAM 7). Thus, the atmOmegaWallFunction and the atmNutUWallFunction are rewritten to OpenFOAM 7 syntax and compiled as custom wall fuctions. The code of these custom wall functions can be found in Appendix. B.

Table 12: Boundary and initial conditions for the heterogeneous roughness case.

Mesh part	U [m s^{-1}]	p [m^2 s^{-2}]	k [m^2 s^{-2}]	ω [s^{-1}]	ν_t [m^2 s^{-1}]
Inflow	cyclic				
Outflow					
WallBottom	noSlip	zeroGradient	zeroGradient	atmOmega-WallFunction	atmNutU-WallFunction
SymmetryLeft	symmetry				
SymmetryRight					
SymmetryTop					
Initial condition	[U_b 0 0]	0	0.5	0.0192	0

The initial conditions in Tab. 12 are chosen close to their expected values for fast convergence. Note that the pressure is set to 0 m^2 s^{-2} since this is a relative pressure rather than atmospheric pressure. Finally, U_b (used in the initial condition of U) is the bulk x-velocity in the domain. Rather than a specified bulk velocity, Amarloo et al. used a specified force to drive the flow, giving a different bulk velocity for each case. In this work, the appropriate bulk velocity is found by integrating the streamwise velocity from their LES data. The solver automatically calculates the required pressure gradient to drive the flow to the U_b specified. The bulk velocity is given for each case in Tab. 13.

Table 13: Bulk velocity specified for each case.

Case identifier	hr00	hr03	hr04	hr05	hr06	hr07	hr08
Bulk velocity [m/s]	8.37	7.51	9.15	9.84	10.89	13.41	8.91
Case identifier	hr09	hr10	hr13	hr14	hr15	hr16	
Bulk velocity [m/s]	8.09	7.56	11.87	14.10	12.89	12.36	

The inner and outer residuals as well as the relaxation factors for the baseline case are given in in Tab. 14. Since there is no refinement near the wall, the range of ω values is not as large, so it has the same residuals as the other parameters. The chosen relaxation factors give good stability and a reasonably fast convergence for all cases. Next, the outer residuals are plotted for case hr14 in Fig. 34. This case is chosen as it is considered most critical since it has the highest y^+ value and thus the highest Re_τ (see Eq. 69). As explained in Sec. 6.2, p, U_y and U_z remain zero meaning their normalized residuals do not converge. Again, the outer residuals of k and ω are specified at 10^{-6} to ensure the outer residual of U_x is also sufficiently low, indeed they all reach below 1×10^{-5}. Finally, probes are placed in the flow to further assess convergence; the evolution of k through the run is plotted for case hr14 in Fig. 35. Clearly, at the end of the run, k has assumed a constant value and can be considered converged, this is also observed for ω and U_x. The other baseline roughness cases are also confirmed to be converged using the same check.

Table 14: Residuals and relaxation factors for the baseline heterogeneous roughness case.

Parameter	Inner residual	Outer residual	Relaxation factor
U	10^{-8}	-	0.9
p			0.3
k		10^{-6}	0.95
ω			

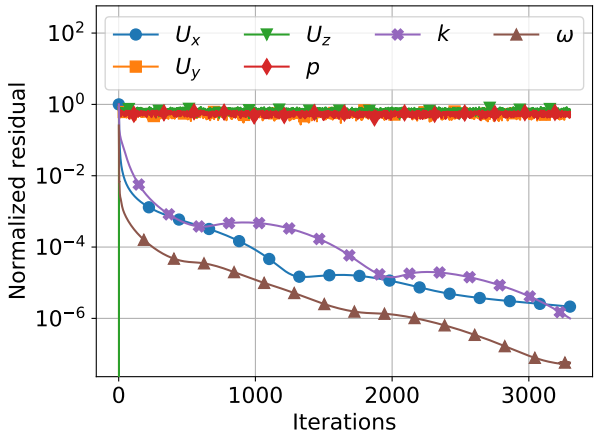


Figure 34: Outer residual versus iteration for various flow parameters, case hr14.

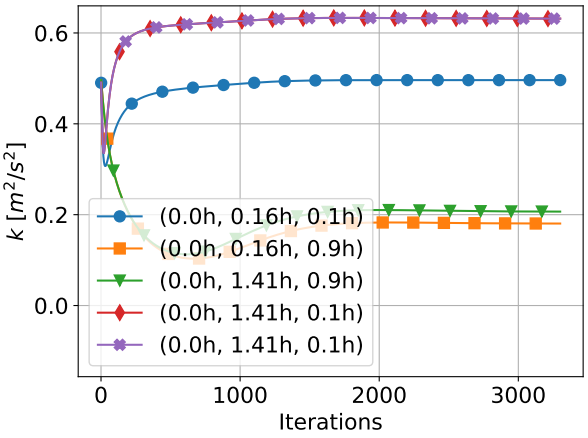
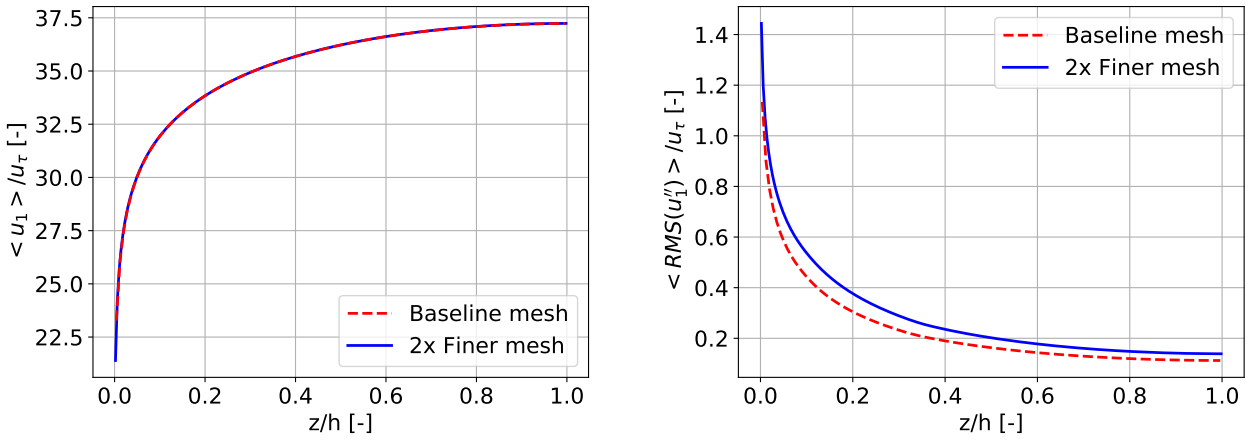


Figure 35: Turbulent kinetic energy versus iteration at various probe locations, case hr14.

Though not explicitly mentioned, it assumed that Amarloo et al. performed a mesh independence study on the mesh. To be certain the mesh is sufficiently fine, such that no discretization errors are trained into the models, an additional mesh independence check is performed. A finer mesh is created by splitting each cell into four cells, yielding a factor two refinement in y- and z-direction. Mesh independence is only checked for case hr14, since it is considered critical. Mesh independence is studied by plotting two variables against z, both also used by Amarloo et al. Firstly, the y-averaged streamwise velocity u_1, nondimensionalized by the friction velocity u_τ, is plotted against z/h in Fig. 36a. Here, angle brackets represent averaging in y-direction. Secondly, the y-averaged RMS dispersive streamwise velocity $< \text{RMS}(u_1'') >$, nondimensionalized by u_τ, is plotted against z/h in Fig. 36b. The dispersive velocity u_i'' is defined as [2]:

$$u_i'' = u_i - < u_i > . \tag{74}$$

In Fig. 36a, there is no visible difference between the two meshes, while in Fig. 36b a small difference is visible. Still, the difference is small enough to consider the baseline mesh sufficiently fine.



(a) y-Averaged streamwise velocity vs z

(b) y-Averaged RMS dispersive streamwise velocity vs z

Figure 36: Mesh independence study of the baseline heterogeneous roughness mesh and a versioned refined by a factor two in y- and z-direction, case hr14.

In order to verify the solver settings discussed above and identify the shortcomings of k-ω SST, a baseline run is performed with the k-ω SST turbulence model for each case. The convergence of this run was already shown for the case hr14 in Fig. 34 and Fig. 35. Case hr00 is used in the comparison with the LES, as it exhibits one of the strongest differences and it is also the training case used by Amarloo et al. [2]. Contours of dispersive x-velocity (see Eq. 74) are shown in Fig. 37, with the in-plane velocity overlaid as a vector field. Dispersive x-velocity is nondimensionalized by U_0; the mean streamwise velocity averaged at $z = h$. As was the case for the rectangular duct, k-ω SST completely fails to predict the in-plane motions in this case. This is caused by the same mechanism as the duct discussed in Sec. 6.2; in-plane gradients of in-plane Reynolds stresses are erroneously predicted as zero. The lack of in-plane motions also has a significant effect on the streamwise velocity, as u_1'' is significantly different between baseline and the LES.

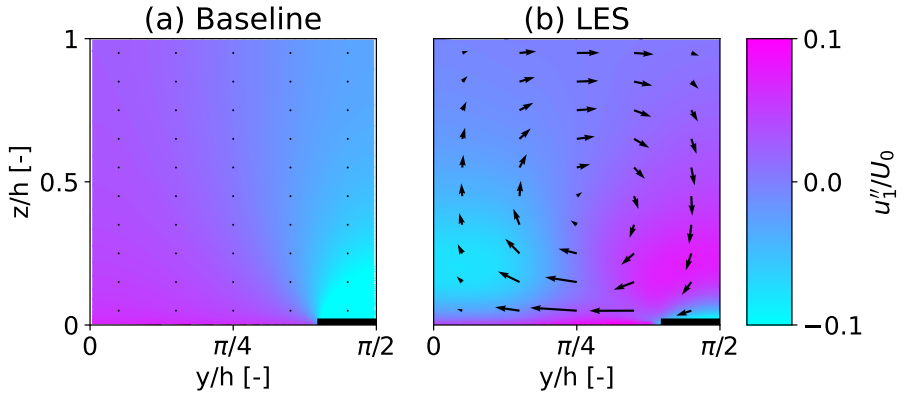


Figure 37: Contours of dispersive x-velocity (see Eq. 74) with the in plane velocity vector field overlaid, comparing between baseline (k-ω SST) and WMLES results, case hr00.

7.3 Finding correction fields

The k-corrective-frozen approach further explained in Sec. 3.3.3 is used to find the correction terms b_{ij}^{Δ} and R in each cell to address the discrepancy between RANS and LES. Again the custom `frozenSimpleFoam` solver is used together with the custom `frozenkOmegaSST` turbulence model to find these correction terms. Before the run, however, the velocity, Reynolds stress and turbulent kinetic energy must be transformed from the LES to the RANS mesh. The LES mesh is different from the RANS mesh in that it models the full domain from $y = 0$ till $y = 2\pi$ (see Fig. 31), has many cells in x-direction and is solved over many timesteps. In the LES data provided, averaging is already performed over time, in the x-direction and over the symmetry at $y = \pi$. This means the LES data is in the y-z plane from $y = 0$ till $y = \pi$.

In order to transform the LES data to the RANS mesh, it is first mirrored through the plane at $y = \pi/2$ and then averaged, such that it has the same size as the RANS mesh. Since the LES mesh used the same cell spacings in y- and z-direction as the RANS mesh, there is a direct overlap between their cells. To account for slight differences in the cell center coordinates (especially in the left and right columns), a matching is performed where the closest LES cell is found for each RANS cell. The velocity, Reynolds stress and turbulent kinetic energy values of this LES cell are then assigned to the RANS cell.

As explained in Sec. 6.3, ω is the only parameter being solved for. Again, the outer and inner residual of ω are set to 10^{-15} and a relaxation factor of 0.9 is used. The same boundary conditions as those listed in Tab. 12 are used. For the Reynolds stress tensor field (τ_{ij}), the same boundary conditions are used as for k, except at the wall where a fixedValue 0_{ij} is used. For p, ω and ν_t, the initial conditions from Tab. 12 are used while the LES value is used for U, k and τ_{ij}.

Convergence of the run is assessed by monitoring the residual of ω, plotted in Fig. 38. After some initial iterations, ω starts converging linearly in the log-plot, as expected. At the very end of the run, there is a slight decrease in slope; this is thought to be due to the inner residual being the same as the outer residual. If this phenomenon leads to nonconvergence, the inner residual should be lowered. Additionally, probes are placed in the flow to check the convergence of ω as well as the correction terms. The b_{22}^{Δ} term showed the slowest convergence, so its probed values are plotted against iteration in Fig. 39. At the end of the run, all probes reach a constant value, indicating convergence. Similar convergence for both the outer residual as well as the probes is observed for all frozen roughness cases. Convergence seems to be reached around iteration 1000, at which point the outer residual of ω is 10^{-6}. As was the case for the rectangular duct, this is a much lower convergence residual than in a classical RANS run.

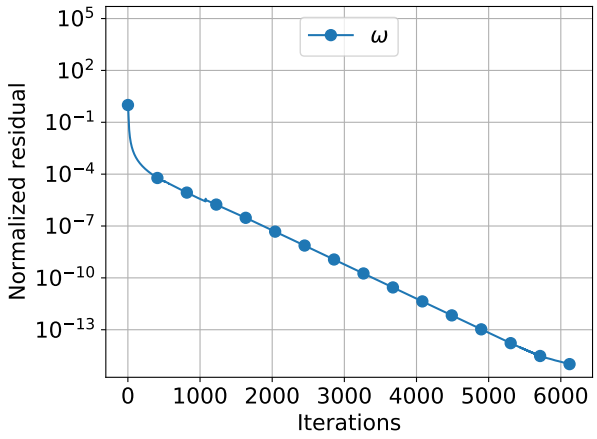


Figure 38: Outer residual of ω versus iteration for frozen run of case hr00.

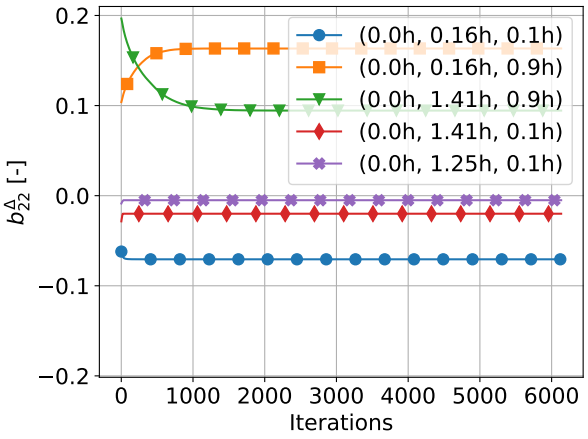
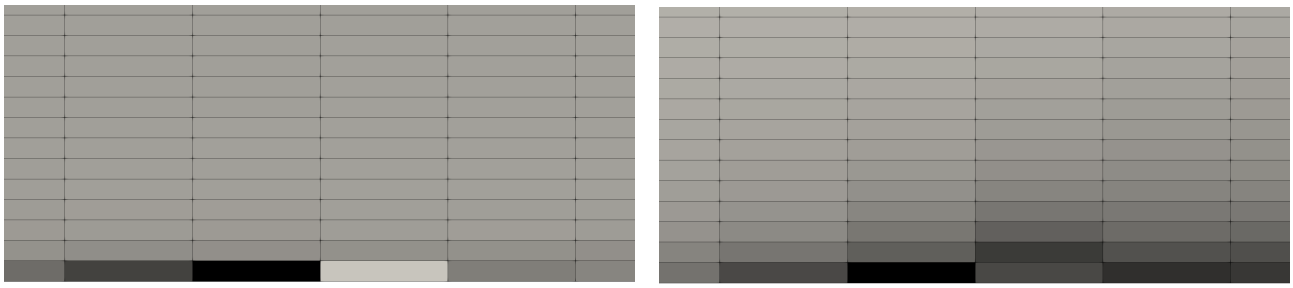


Figure 39: b_{22}^{Δ} versus iteration at various probe locations for frozen run of case hr00.

Next, the correction fields are inspected, where b_{ij}^{Δ} is multiplied by k to get its actual contribution to the Reynolds stress tensor. Corrections are close to zero over most of the domain, except near the transition from smooth to rough wall. Contours of R and $k \cdot b_{ij}^{\Delta}$ magnitude are shown at this location in Fig. 40, where each mesh cell is uniformly filled with its value. Both corrections have high gradients at the wall transition, the cells appear far too coarse for second order finite volume (used in the present work) to accurately resolve these gradients. Note that this mesh is actually sufficiently accurate for the LES, as the LES is based on a pseudo-spectral solver which is able to represent gradients much more accurately [3].

Given that the mesh appears too coarse for the present finite volume based RANS, further mesh refinement is desired. However, refinement of the mesh is not trivial, as the high-fidelity data is only available on the current mesh, requiring some form of upsampling. This could be possible with the derivatives from the LES, but these are not readily available. Also, the first cell height needs to be twice the roughness height, meaning little wall-normal refinement is possible. Thus, the mesh is not further refined for now, the impact of these non-smooth correction fields is further assessed in their propagation and in model testing.



(a) R

(b) $k \cdot \left|\left| b_{ij}^{\Delta} \right|\right|$

Figure 40: Magnitude of correction fields near the transition from smooth (left) to rough (right) wall, case hr00.

7.4 Validating the correction fields

The correction fields found in the previous section are now validated in a full RANS run where the velocity and Reynolds stress fields are allowed to update. Again, the modified turbulence model `propagationkOmegaSST` is used together with the native `simpleFoam` solver. The same boundary conditions as listed in Tab. 12 are used. The baseline results are again used as initial conditions: as explained in Sec. 6.4, the LES fields would provide better initial conditions, but could potentially produce misleading results.

The residuals and relaxation factors used are tabulated in Tab. 15. Due to stability issues, the relaxation factors are significantly lower than for the baseline, leading to much slower convergence. The inner residuals are kept the same as the baseline. Next, since p, U_y and U_z are now nonzero, their outer residuals converge as well. Thus, outer residuals are specified for all variables. Since U_x is much larger than U_y and U_z, the outer residual needs to be lower than usual. For simplicity, all outer residuals are set to 10^{-6}. In order to assess convergence, the outer residuals are plotted against iteration in Fig. 41. After some startup, all residuals reach below their specified tolerance. As mentioned, U_y and U_z are critical for convergence, so the probe convergence of U_z is shown in Fig. 42. After roughly 20,000 iterations U_z reaches a constant value, confirming convergence. This corroborates that the standard 10^{-5} outer residual tolerance would have been too high for U_z. Convergence is confirmed for all other cases using a similar check.

Table 15: Residuals and relaxation factors for the propagation heterogeneous roughness cases.

Parameter	Inner residual	Outer residual	Relaxation factor
U	10^{-8}	10^{-6}	0.7
p			0.3
k			0.5
ω			

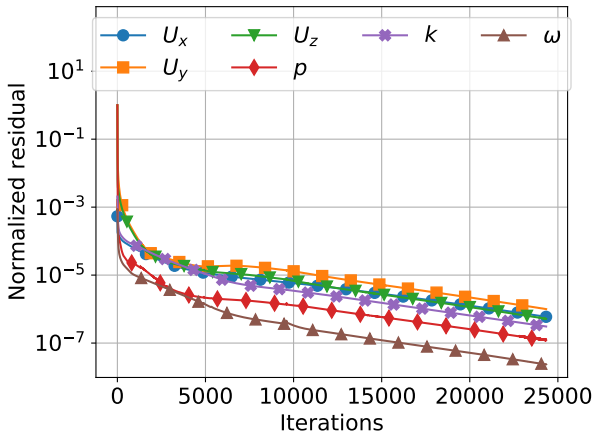


Figure 41: Outer residual versus iteration for various flow parameters, propagation of case hr14.

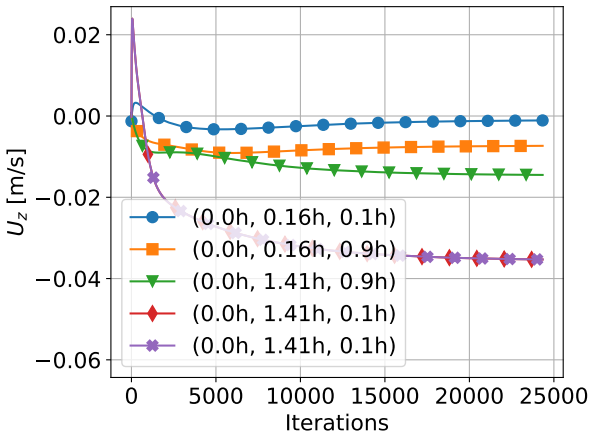


Figure 42: z-velocity versus iteration at various probe locations, propagation of case hr14.

For each case, a propagation run is performed using the settings just described. The results of case hr00 (which was also propagated by Amarloo et al. [2]) are presented; contour plots of dispersive x-velocity with the in-plane velocity vector field overlaid are shown in Fig. 43. In this figure, there is no discernible difference between the propagation results and the LES, indicating valid correction fields. Possible differences between propagation and the LES are further investigated by plotting four y-averaged quantities against z, see Fig. 44, the same quantities are plotted by Amarloo et al. The y-averaged streamwise velocity in Fig. 44a matches the LES perfectly, while the other three plots show a notable deviation between propagation and the LES. This is not surprising, as these include RMS values and/or the dispersive z-velocity. These quantities are more sensitive to small differences and thus more prone to errors, as also seen in the mesh independence study in Fig. 36.

In Fig. 44b and Fig. 44d, there is a large deviation at $z \approx 0$ (the wall). This deviation was also observed by Amarloo et al. for propagation of k-corrective-frozen RST. However, this deviation does not seem to influence the mean velocity field, which is the quantity one is usually interested in. Next, the discrepancy between propagation and LES is smaller than the one observed by Amarloo et al. for k-corrective-frozen RST. This is likely due to the difference in flow forcing; Amarloo et al. specify a force, while a bulk velocity (equal to the LES) is specified in the current work. This then gives a much better match of the $< U_x >$ profile in Fig. 44a, which likely also improves the other three plots. Finally, though a satisfactory propagated velocity is found, errors are significantly larger than for propagation of prior cases (such as the duct in Sec. 6.4). This larger error is speculated to come from the insufficient refinement near the smooth to rough wall transition discussed in Sec. 7.3.

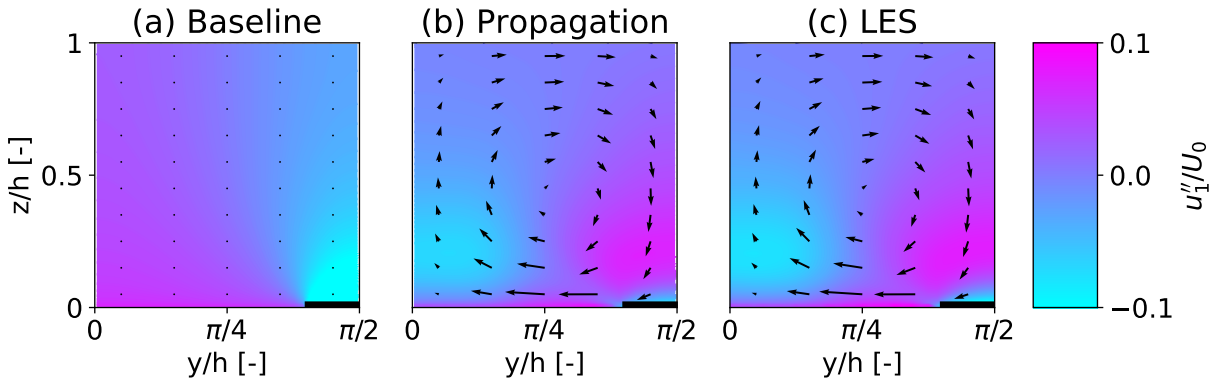
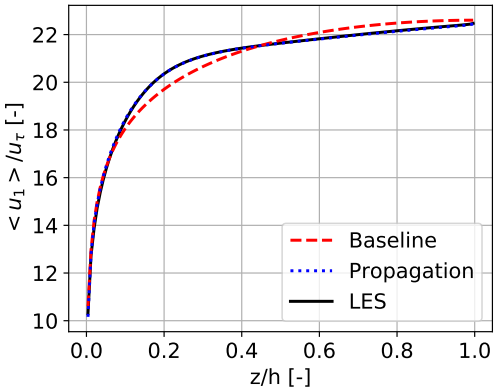
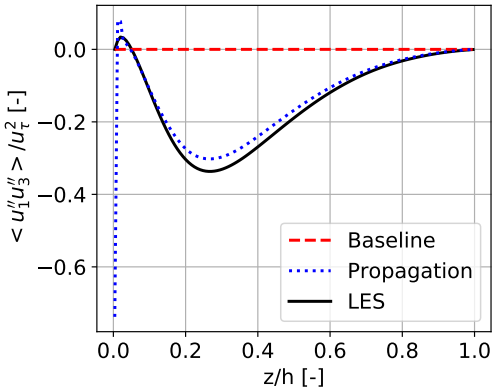


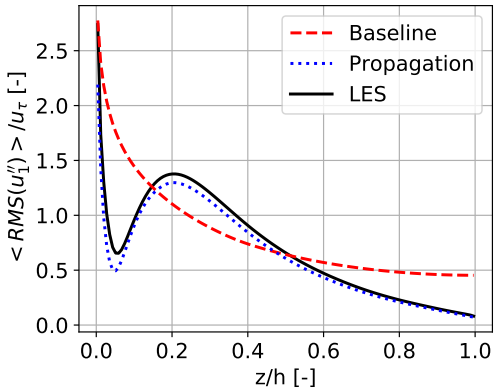
Figure 43: Contours of dispersive x-velocity (see Eq. 74) with the in plane velocity vector field overlaid, comparing between baseline (k-ω SST), propagation (k-ω SST with correction terms b_{ij}^{Δ} and R) and WMLES results, case hr00.



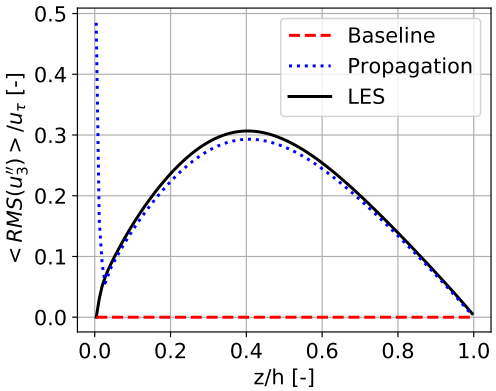
(a) y-Averaged streamwise velocity vs z



(b) y-Averaged product of dispersive x- and z-velocity vs z



(c) y-Averaged RMS dispersive streamwise velocity vs z



(d) y-Averaged RMS dispersive z-velocity vs z

Figure 44: Plots of various y-averaged quantities vs z, comparing between baseline (k-ω SST), propagation (k-ω SST with correction terms b_{ij}^{Δ} and R) and WMLES results, case hr00.

The frozen approach is applied to each roughness case in Tab. 11 followed by a propagation run to validate the correction fields. Propagation of each case converges under the settings in Tab. 15 except for case hr05. Though this case can likely be converged with different settings, it decided to leave it out of further tests given the abundance of other roughness cases. For the remaining cases, similar propagation performance as observed for case hr00 in Fig. 43 and Fig. 44 is observed, except for case hr14. Contour plots of dispersive x-velocity with the in-plane velocity vector field overlaid are shown for this case in Fig. 45. The propagation shows a definite improvement over the baseline in that the in-plane vortex is now predicted, roughly at the right location and with the right rotation direction. However, the strength of this vortex is significantly underpredicted by the propagation. Also the dispersive x-velocity is predicted better by the propagation, but still lacking in magnitude.

Next, the four y-averaged quantities plotted for case hr00 in Fig. 44 are also plotted for case hr14 in Fig. 46. For the plots where the dispersive velocity is plotted (Fig. 46b, Fig. 46c and Fig. 46d), the propagation shows a definite improvement over the baseline. However, the discrepancy between the propagation and the LES is much larger than for other cases. Now consider the y-averaged streamwise velocity in Fig. 46a; the propagation shows no improvement over the baseline, but both are already very close to the LES results. This is not surprising, as case hr14

has the smallest strip width and the lowest roughness heights (see Tab. 11), giving it the weakest secondary vortex. This makes the case almost equal to flat-plate flow, for which baseline k-ω SST already performs well (see Sec. 9). For all other cases, the secondary vortex is significantly stronger, giving a larger improvement of the propagation with respect to baseline and a better match with the LES results. Hence, case hr14 is excluded from further model training/testing.

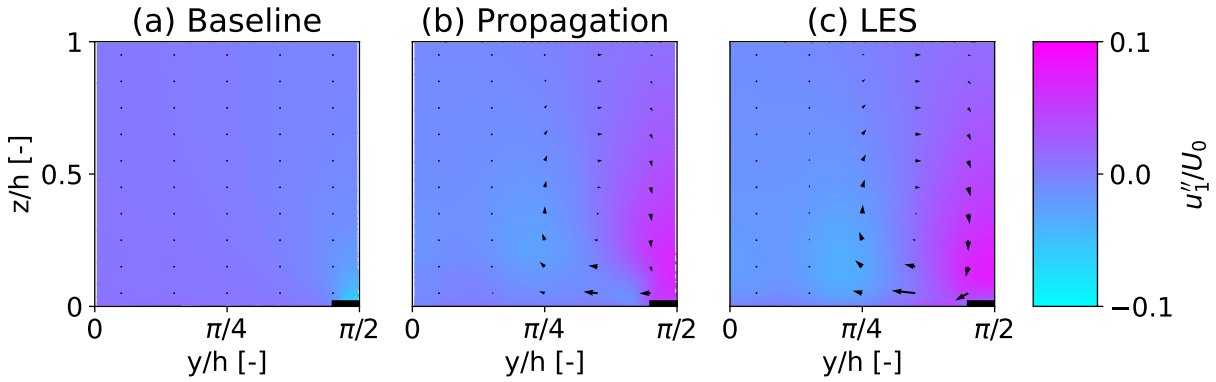
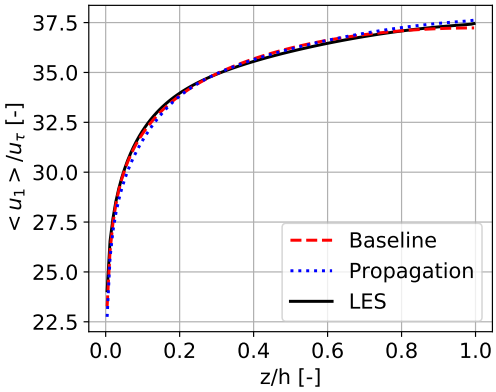
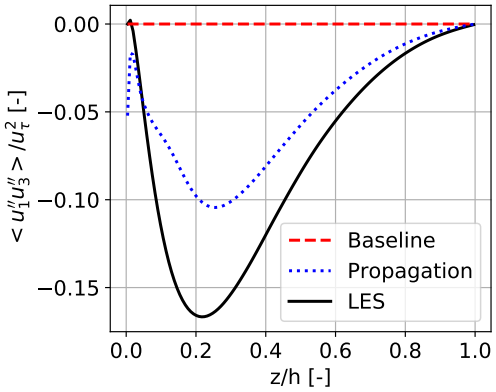


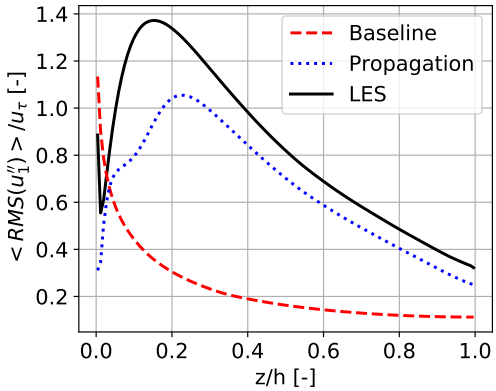
Figure 45: Contours of dispersive x-velocity (see Eq. 74) with the in plane velocity vector field overlaid, comparing between baseline (k-ω SST), propagation (k-ω SST with correction terms b_{ij}^{Δ} and R) and WMLES results, case hr14.



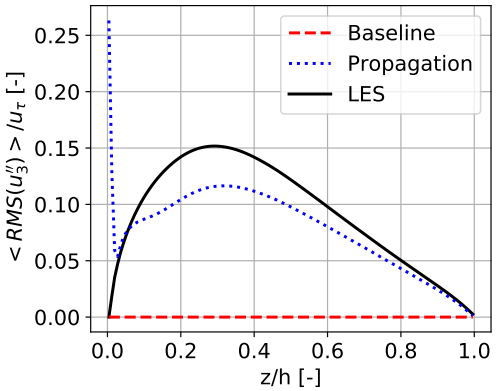
(a) y-Averaged streamwise velocity vs z



(b) y-Averaged product of dispersive x- and z-velocity vs z



(c) y-Averaged RMS dispersive streamwise velocity vs z



(d) y-Averaged RMS dispersive z-velocity vs z

Figure 46: Plots of various y-averaged quantities vs z, comparing between baseline (k-ω SST), propagation (k-ω SST with correction terms b_{ij}^{Δ} and R) and WMLES results, case hr14.

8 Channel flow case setup

The channel flow case is one of the cases in the collaborative testing challenge for data-driven turbulence modeling, part of NASA's 2022 symposium on turbulence modeling [42]. Two important changes are made with respect to their channel case: Firstly, an incompressible solver is used rather than a compressible one. This is because k-corrective-frozen currently only works for incompressible flows, and the case Mach number is sufficiently low to be considered incompressible (Mach 0.2). Secondly, an infinite domain length in flow direction is used with a specified bulk velocity, rather than a finite domain length with specified inlet total pressure and outlet static pressure. This is done to improve stability and make the run faster (1D mesh rather than 2D). The impact of these changes is evaluated later in the section by comparing new results with those from NASA. The schematic of the updated channel case is shown in Fig. 47, where the main flow is in $+x$-direction. The case can be described as two infinitely large plates at a finite distance h from eachother, with forced flow between them.

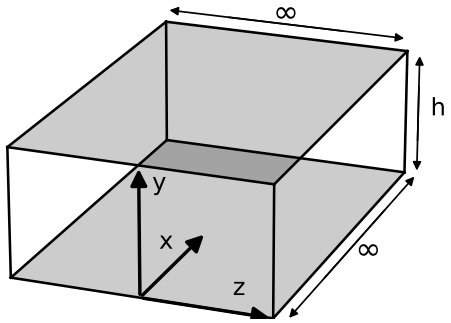


Figure 47: Schematic of the channel case, main flow in $+x$-direction.

The case of channel flow is well known and one for which most RANS models perform well. The flow is actually so simple that approximate analytical solutions exist which agree well with experiments [63, p. 424]. Thus, the goal of including this case is not to find corrections to the k-ω SST model to improve results. Rather, it is to ensure that discovered models retain the same performance as k-ω SST for simple cases. This 'do no harm' goal is also mentioned by Spalart during the 2022 symposium on turbulence modeling [45] and is the reason for including the case in the challenge. The section is structured as follows: A mesh is made of the domain in Sec. 8.1, based on a specified first cell height and cell growth ratio. Then in Sec. 8.2, the setup for a baseline k-ω SST run is given and case convergence is verified. Furthermore, a mesh independence study is performed with respect to both the first cell height and the cell growth ratio. Finally, the results are compared with NASA's results for further verification.

8.1 Channel mesh

In Fig. 47, there is a symmetry plane at $y = h/2$ which is exploited when generating the mesh; only the part from $y = 0$ till $y = h/2$ is meshed. The boundary types assigned to the six faces in this subdomain are shown in Fig. 48. Since the plates have an infinite length in z-direction, all gradients with respect to z are zero. Furthermore, since the flow is driven only in x-direction, U_z is also zero. Thus, all z-components of flow parameters can be ignored, which is done by specifying the empty condition for the two patches in the x-y-plane. This also means that only one cell is necessary in z-direction. Next, the cyclic condition is used to achieve the steady state solution at $x = \infty$. At steady state, gradients of flow parameters with respect to x are zero, meaning only one cell is needed in x-direction. Finally, the bottom plate is specified as a wall, where the velocity is zero. Thus, there is a velocity gradient in y-direction, meaning multiple cells are needed in y-direction. This means, the final mesh is 1-dimensional with cells in y-direction.

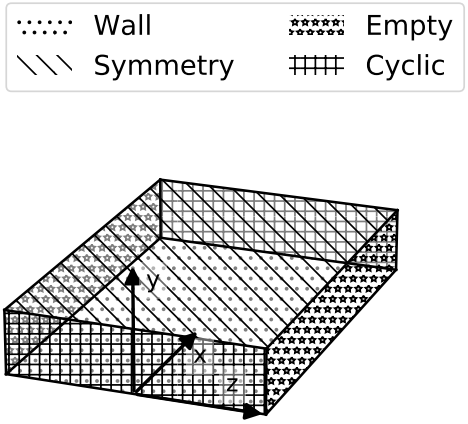


Figure 48: Boundary types used in the channel mesh.

There are two relevant parameters when generating the cells in y-direction; the nondimensional first cell-center height $y^+_{1/2}$ and the cell growth ratio β. The mesh is made using OpenFOAM's native `blockMesh` utility, which does not let one specify these directly. Thus, the relations laid out in Sec. 6.1 are used to convert $y^+_{1/2}$ and β to the appropriate inputs to `blockMesh` (specifically Eq. 72 and Eq. 73). Note that both equations depend on the dimensional first cell height δ_1 rather than the nondimensional first cell-center height $y^+_{1/2}$. In Sec. 6.1, $y^+_{1/2}$ is converted to δ_1 using Eq. 69, where the friction Reynolds number is given for each case. For the channel case, NASA provides the nondimensionalized friction velocity $u_\tau/U_\infty = 0.02655$ at the point where the flow is fully developed. Of course this quantity will be slightly different in the current run, but it is assumed to be close enough to provide a good estimation for $y^+_{1/2}$. This parameter is inserted into Eq. 67 to give a new formula for δ_1:

$$\delta_1 = \frac{2y^+_{1/2}\nu}{(u_\tau/U_\infty)U_\infty}. \tag{75}$$

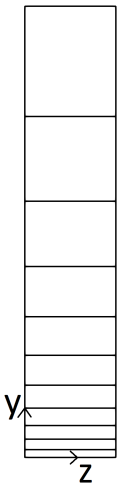


Figure 49: Channel mesh with $y^+_{1/2} = 10000$ and $\beta = 1.3$ for clear visualization.

Since all results are nondimensionalized and the solver is incompressible, the value of ν is not relevant, as long as the same value is used consistently. In this case, ν is set to $1.5 \times 10^{-5}\,\mathrm{m^2\,s^{-1}}$. Then one quantity remains unknown in Eq. 75; the free stream velocity U_∞. It is calculated based on the Reynolds number based on the channel height ($Re_h = U_\infty h/\nu$), which is specified as 8×10^7 by NASA. Like ν, h can be set to any consistent value, it is set to 100 m such that it gives a sensible freestream velocity of $12\,\mathrm{m\,s^{-1}}$. With this, all equations are closed and a mesh can be generated with specified $y^+_{1/2}$ and β. An example mesh is shown in Fig. 49 where $y^+_{1/2} = 10000$ and $\beta = 1.3$ are used for clear visualization.

8.2 Baseline setup and convergence study

OpenFOAM's `simpleFoam` solver (SIMPLE algorithm [7]) is again used to solve the incompressible Navier-Stokes equations, using the k-ω SST turbulence model. The boundary and initial conditions are given in Tab. 16. No wall function is used for k; this is observed to result in bad propagation for cases with small $y^+_{1/2}$. The initial conditions are chosen close to their expected value to give fast convergence. Next, since x goes to infinity, it is necessary to drive the flow, otherwise it would stop. A bulk velocity U_b is defined in fvOptions and the solver automatically calculates and applies the required pressure gradient to maintain this bulk velocity. To find the required bulk velocity, the NASA profile of Y versus u/U_∞ is integrated from $y/h = 0$ till $y/h = 1$. This is done by first constructing a cubic spline of the profile and then using scipy's quad function for integration. This gives a ratio $U_b/U_\infty = 0.966$, which is multiplied by $U_\infty = 12\,\mathrm{m\,s^{-1}}$ to give $U_b = 11.6\,\mathrm{m\,s^{-1}}$.

Table 16: Boundary- and initial conditions for the channel case.

Mesh part	U [$\mathrm{m\,s^{-1}}$]	p [$\mathrm{m^2\,s^{-2}}$]	k [$\mathrm{m^2\,s^{-2}}$]	ω [$\mathrm{s^{-1}}$]	ν_t [$\mathrm{m^2\,s^{-1}}$]
Inflow	cyclic				
Outflow					
WallBottom	noSlip	zeroGradient	fixedValue 0	omegaWall-Function	nutUSpalding-WallFunction
Left	Empty				
Right					
SymmetryTop	Symmetry				
Initial condition	[U_∞ 0 0]	0	0.02	10	0

The inner and outer residuals as well as the relaxation factors are given in Tab. 17. Since a large part of the wall is resolved, the range of values for ω is large, so its residuals are set to much lower values. As for the prior two cases, p and U_y are initialized as zero and remain zero, so their normalized residuals do not converge. Thus outer residuals are specified only for k and ω, where k has slightly stricter value than necessary to ensure U_x is also sufficiently converged. Finally, the relaxation factors give good stability and an acceptable convergence speed.

Table 17: Residuals and relaxation factors for the baseline channel case.

Parameter	Inner residual	Outer residual	Relaxation factor
U	10^{-8}	-	0.9
p			0.3
k		10^{-6}	0.8
ω	10^{-15}	10^{-10}	

A run is performed with the settings laid out in Tab. 16 and Tab. 17, using a mesh with $y^+_{1/2} = 0.1$ and $\beta = 1.1$. The residuals during this run are plotted against iteration in Fig. 50; after some startup, U_x, k and ω start converging in a straight line in the log-plot and reach their specified tolerance. The flow is also probed at various locations during the run, the evolution of k at these probe locations is plotted against iteration number in Fig. 51. At the end of the run, k has assumed a constant value, indicating the run has indeed converged, this is also true for U_x and ω.

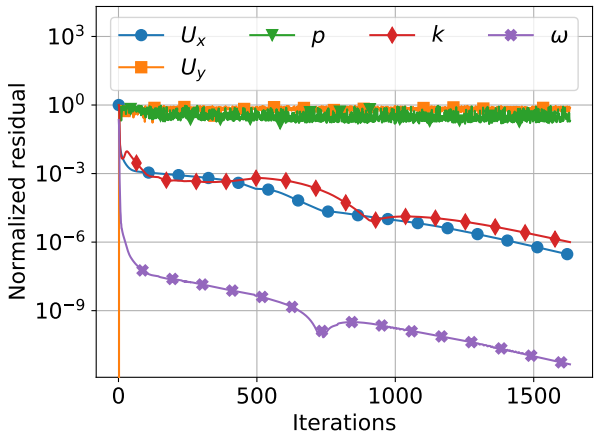


Figure 50: Outer residual versus iteration for various flow parameters, channel case with $y^+_{1/2} = 0.1$ and $\beta = 1.1$.

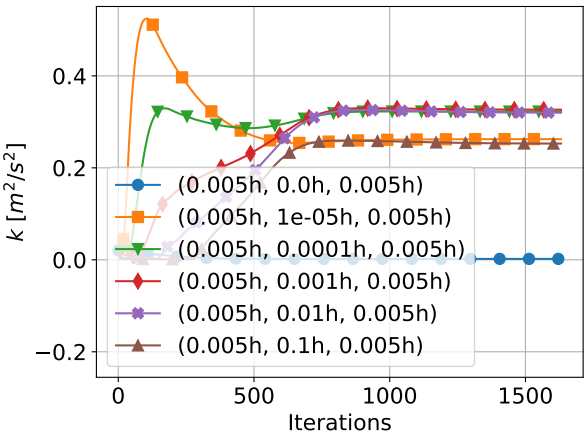


Figure 51: Turbulent kinetic energy versus iteration at various probe locations, channel case with $y^+_{1/2} = 0.1$ and $\beta = 1.1$.

In a number of figures provided by NASA for the same case, so-called + quantities appear, which are quantities nondimensionalized by the friction velocity u_τ. Firstly, y^+ is defined similarly to y^+_1 in Eq. 67, except it depends on the general distance y to the nearest wall, rather than the first cell height:

$$y^+ = \frac{y u_\tau}{\nu}, \tag{76}$$

where ν is the laminar kinematic viscosity. Furthermore, the friction velocity u_τ is defined as:

$$u_\tau = \sqrt{\frac{|\tau_{w,i}|}{\rho_w}}, \tag{77}$$

where $\tau_{w,i}$ is the wall shear stress vector; $\tau_{w,i}/\rho_w$ is attained by post-processing using the `wallShearStress` function. Next, u^+ is defined as:

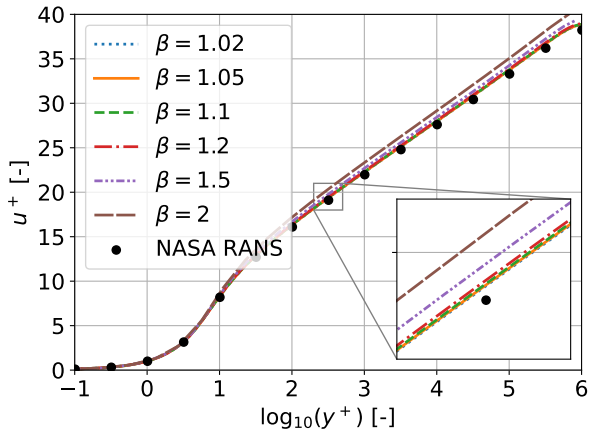
$$u^+ = \frac{u}{u_\tau}, \tag{78}$$

where u is the x-component of the velocity vector. Finally, k^+ is defined as:

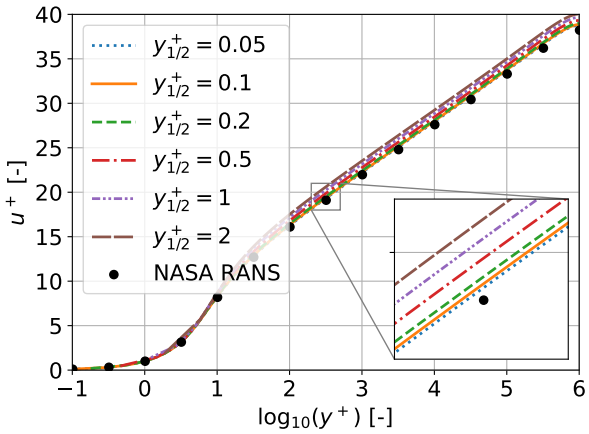
$$k^+ = \frac{k}{u_\tau^2}, \tag{79}$$

where k is the turbulent kinetic energy.

As explained in Sec. 6.2, it must also be proven that the solution obtained on the discretized mesh is sufficiently close to the continuous solution, which is done in a mesh independence study. The u^+ versus y^+ plot shows the clearest differences between meshes and is thus used for the mesh independence study. In Fig. 52a, u^+ is plotted against y^+ for various values of β while $y^+_{1/2}$ is fixed at 0.1. In Fig. 52b, u^+ is plotted against y^+ for various values of $y^+_{1/2}$ while β is fixed at 1.1. In both plots, the k-ω SST results obtained by NASA are also plotted for comparison. The mesh independent value of β is determined to be 1.1 while the mesh independent value of $y^+_{1/2}$ is determined to be 0.1. These converged values are on the strict side; this is such that mesh independence also holds if a certain propagated model has slightly higher gradients.



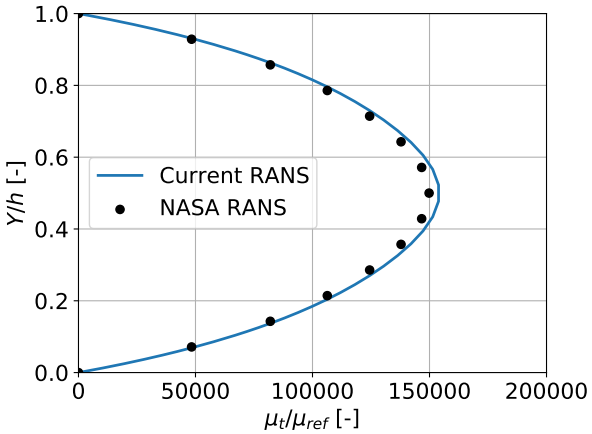
(a) Varying β, $y^+_{1/2} = 0.1$



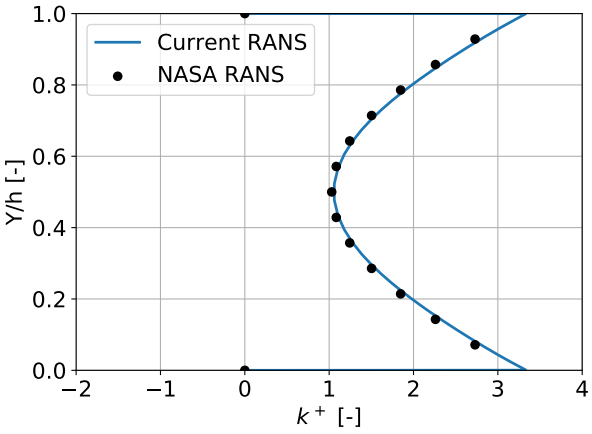
(b) Varying $y^+_{1/2}$, $\beta = 1.1$

Figure 52: Channel u^+ versus $\log_{10}(y^+)$, for meshes with various β and $y^+_{1/2}$, also including NASA's k-ω SST results for comparison.

Comparing between the converged result and the NASA RANS data in Fig. 52b, a slight difference still would remain at infinite mesh density. This is even more pronounced for the profile of ν_t, shown in Fig. 53a. The deviation for the profile of k^+, shown in Fig. 53b, is not as large as that of ν_t. Thus, k seems to be overestimated at the channel center while ω seems to be underestimated at the channel center with respect to the NASA RANS results. Still, the deviation of ν_t is at most $\sim 2.5\%$, which is attributed to solver differences and small differences in formulation of the k-ω SST model. Hence, the RANS run of the channel is considered verified. Since the NASA RANS shows good agreement with channel flow theory and the current run shows good agreement with the NASA RANS, plain k-ω SST already performs well for the case of channel flow. Thus, as mentioned at the beginning of this section, no correction fields need to be found as no corrections are needed.



(a) Y/h versus ν_t/ν



(b) Y/h versus k^+

Figure 53: Channel profiles comparing between the current RANS run and the reference run from NASA, $\beta = 1.1$ and $y^+_{1/2} = 0.1$.

9 Flat plate flow case setup

The flat plate case also comes from the collaborative testing challenge in NASA's 2022 symposium on turbulence modeling [42]. As for the channel case, an incompressible solver is used rather than a compressible one as used by NASA. This is because the k-corrective-frozen infrastructure currently only works for incompressible cases, and the case Mach number is sufficiently low to be considered incompressible (Mach 0.2). The schematic of the case is shown in Fig. 54, where the main flow is in $+x$-direction. Essentially, the case consists of uniform flow encountering an infinitely wide, infinitely long flat plate. A boundary layer will form on the plate, which grows with x. Profiles of this boundary layer are taken at $x = L$, so the domain length is $2L$ to remove any downstream influences at $x = L$. The domain z-height is theoretically also infinite, so the value L is chosen such that the top boundary has a negligible effect on the boundary layer, this is confirmed by NASA [42].

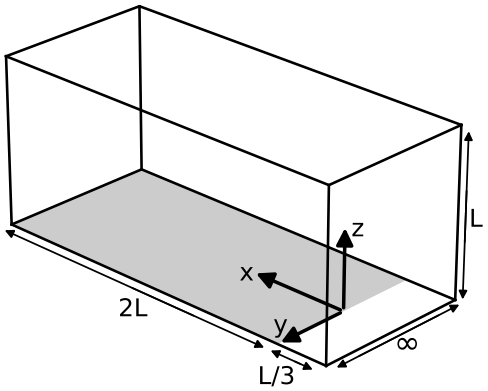


Figure 54: Schematic of the flat plate case, main flow in $+x$-direction.

Flat plate flow is a well known case, both experimentally and numerically and one for which most RANS models perform well. The flow is so simple that approximate analytical solutions exist which agree well with experiments [63, p. 430]. This is also true for the channel case in Sec. 8, where the 'do no harm' goal is mentioned. This goal is also the reason for including the flat plate case; newly found models should not negatively effect the results of this case. The section is structured as follows: In Sec. 9.1, the domain boundary types are given and the procedure to convert NASA meshes to OpenFOAM format is laid out. Then in Sec. 9.2, the setup of a baseline k-ω SST run is provided and the convergence of this case is verified. Furthermore, a mesh independence study is performed on the NASA meshes and the results are compared to the k-ω SST results from NASA for further verification.

9.1 Flat plate mesh

A mesh is made of the domain shown in Fig. 54, the boundary types that are assigned to the six faces in the domain (with the bottom face being split in two) are shown in Fig. 55. Since the plate has an infinite width in y-direction, gradients of all quantities will be zero in y-direction. Furthermore, the mean velocity is only in x-direction, so U_y will also be zero. Hence, all y-components of flow parameters can be ignored, which is implemented by specifying the empty condition for the two patches in the x-z-plane. This is also means that only one cell is needed in y-direction. As mentioned, uniform flow enters the domain at the inlet and develops into a boundary layer on the plate. As the flow is subsonic, there will be an upstream influence of the plate. In order to introduce this influence to the uniform flow, a small symmetry plane is defined before the plate where $U_x > 0\,\mathrm{m\,s^{-1}}$ but $U_z = 0\,\mathrm{m\,s^{-1}}$. A symmetry plane is also defined for the top of the domain, such that no flow enters the domain at the top, but the flow is allowed to have $U_x > 0\,\mathrm{m\,s^{-1}}$. Finally, the inlet and outlet of the domain are defined as patches, their exact boundary conditions are given in the next section, but the idea is to define the pressure/velocity so as to drive the flow.

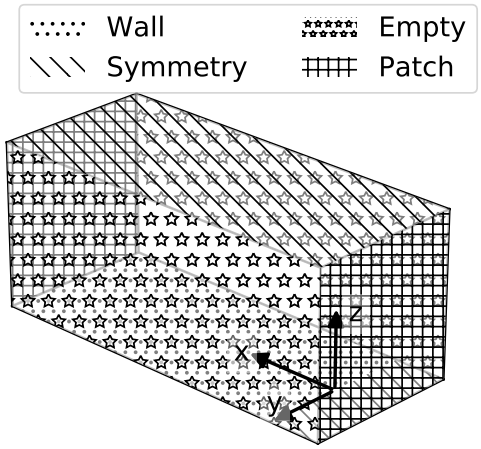


Figure 55: Boundary types used in the flat plate mesh.

Since the case comes directly from the collaborative testing challenge and the domain is the same (only the solver is different), the meshes used by NASA can also be used here. NASA provides a number of meshes with different levels of refinement, conveniently given as the number of mesh points on the plate. The number of points on the plate along the with corresponding total number of mesh cells is given for each mesh in Tab. 18. The mesh that is used for further testing is determined using a mesh independence study, which is laid out in the next section. For this mesh independence study, however, each mesh should be converted to OpenFOAM format, which is discussed next.

Table 18: Number of mesh points on the flat plate and the associated total number of cells of each mesh provided by NASA.

Plate points	Cells
29	816
57	3264
113	13056
225	52224
449	208896

The NASA meshes are provided in plot3d format, which can be converted to OpenFOAM format using the `plot3dToFoam` utility. Unfortunately, the plot3d files do not contain any patch information, so the patches have to be assigned again. The `autoPatch` utility is used to automatically split patches on 90 deg corners. This results in six patches; each domain face in Fig. 47 is assigned a different patch. Thus, the whole domain bottom is assigned to one patch, while it should be split in two. To address this, the bottom patch is first converted to a set using the `topoSet` utility. This set is then copied to produce two identical sets, both containing the bottom faces. Then the `topoSet` utility is used again, this time to delete all faces with $x > 0$ and $x < 0$

respectively. The resulting sets are converted back to patches; one with the bottom faces with $x > 0$ (the plate) and one with the bottom faces with $x < 0$ (the symmetry inflow). Finally, all patches are assigned the correct name and type (types given in Fig. 55) using the `createPatch` utility. A side view of the coarsest mesh (29 points) is shown in Fig. 56.

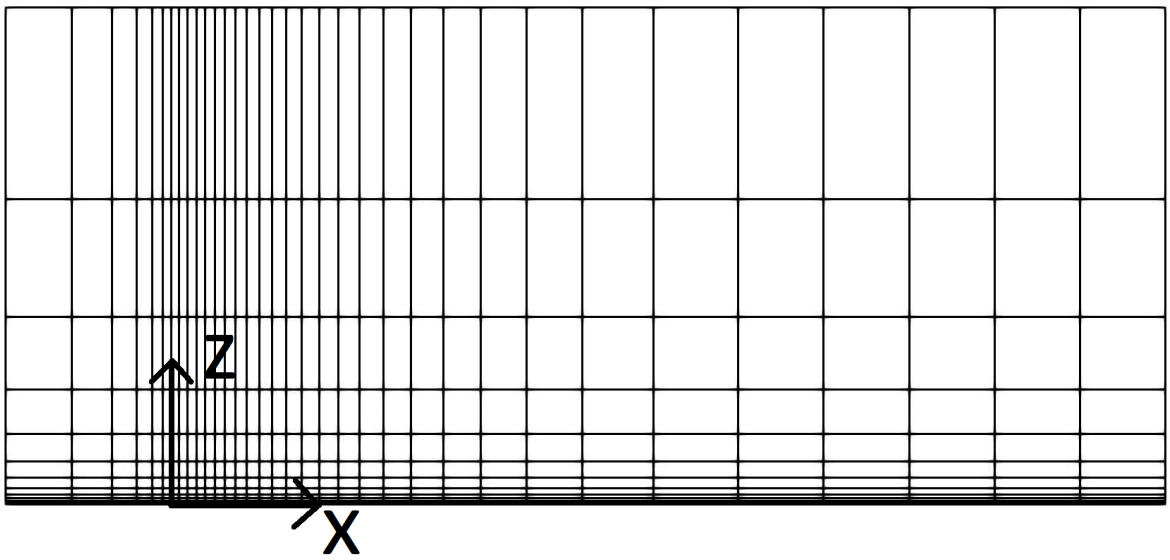


Figure 56: Coarsest flat plate mesh (29 plate points).

9.2 Baseline setup and convergence study

OpenFOAM's `simpleFoam` solver is used to solve the incompressible Navier-Stokes equations for this case, using the k-ω SST turbulence model. The consistent flag is set to true in fvSolution, meaning actually the SIMPLEC algorithm is used [11], this is done for stability. The boundary and initial conditions are given in Tab. 19. Note that a fixed velocity is specified at the inlet rather than a fixed total pressure, which is used by NASA. This change is made to improve stability, the impact is later evaluated by comparing to NASA's results. Also, no wall function is used for k, as this is found to give bad propagation results.

The freestream velocity U_∞ in Tab. 19 is calculated from the length based Reynolds number ($Re_L = U_\infty L/\nu$), specified as 5×10^6 by NASA. The length L is set to 1 m and the kinematic viscosity to 1.5×10^{-5} m^2 s^{-1}, since the solver is incompressible they can be set to any consistent value. The resulting freestream velocity is 75 m s^{-1}. The freestream turbulence intensity is specified as $I = 0.039\%$ by NASA and the freestream eddy viscosity $\nu_{t,\infty}$ as 0.009 m^2 s^{-1}. The freestream turbulent kinetic energy k_∞ is calculated using:

$$k_\infty = \frac{3}{2}(U_\infty \cdot I)^2, \tag{80}$$

giving $k_\infty = 0.0013\,\mathrm{m^2\,s^{-2}}$; the freestream ω is calculated to be $\omega_\infty = k_\infty/\nu_{t,\infty} = 0.14\,\mathrm{s^{-1}}$. These freestream values are also used as initial conditions for k and ω.

Table 19: Boundary- and initial conditions for the flat plate case.

Mesh part	U [m s^{-1}]	p [m^2 s^{-2}]	k [m^2 s^{-2}]	ω [s^{-1}]	ν_t [m^2 s^{-1}]
Inflow	fixedValue $[U_\infty$ 0 0]	zeroGradient	turbulent-Intensity-Kinetic-EnergyInlet 0.039%	fixedValue ω_∞	calculated
Outflow	zeroGradient	fixedValue 0	zeroGradient		
WallBottom	noSlip	zeroGradient	fixedValue 0	omegaWall-Function	nutUSpalding-WallFunction
Left	Empty				
Right					
Symmetry-Bottom	Symmetry				
SymmetryTop					
Initial condition	$[U_\infty$ 0 0]	0	k_∞	ω_∞	$\nu_{t,\infty}$

The inner and outer residuals together with the relaxation factors are given in Tab. 20. Lower residuals are used for ω since, especially for the fine meshes, a large part of the wall is resolved, resulting in ω varying over many orders of magnitude. Contrary to the prior baseline cases, both p and U_z converge to a nonzero value during the run, meaning outer residuals can be specified for all flow parameters, 10^{-6} proves to be necessary for full convergence of the finest mesh. The relaxation factors are chosen to give stable, yet fast convergence.

Table 20: Residuals and relaxation factors for the baseline flat plate case.

Parameter	Inner residual	Outer residual	Relaxation factor
U	10^{-8}	10^{-6}	0.9
p			0.7
k			0.7
ω	10^{-15}	10^{-10}	0.95

In order to verify that the residuals and relaxation factors listed in Tab. 20 actually lead to a converged solution, the evolution of the outer residuals is plotted against iteration for the 225 point mesh in Fig. 57. After some initial startup, all residuals start converging in an approximately straight line in the log-plot and reach their specified tolerances. Next, the flow is probed at various locations and the evolution of a certain flow parameter is plotted against iteration. The most critical parameter turns out to be k; its probed value is plotted against iteration for the 225 point mesh in Fig. 58. At the end of the run, k has assumed a constant value at each probe, indicating the solution has converged.

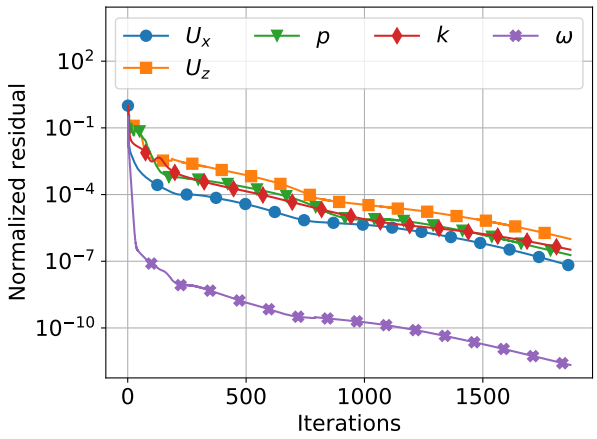


Figure 57: Outer residual versus iteration for various flow parameters, flat plate case with 225 point mesh.

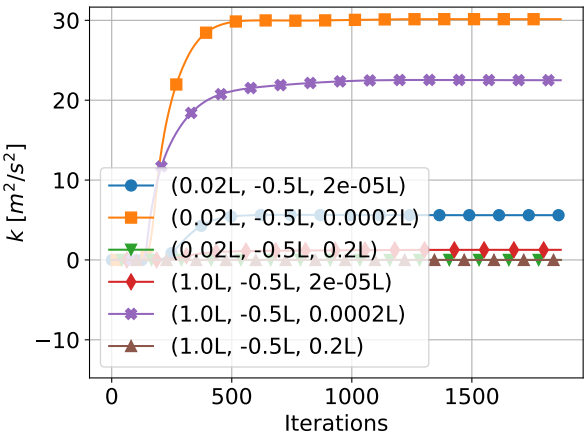


Figure 58: Turbulent kinetic energy versus iteration at various probe locations, flat plate case with 225 point mesh.

For comparison with the NASA data, some derived quantities need to be calculated. Firstly, the skin friction coefficient C_f is calculated along the plate using:

$$C_f = \frac{\tau_{w,i}\hat{w}_{||,i}}{\frac{1}{2}\rho_\infty U_\infty^2}. \tag{81}$$

Here, $\tau_{w,i}$ is the wall shear stress vector; $\tau_{w,i}/\rho_\infty$ is obtained by post-processing using the `wallShearStress` function. Next, $\hat{w}_{||,i}$ is the wall tangential unit vector, equal to $[\ 1 \quad 0 \quad 0\ ]^T$, so $\tau_{w,i}\hat{w}_{||,i}/\rho_\infty$ is simply the x-component of the output of the `wallShearStress` function. Furthermore, U_∞ was calculated to be $75\,\mathrm{m\,s^{-1}}$ at the beginning of this section. Secondly, the Reynolds number based on the momentum thickness, Re_θ, is calculated as:

$$Re_\theta = \frac{U_\infty \theta}{\nu}. \tag{82}$$

Here, ν is the laminar kinematic viscosity and θ is the momentum thickness which is defined as:

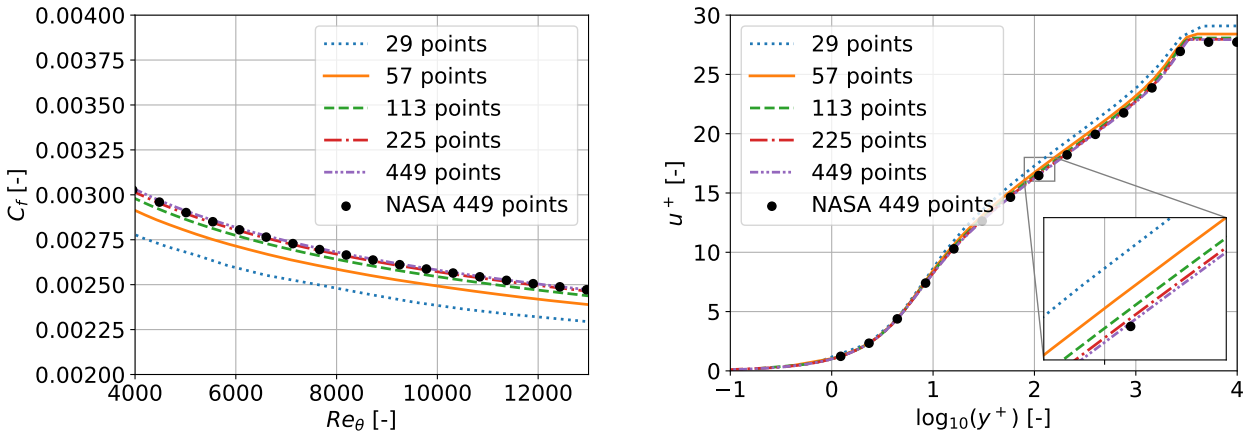
$$\theta = \int_0^\infty \frac{\rho}{\rho_\infty}\frac{u}{U_\infty}\left(1-\frac{u}{U_\infty}\right)dz, \tag{83}$$

where u is the x-component of the velocity vector, which is available in each mesh cell. Furthermore, $\rho/\rho_\infty = 1$ since a constant density solver is used.

The integral in Eq. 83 is in wall normal direction. Evaluating this integral is simplified by the orthogonality of the mesh; there are 'stacks' of cells with the same x-coordinate, but different z (wall normal) coordinates. Note that most of this stack of cells is outside the boundary layer, meaning $u \approx U_\infty$, leading to numerical cancellation errors, making the integral inaccurate. To mitigate these errors, integration is only performed up to the z-coordinate at which u first reaches 99.9% of U_∞. In addition to the cell centers within this region, two points are added to the numerical integral: one at $z = 0$, with $u = 0\,\mathrm{m\,s^{-1}}$ due to the no-slip condition. The other one at the edge of the boundary layer, having $u = 0.999U_\infty$, where its z-coordinate is found using linear interpolation between the two nearest cells. Finally, the integral is found by first constructing a cubic spline of the points and then integrating it using numerical quadrature. This yields a θ value at each wall face.

As explained in Sec. 6.2, a mesh independence study shall be performed to prove that the solution on the discretized mesh is sufficiently close to the continuous solution. The study is performed on the meshes provided by NASA, laid out in Tab. 18. The two figures for which NASA gives their own k-ω SST results are also used in the mesh independence study and NASA's results are added for comparison. Firstly, C_f is plotted against Re_θ in Fig. 59a, where the 225 point mesh is deemed close enough to the 449 point mesh to be considered mesh independent. Secondly, u^+ is plotted against y^+ in Fig. 59b (both defined in Sec. 8.2), where again the 225 point mesh is deemed close enough to the 449 point mesh to be considered mesh independent. Hence, the 225 point mesh is used for further testing.

In both Fig. 59a and Fig. 59b, the 225 point and the 449 point mesh are right on top of the NASA results (449 closer as it is the same mesh). This further verifies the setup of the current run. Also, the NASA results are right on top of analytical results for flat plate flow. Hence, the current run is close to analytical flat plate results, meaning the k-ω SST model in OpenFOAM already performs well and no corrections are needed.



(a) C_f versus Re_θ

(b) u^+ versus y^+ at $Re_\theta = 10000$

Figure 59: Results of various meshes with increasing refinement, plotted together with results from NASA for comparison, flat plate case.

10 Wall-mounted hump case setup

The wall-mounted hump case is part of the collaborative testing challenge, which is part of NASA's 2022 symposium on turbulence modeling [42]. As for the other symposium cases, an incompressible solver is used rather than a compressible one as used by NASA. Again, the k-corrective-frozen infrastructure is currently only written for incompressible cases, and the incoming Mach number of the case is only Mach 0.1, meaning the maximum velocity is not expected to exceed Mach 0.2. The schematic of the hump is shown in Fig. 60, where the main flow is in $+x$-direction. The case is based on an experiment performed as part of the CFDVAL2004 workshop [44], the experiment is further described by Greenblatt et al. [18]. To account for end-plate blockage in this experiment, the mesh height decreases slightly at the start of the hump. The length of the hump is referred to as the hump chord c. The length of the inlet and outlet (L_{inlet} and L_{outlet} respectively) are mesh dependent, as will be explained later in the section.

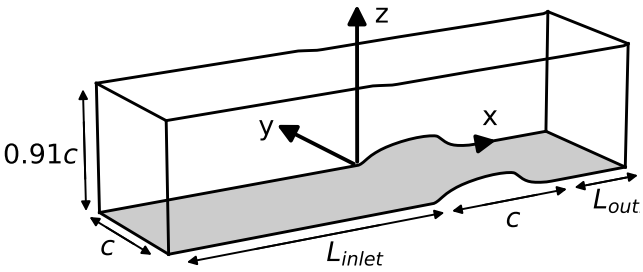


Figure 60: Schematic of the wall-mounted hump, main flow in $+x$-direction.

In the aforementioned experiment by Greenblatt et al., PIV measurements are also performed of the hump. However, these do not cover the whole domain and are thus not suitable for training correction fields using the k-corrective-frozen approach described in Sec. 3.3.3. Uzun et al. performed a wall-resolved LES of the same case, which is used for training correction fields instead [56]. Unfortunately, the inlet and outlet length of their LES domain are much smaller than those of the RANS domain of the meshes provided by NASA. Hence, training and validation of the correction fields are performed on the reduced LES domain, while model testing is performed on the full NASA RANS domain.

The section is structured as follows: In Sec. 10.1, the used boundary types of the domain are given and the procedure to convert NASA meshes to OpenFOAM format is laid out. Next, in Sec. 10.2, the procedure to coarsen the LES mesh as well as the LES data is laid out. Then, the setup of baseline k-ω SST runs for both the RANS domain and the LES domain meshes is given in Sec. 10.3 and Sec. 10.4 respectively. In each section, case convergence is verified, a mesh independence study is performed and results are compared to k-ω SST results from NASA for verification and LES results for identification of model errors. The discrepancies between k-ω SST and LES are addressed by training correction fields using k-corrective-frozen on the LES domain, this is laid out in Sec. 10.5. Finally, in Sec. 10.6, the correction fields are injected into the k-ω SST turbulence model and run in a full RANS solver on the LES domain for validation of these fields (propagation).

10.1 Hump RANS domain meshes

The boundary types used for both the RANS and LES domain meshes are given in Fig. 61. The bottom is modeled as a wall, while the top is modeled as a slip wall. The inlet and outlet are defined as patches, their exact boundary conditions are given in Sec. 10.3 for the RANS domain and Sec. 10.4 for the LES domain. The general idea is to specify a pressure/velocity at the inlet/outlet to drive the flow through the domain. Finally, side plates are used in the experiment, so one would expect a similar condition for the sides here. However, such a condition proves too expensive for the LES, which is the data that is to be matched. Instead, the LES uses periodic boundaries in y-direction and modifies the top boundary to account for side-plate blockage (which is also done for the RANS domain). Hence, the LES actually models an infinitely wide hump, which implies zero gradients in y-direction for averaged quantities. Thus, the empty condition is used for the two sides in the x-z-plane for the current RANS runs.

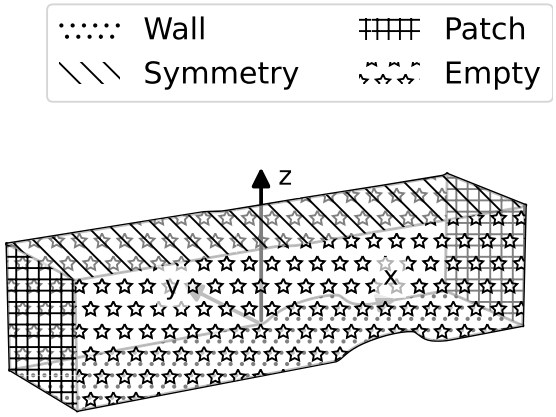


Figure 61: Boundary types used in the hump meshes.

The RANS domain meshes are provided by NASA on the challenge website [42], they are characterized by $L_{inlet} = 6.39c$ and $L_{outlet} = 3c$ (see Fig. 60). These meshes are in plot3d format, they are converted to OpenFOAM format using the `plot3dToFoam` utility. The meshes provided by NASA are nondimensional, meaning they use a chord length of 1 m. In the current work, the same chord length as the experiment is used; 0.42 m. Hence, during the conversion to OpenFOAM, the scale argument is set to 0.42. Next, since the plot3d files only contain point coordinates, the various patches have to be assigned again. This is done straightforwardly using the `autoPatch` utility, which already separates all patches since they are at a 90 deg angle with respect to eachother. Then, each patch is assigned the correct name and type using the `createPatch` utility. NASA provides five meshes at different levels of refinement, the number of streamwise points and the associated total number of cells is given for each mesh in Tab. 21. The coarsest mesh (103 points) is shown in Fig. 62.

Table 21: Number of streamwise points on the RANS domain hump mesh and the associated total number of cells.

Streamwise points	Cells
103	2754
205	11016
409	44064
817	176256
1633	705024

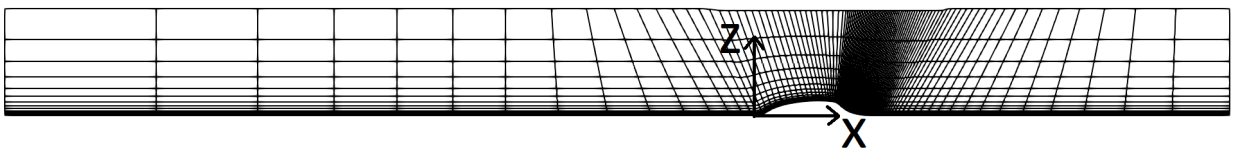


Figure 62: Coarsest RANS domain hump mesh (103 points).

10.2 Hump LES domain meshes

The meshes of the LES domain are based directly on the mesh used for the LES by Uzun et al. [56]. This mesh is similar to the RANS domain mesh, using the same boundary types shown in Fig. 61. The only difference is that it has a much shorter inlet and outlet length ($L_{inlet} = 2.03c$; $L_{outlet} = 1.6c$). Also, it uses two regions; one region near the wall and one region further from the wall, up to the top of the domain. The near wall region has twice as many streamwise cells as the far-wall region. These separate regions exist to be able to increase near wall accuracy without massively increasing the total number of cells. A blending function is used to couple these regions in the solver. Given the more lenient refinement requirements and lower computational cost of RANS, this two-region approach is not needed for the RANS run on this mesh. Hence, half the streamwise points of the near-wall region are removed and it is stitched to the far-wall region to form one coherent mesh.

Even this mesh is likely much finer than what is required for a RANS run, so it is coarsened further. This is done by defining a coarsening factor and merging this number of cells in each direction. For instance, for a coarsening factor of 3, 3 consecutive cells in streamwise direction and 3 consecutive cells in wall-normal direction are merged, giving a 3×3 square of cells that is merged into one new cell. Thus, the total number of cells will reduce by the coarsening factor squared. Of course, the number of streamwise/wall-normal cells will not always be exactly divisible by the coarsening factor. To address this, a smaller number of cells is merged at the inlet and top of the domain, such that the remaining number of streamwise/wall-normal cells is exactly divisible by the coarsening factor. Naturally, this leads to a jump in cell size, but as the inlet/top cells are finer than their neighbouring cells and they are far from the hump, no impact is expected on the results. If there is a significant impact, this will become clear in the mesh independence study performed in Sec. 10.4.4.

Once a mesh has been coarsened, its x- and z-coordinates are available as 2D arrays. To convert these to a 3D OpenFOAM mesh, they are first converted to a plot3d file, which is created using a custom Python function, given in Appendix C (`convertToPlot3d()`). This function converts the mesh to 3D by placing the 2D mesh points at $y = 0$ and $y = -c$, giving one cell thickness in y-direction. The mesh in the plot3d file is then converted to OpenFOAM using the `plot3dToFoam` utility. Similar to the RANS domain mesh, patches are then created and assigned the right name and type using the `autoPatch` and `createPatch` utility respectively. Four OpenFOAM meshes are created using this procedure; one with no coarsening, the others with coarsening factors 2, 4 and 8. The total number of cells for each of these meshes is given in Tab. 22. For better visualization, a mesh with coarsening factor 32 is made, which is shown in Fig. 63.

Table 22: Coarsening factor for the LES domain hump mesh and the associated total number of cells.

Coarsening factor	Cells
1	825020
2	206338
4	51626
8	13062

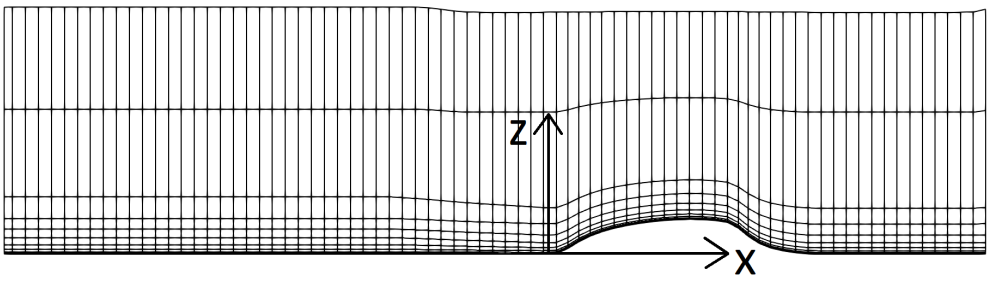


Figure 63: LES domain hump mesh with coarsening factor 32.

10.3 Hump RANS domain setup and convergence study

OpenFOAM cases are set up for the hump RANS domain meshes of increasing refinement described in Sec. 10.1. The boundary and initial conditions are described in Sec. 10.3.1 together with solver settings. Furthermore, these settings are verified to lead to a converged solution in this section. Next, the calculation of derived variables such as the pressure coefficient is laid out in Sec. 10.3.2. Finally, in Sec. 10.3.3, a mesh independence study is performed to determine a suitable mesh refinement. Also, results are compared to NASA's RANS results for verification and to LES results for validation (identifying model inadequacies).

10.3.1 Initial and boundary conditions, solver settings and run convergence

OpenFOAM's `simpleFoam` solver is again used to solve the incompressible Navier-Stokes equations for this case, using the k-ω SST turbulence model. As for the flat plate, the consistent flag is set to true such that actually the SIMPLEC algorithm is used, which is needed for stability. The boundary and initial conditions are given in Tab. 23, they are mostly adapted from the k-ω SST setup of the NASA challenge [42]. The reference velocity U_{ref} is calculated using:

$$U_{ref} = M_{ref}\sqrt{\gamma R T_{ref}}, \tag{84}$$

where $M_{ref} = 0.1$ is the reference Mach number and $T_{ref} = 298.3\,\mathrm{K}$ is the reference temperature. Furthermore, $\gamma = 1.4$ is the heat capacity ratio and $R = 287.05\,\mathrm{m^2 s^{-2} K^{-1}}$ is the specific gas constant (both for air). This gives $U_{ref} = 34.6\,\mathrm{m\,s^{-1}}$, which is also found by NASA.

Table 23: Boundary- and initial conditions for the hump case on the RANS domain.

Mesh part	U [m s^{-1}]	p [m^2 s^{-2}]	k [m^2 s^{-2}]	ω [s^{-1}]	ν_t [m^2 s^{-1}]
Inflow	fixedValue $[U_{ref}\ 0\ 0]$	zeroGradient	turbulent-Intensity-Kinetic-EnergyInlet 0.077%	fixedValue ω_{ref}	calculated
Outflow	zeroGradient	fixedValue $-0.00038\frac{p_{ref}}{\rho}$	zeroGradient		
WallBottom	noSlip	zeroGradient	kqRWall-Function	omegaWall-Function	nutUSpalding-WallFunction
Left	Empty				
Right					
SymmetryTop	Symmetry				
Initial condition	[1 0 1]	0	$1000k_{ref}$	$1000\omega_{ref}$	$\nu_{t,ref}$

The laminar kinematic viscosity ν is found from the chord-based Reynolds number, given by NASA as $Re_c = 9.36 \times 10^5$, the expression is as follows:

$$\nu = \frac{U_{ref}\, c}{Re_c}. \tag{85}$$

The chord length c is set the same as in the experiment; $c = 0.420\,\text{m}$, resulting in a laminar kinematic viscosity $\nu = 1.55 \times 10^{-5}\ \text{m}^2\,\text{s}^{-1}$. At the inflow, the freestream turbulence intensity is specified by NASA as 0.077% and the reference eddy viscosity as $\nu_{t,ref} = 0.009\,\text{m}^2\,\text{s}^{-1}$. The reference turbulent kinetic energy k_{ref} is calculated as $0.001\,07\,\text{m}^2\,\text{s}^{-2}$ using Eq. 80; the reference ω is then $\omega_{ref} = k_{ref}/\nu_{t,ref} = 0.118\,\text{s}^{-1}$. NASA specifies the turbulence intensity and the eddy viscosity at the inlet, which is not possible in OpenFOAM as ω also needs to be specified. Since velocity varies over the inlet, k varies as well, meaning ω would need to vary to keep a constant ν_t at the inlet. This is rather difficult to implement, especially due to the ν_t limiter in k-ω SST. Thus, ω is simply set to ω_{ref} over the whole inlet, which is justified by the fact that the boundary layer is still small there. For further verification of this inlet condition, results are compared to those from NASA later in this section.

Since NASA uses a compressible solver, they use the actual pressure in their run. However, for the current incompressible run, a reference pressure of zero is desired; both for consistency with other cases and easier post-processing. NASA specifies the total pressure at the inlet as $p_{t,inlet} = 1.007 p_{ref}$ and the static pressure at the outlet as $p_{outlet} = 0.99962 p_{ref}$, so setting $p_{ref} = 0$ would result in stationary flow. In order to be able to use $p_{ref} = 0$, the difference between p_{ref} and the inlet/outlet (total) pressure has to be found. Actually, the pressure specified in an incompressible solver is divided by the density, so these absolute differences should be divided by the density. The resulting expression for the inlet total pressure difference is:

$$\frac{p_{t,inlet} - p_{ref,0}}{\rho} = 0.007\frac{p_{ref,NASA}}{\rho}; \tag{86}$$

the expression for the outlet static pressure difference is:

$$\frac{p_{outlet} - p_{ref,0}}{\rho} = -0.00038 \frac{p_{ref,NASA}}{\rho}. \tag{87}$$

These expressions require the reference pressure used by NASA divided by the density to be known. The perfect gas law is rewritten to yield an expression for $p_{ref,NASA}/\rho$:

$$\frac{p_{ref,NASA}}{\rho} = RT_{ref}, \tag{88}$$

$p_{ref,NASA}/\rho$ is found to be 8.56×10^4 $\mathrm{m^2\,s^{-2}}$. This gives $(p_{t,inlet} - p_{ref,0})/\rho = 599\,\mathrm{m^2\,s^{-2}}$ and $(p_{outlet} - p_{ref,0})/\rho = -32.5\,\mathrm{m^2\,s^{-2}}$.

The inner and outer residuals as well as the relaxation factors are given in Tab. 24. Again, lower residuals are used for ω since the finest meshes resolve a large part of the wall, giving an order of magnitude variation in ω. Also, for the other variables an outer residual tolerance of 10^{-6} is necessary for convergence. Finally, the relaxation factors give fast convergence while the solution remains stable. Actually, stability turns out to be a significant issue for this case. This is why different initial conditions are used (see Tab. 23) compared to other cases. Starting with a lower velocity and larger k and ω gives more stability and faster convergence. Also velocity is specified at the inlet rather than total pressure (which is used by NASA) to boost stability. To further mitigate the stability issues, velocity magnitude is limited to $70\,\mathrm{m\,s^{-1}}$, a minimum ω of $0.1\,\mathrm{s^{-1}}$ is enforced and one non-orthogonal corrector step is used. Even then, the 817 point mesh fails to converge, so its U relaxation factor is decreased to 0.8.

Table 24: Residuals and relaxation factors for the baseline hump case on the RANS domain (except 817 point mesh).

Parameter	Inner residual	Outer residual	Relaxation factor
U	10^{-8}	10^{-6}	0.95
p			1
k			0.7
ω	10^{-15}	10^{-10}	

In order to verify that the settings laid out above lead to convergence, the outer residuals are plotted against iteration for the 409 point mesh in Fig. 64. After some startup, all residuals go down in a straight line in the log-plot and reach their specified tolerance. Next, the flow is probed at various locations to monitor the evolution with iteration. The most critical parameter for convergence turns out to be k, its evolution is plotted for the 409 point mesh in Fig. 65. At the end of the run, k has reached a constant value at each probe, further verifying convergence. Convergence is also verified for the other meshes using the same check.

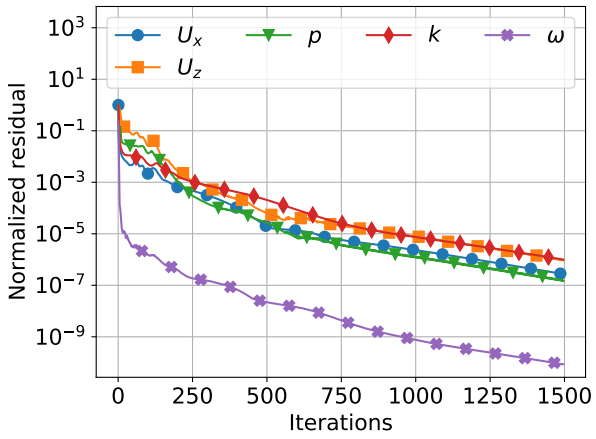


Figure 64: Outer residual versus iteration for various flow parameters, RANS domain hump case with 409 point mesh.

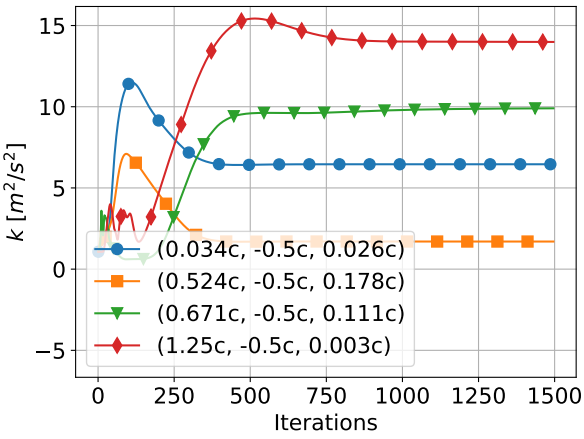


Figure 65: Turbulent kinetic energy versus iteration at various probe locations, RANS domain hump case with 409 point mesh.

10.3.2 Calculation of derived variables

For further analysis, some derived variables are needed, both from the OpenFOAM RANS run and the LES data, their calculation is laid out next. Firstly, the pressure coefficient C_p is needed, defined as:

$$C_p = \frac{p - p_{ref}}{\frac{1}{2}\rho U_{ref}^2}. \tag{89}$$

For the OpenFOAM RANS run, this equation can be directly used as $p_{ref} = 0$ and the solver directly outputs p/ρ. For the LES data, p_{ref} is nonzero and pressure is provided as p/p_{ref}. The expression for C_p is rewritten and combined with the perfect gas law in Eq. 88 to attain the following expression for C_p:

$$C_p = \frac{p_{ref}}{\rho}\left(\frac{p}{p_{ref}} - 1\right)\frac{1}{U_{ref}^2} = \frac{RT_{ref}}{U_{ref}^2}\left(\frac{p}{p_{ref}} - 1\right). \tag{90}$$

Secondly, the friction coefficient C_f is to be obtained along the wall, its expression is provided in Eq. 81. For the OpenFOAM RANS run, $\tau_{w,i}/\rho_\infty$ can be obtained by post-processing using the `wallShearStress` function and U_∞ is known ($34.6\,\mathrm{m\,s^{-1}}$). The wall tangential unit vector $\hat{w}_{||,i}$ varies along the wall; for the kth wall panel, it is equal to $[\ dx_k \quad 0 \quad dz_k\ ]^T$. To obtain dx and dz for each wall panel, the corner points are needed. To this end, the `topoSet` utility is used to convert the wall patch to a faceSet. This is then converted to an ASCII VTK file using the `foamToVTK` utility. Then, x- and z-coordinates are extracted at one y-location from this ASCII VTK file and sorted by x. These sorted point coordinates are then used to obtain arrays with dx_k and dz_k

For the LES data, $|\tau_{w,i}|$ has to be calculated using its definition:

$$\frac{|\tau_{w,i}|}{\rho_\infty} = \nu \left. \frac{\partial u_{wallperpendicular}}{\partial x_{wallnormal}} \right|_{wall}, \tag{91}$$

which already has the correct sign. For this equation, a velocity derivative in wall normal coordinates is needed, meaning the velocity gradient tensor has to be transformed from global to local wall-normal coordinates. The transformation is purely rotational and is characterized by the angle ϕ defined counter-clockwise positive, it is visualized in Fig. 66. To find each ϕ along the wall, the angle between each wall panel and the global x-axis is calculated using a simple arctangent. The 2D rotation matrix Q_{ik} used to transform a point from the global to the local wall-normal coordinate system is as follows:

$$Q_{ik} = \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}. \tag{92}$$

Then, the velocity gradient tensor is transformed to wall-normal coordinates as follows:

$$\nabla_i u_j \big|_{wall} = Q_{ik} (\nabla_k u_l) Q_{jl}. \tag{93}$$

The velocity derivative in Eq. 91 is simply the $\partial u/\partial y$ component of this transformed tensor. This transformation is applied to each wall panel to find C_f.

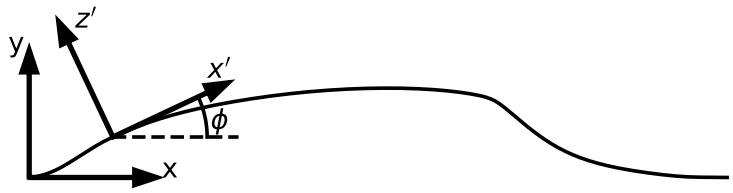


Figure 66: Definition of rotated wall-normal coordinate system.

The third and final derived quantity is the $\langle u'w' \rangle$ component of the Reynolds stress tensor. This quantity is directly available in the LES data, however, it needs to be calculated for the OpenFOAM RANS run. In the k-ω SST turbulence model, the Reynolds stress tensor is approximated using Boussinesq's eddy viscosity hypothesis given in Eq. 14; the equation for the $\langle u'w' \rangle$ component is:

$$\langle u'w' \rangle = -\nu_t \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right). \tag{94}$$

Both the eddy viscosity ν_t and the velocity derivatives are available in the OpenFOAM RANS run.

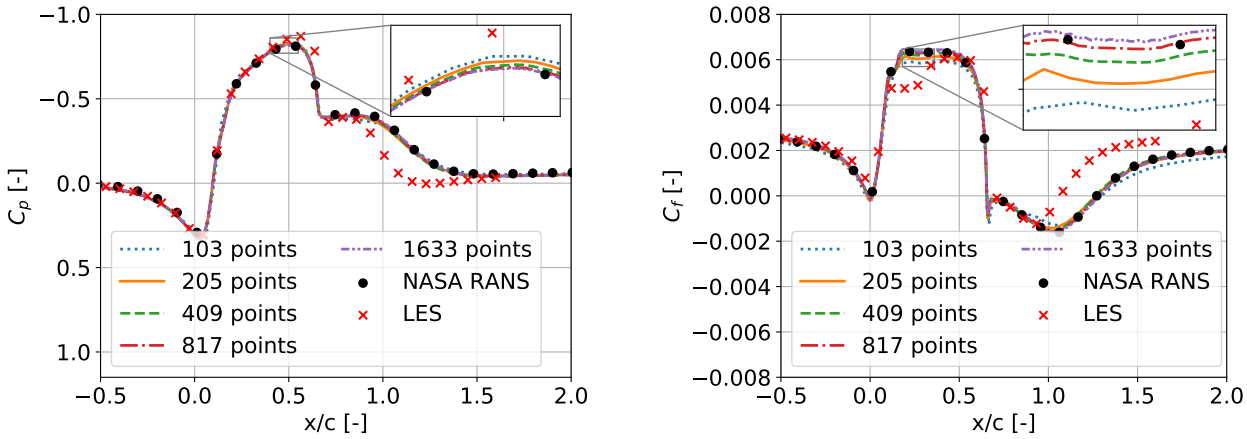
10.3.3 Mesh independence study and validation

As described in Sec. 10.1, NASA provides five meshes of increasing refinement of the hump RANS domain. As for the other cases, a mesh independence study is performed to find the coarsest mesh that gives sufficiently accurate results. The pressure coefficient C_p and the friction coefficient C_f are plotted along the hump wall in Fig. 67a and Fig. 67b respectively, NASA's RANS results and LES results are added for comparison. For the C_p plot, all mesh results are almost exactly on top of eachother, the discrepancy with respect to the LES data is much larger than the discretization error. For the C_f plot, the difference between meshes is somewhat larger, especially at $x/c \approx 0.2$, but the discrepancy with respect to the LES is still much larger.

In addition to the pressure and friction coefficient along the wall, the x-velocity and the off-diagonal Reynolds stress $\langle u'w' \rangle$ are plotted as a profile at a fixed x/c station. Since both C_p and C_f show a large deviation between RANS and LES at $x/c = 1.1$, this location is chosen to show these profiles. The x-velocity and $\langle u'w' \rangle$ profiles are plotted in Fig. 68a and Fig. 68b respectively, where the NASA RANS and LES results are added for comparison. For the x-velocity plot, all meshes have similar profiles, except for the 103 points mesh, which shows a deviation at $0.75 < x/c < 1.1$. The other meshes are rather close together; the discrepancy with respect to the LES is much larger than the discretization error from the 205 points mesh onwards. Next, for the $\langle u'w' \rangle$ profiles, the difference between meshes is much more pronounced. Both the 103 and 205 points mesh show a significant discrepancy with respect to the profiles of the finer meshes. Though the finer meshes also have a discrepancy between their profiles, this discrepancy is much smaller than the discrepancy with respect to the LES.

All in all, the results converge when the mesh is refined, though even the finest mesh still has some discretization error (especially for the $\langle u'w' \rangle$ profile). The choice of mesh depends on how much discretization error is acceptable. As explained, these RANS domain meshes cannot be used for training correction fields, only for testing models. Thus, many runs are to be performed with the chosen mesh, meaning computational cost should be reduced as much as possible. As the discrepancy between the LES and RANS results is much larger than the RANS discretization error, even a relatively large discretization error is acceptable. Hence, the 409 points grid is chosen as it is computationally cheap, while it has an acceptable discretization error. Finally, the NASA RANS data is included in each figure for verification. Though small, there are notable differences between the current run and the NASA results. These are speculated to originate from different turbulence inflow conditions and different wall conditions for the wall profiles. Since the current results are still close to the NASA results, they are considered verified.

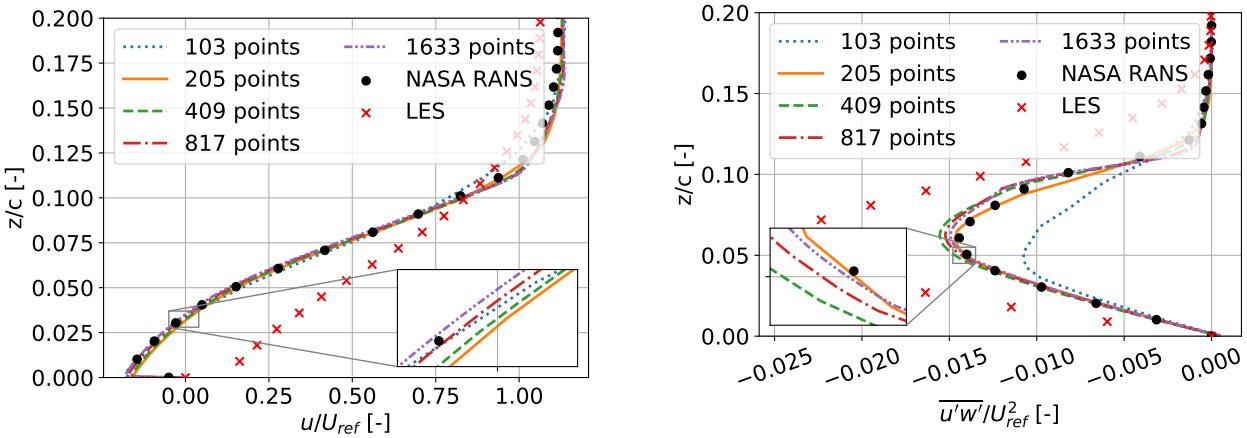
Finally consider the discrepancy between RANS and LES, starting with C_f along the wall in Fig. 67b. LES and RANS are quite close at the inlet, which is to be expected as this is close to flat-plate flow. Then, at the left side of the hump, the LES has a much lower C_f than RANS, which is thought to be due to relaminarization of the flow which is not modelled in standard k-ω SST. This relaminarization 'plateau' is also mentioned in the source paper of the LES [56]. Then, behind the hump ($x > c$), the LES C_f becomes positive much earlier than RANS, indicating earlier flow reattachment. This is also visible in an earlier increase in C_p in Fig. 67a and the fully positive U_x profile at $x = 1.1c$ in Fig. 68a. The reason for the overprediction of the recirculation zone by k-ω SST becomes clear in Fig. 68b; it significantly underpredicts the production of turbulence at the point of separation, which delays reattachment. This is a well known modeling inaccuracy of RANS models [22].



(a) C_p vs x/c

(b) C_f vs x/c

Figure 67: Pressure and friction coefficient along the hump wall for RANS domain meshes at increasing refinement, plotted together with NASA RANS results and LES results for comparison.



(a) z/c vs x-velocity$/U_{ref}$

(b) z/c vs $\langle u'w' \rangle / U_{ref}^2$

Figure 68: Profiles at $x/c = 1.1$ for RANS domain meshes at increasing refinement, plotted together with NASA RANS and LES results for comparison.

10.4 Hump LES domain setup and convergence study

OpenFOAM cases are set up for the hump LES domain meshes of increasing refinement, described in Sec. 10.2. The boundary and initial conditions are given in Sec. 10.4.1 as well as the solution to a discontinuity in the LES data. Next, in Sec. 10.4.2, various ways of approximating the inlet ω from the high fidelity data are studied. Furthermore, the interpolation method from fine LES mesh points to coarse inlet/outlet faces is laid out. Then, in Sec. 10.4.3, the solver settings are given and they are verified to lead to a converged solution. Finally, a mesh independence study is performed in Sec. 10.3.3 and results are compared to the RANS domain and NASA's RANS for verification and to LES for validation (identification of model inadequacies).

10.4.1 Initial and boundary conditions and discontinuity

For the LES domain runs, the `simpleFoam` solver with the k-ω SST turbulence model and the consistent flag set to true is used, just like for the RANS domain. The boundary and initial conditions are given in Tab. 25, where the inlet and outlet values are taken directly from the LES data. This is done to ensure any discrepancies between RANS and LES originate from the RANS equations being solved in the domain, rather than a difference in boundary conditions. A fixed velocity inflow combined with a fixed static pressure outflow is chosen since this is a stable combination of inlet/outlet boundary conditions.

The nondimensionalized LES data is available on each mesh node in the form of the coordinates, velocity, velocity gradient, pressure, Reynolds stress tensor and density. For dimensionalizing the coordinates, velocity, velocity gradient and Reynolds stress tensor, only the chord c and reference velocity U_{ref} are needed (0.42 m and 34.6 m s^{-1} respectively, see Sec. 10.3.1). Dimensionalizing the pressure is somewhat more involved, as a reference pressure of zero is again desired. The LES pressure is given as p/p_{ref}, whereas the input to the incompressible solver should be $(p-p_{ref})/\rho$. This required input is rewritten in terms of p/p_{ref} as follows, where Eq. 88 is used in the last step:

$$\frac{p-p_{ref}}{\rho} = \frac{p_{ref}}{\rho}\left(\frac{p}{p_{ref}}-1\right) = RT_{ref}\left(\frac{p}{p_{ref}}-1\right). \tag{95}$$

Table 25: Boundary- and initial conditions for the hump case on the LES domain.

Mesh part	U [m s^{-1}]	p [m^2 s^{-2}]	k [m^2 s^{-2}]	ω [s^{-1}]	ν_t [m^2 s^{-1}]
Inflow	fixedValue $U_{LES,inlet}$	zeroGradient	fixedValue $k_{LES,inlet}$	fixedValue $\omega_{LES,inlet}$	calculated
Outflow	zeroGradient	fixedValue $p_{LES,outlet}$	zeroGradient		
WallBottom	noSlip	zeroGradient	kqRWall-Function	omegaWall-Function	nutUSpalding-WallFunction
Left	Empty				
Right					
SymmetryTop	Symmetry				
Initial condition	[U_{ref} 0 0]	0	k_{ref}	ω_{ref}	$\nu_{t,ref}$

Now, the dimensional U, p and k can be extracted from the LES data (k is simply half the trace of the Reynolds stress tensor). However, a closer inspection of the LES data reveals a discontinuity which is shown for the $\langle u'u' \rangle$ profile at $x = 0.89c$ in Fig. 69. This discontinuity is present for all Reynolds stress components and velocity derivatives in the LES data, throughout the domain (including the inflow and outflow). The location of the discontinuity is always at the interface between the fine near-wall region and the coarse far-wall region. This also explains the origin of the discontinuity: The LES resolves a larger portion of the Reynolds stress on the fine grid, leading to slightly higher Reynolds stress components in the near-wall region. Why this results in a discontinuity in velocity derivatives is left for further research.

One way to address the discontinuity would be to add the approximate subgrid-scale Reynolds stress components, however, these are not directly available. The discontinuity shown in Fig. 69 has a rather small relative magnitude; for other variables and profile locations, the relative magnitude is even smaller. However, in the k transport equation, first and second derivatives of k appear. Clearly, these would attain extreme values near the discontinuity, ruining the solution. Thus, only the discontinuity in first and second derivatives has to be addressed. This is done by smoothing the solution around the interface, the smoothing procedure is laid out next.

As can be seen in Fig. 63, the mesh consists of columns of points (nodes), all sharing the same x-coordinate. The smoothing process is applied to one column at a time; a 1D cubic spline is constructed of each discontinuous variable against z. This spline is based on all points in the column except those within 20 points of the mesh interface (near the discontinuity). Then, the spline is evaluated at these points within 20 points of the mesh interface and their value is updated to that of the spline. The smoothed result is shown for the $\langle u'u' \rangle$ profile in Fig. 69. Now, the discontinuity in first and second derivatives is gone and the magnitude of $\langle u'u' \rangle$ is still close to the raw data.

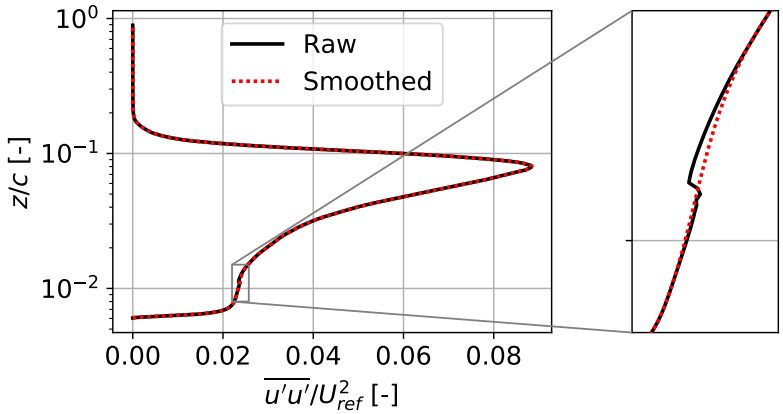


Figure 69: Profile of $\langle u'u' \rangle$ at $x = 0.89c$, with both the raw LES data and the data smoothed at the mesh interface.

10.4.2 Approximation of inlet ω and interpolation

The last inflow variable required, ω, is not present in the LES data and instead has to be approximated in some way. To this end, the Boussinesq approximation given in Eq. 14 is used. Both the Reynolds stress tensor components and the velocity derivatives are available in the LES data, allowing the calculation of ν_t. Due to the tensorial nature of Eq. 14, there are actually multiple equations available to calculate ν_t. Though a weighted least squares of each equation would be most accurate, this is beyond the scope of the current work; instead, the most accurate of these equations is used. The largest velocity derivative at the inlet is $\partial u/\partial z$, so the most accurate equation to calculate ν_t is that of $\langle u'w' \rangle$, given in Eq. 94. The relation between ν_t and ω used in the k-ω SST turbulence model is given in Eq. 30 and further discussed in Sec. 2.5.4. The F_2 function makes the limiter extremely hard to invert, so to calculate ω at the inflow, it is assumed that $a_1\omega$ is always maximum in Eq. 30. Thus, ω at the inlet is calculated as $\omega_{LES,inlet} = k_{LES,inlet}/\nu_t$, where ν_t is found from Eq. 94. The resulting inflow profile is shown as $\omega_{\nu_t,raw}$ in Fig. 70.

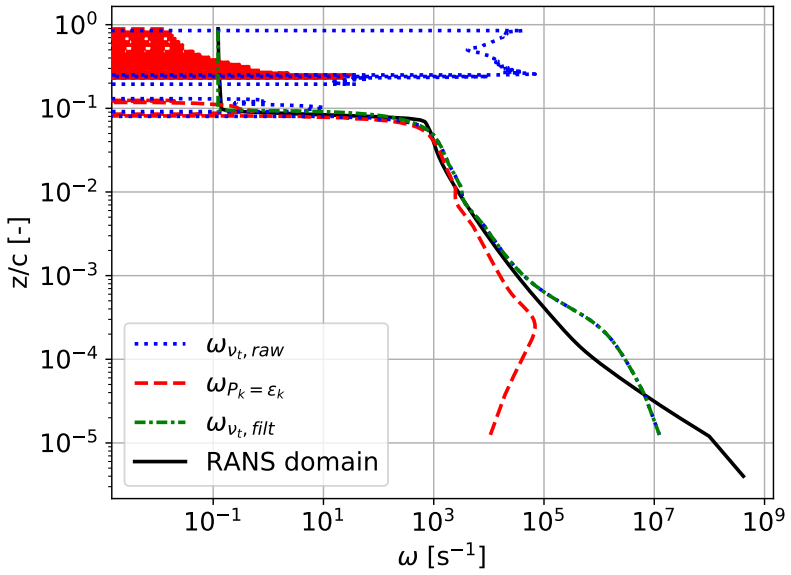


Figure 70: Profiles of ω at the inlet; based on ν_t (raw and filtered), based on $P_k = \epsilon_k$ and extracted from the RANS domain.

Consider the $\omega_{\nu_t,raw}$ profile in Fig. 70; for low z, the profile is smooth, but for high z it is erratic and even goes to negative ω. This erratic behaviour starts around $z = 0.08c$ at the sharp drop of ω, which indicates the edge of the boundary layer. This also explains the erratic behaviour; outside the boundary layer, velocity derivatives and Reynolds stress components are small, so they are prone to fluctuations. Since they are divided to calculate ν_t (see Eq. 94), ν_t has significant fluctuations outside the boundary layer, which further propagates to ω.

To address the erratic ν_t outside the boundary layer, ν_t as calculated from Eq. 94 is no longer used to find ω outside the boundary layer, only inside. The boundary layer edge is defined as the point at which u first reaches 99.9% of U_{ref}. For points outside the boundary layer, consider the RANS domain ω profile at the same location, also plotted in Fig. 70. After the sharp drop at the boundary layer edge, ω goes to a constant value outside the boundary layer. A constant value of ω outside the boundary layer is also used for the LES domain inlet. This constant value is extracted as the minimum value found in the RANS domain at the LES domain inlet location ($x = -0.851c$),

it is equal to $0.123\,\mathrm{s}^{-1}$. To prevent large first and second derivatives resulting from the sudden jump to a constant value, the spline smoothing procedure described earlier in the section is applied again. The resulting filtered ω profile is shown as $\omega_{\nu_t,filt}$ in Fig. 70.

Finally, another way of approximating ω from the LES data is also investigated, namely setting the production of k (P_k) equal to its dissipation (ϵ_k). This results in the following expression:

$$-\left\langle u_i' u_j' \right\rangle \frac{\partial u_i}{\partial x_j} = \beta^* k \omega, \tag{96}$$

where β^* is a model constant equal to 0.09. The resulting ω profile is shown as $\omega_{P_k=\epsilon_k}$ in Fig. 70. Again, ω is erratic outside the boundary layer due to the Reynolds stress and velocity derivatives being prone to fluctuations here. For the outer part of the boundary layer (up till $z \approx 3 \times 10^{-4}$), $\omega_{P_k=\epsilon_k}$ is close to ω_{ν_t}. However, for the inner part, $\omega_{P_k=\epsilon_k}$ unexpectedly decreases again. Likely, the assumption of $P_k = \epsilon_k$ is no longer valid in the inner layer. Since ω_{ν_t} shows a better match with the RANS domain profile, it will be used as the inflow ω profile.

The LES data is given on the mesh nodes, however, the inflow variables are needed on the mesh faces. Furthermore, coarsened versions of the LES mesh are constructed with less inlet/outlet faces, as described in Sec. 10.2. The conversion from nodes to faces is rather trivial; the nodes making up a face are averaged to give the face value. The mesh is coarsened by merging every coarsening factor faces. The variables of the resulting face are calculated as an area weighted average of the underlying faces being merged. Note that a smaller number of faces is merged at the top of the inlet/outlet such that the rest of the faces are evenly divisible by the coarsening factor (see Sec. 10.2 for more details).

10.4.3 Solver settings and run convergence

With the discontinuity addressed and the inlet value of ω approximated, all initial and boundary conditions given in Tab. 25 are available. The inner and outer residuals together with the relaxation factors used in the runs are given in Tab. 26. Again lower inner- and outer residuals are used for ω since it varies over many orders of magnitude. For the other variables, a relatively low outer residual is used; this is needed to properly converge the finest mesh. The relaxation factors used give good stability at a reasonable convergence speed. Though not as severe as for the RANS domain baseline runs, stability is still an issue. To address this, a non-orthogonal corrector step is again introduced as well as a velocity magnitude limiter of $70\,\mathrm{m\,s}^{-1}$ and a minimum ω of $0.1\,\mathrm{s}^{-1}$.

Table 26: Residuals and relaxation factors for the baseline hump case on the LES domain.

Parameter	Inner residual	Outer residual	Relaxation factor
U	10^{-8}	5×10^{-7}	0.9
p			1
k			0.7
ω	10^{-15}	10^{-10}	0.7

To study convergence during the runs, the outer residuals are plotted against iteration for the coarsening factor 2 mesh in Fig. 71. The startup is rather unstable and negative values of k and ω are reported in the log file. However, after some iterations, the residuals assume straight lines in the log-plot and there are no more negative k and ω values, eventually all residuals reach their specified tolerance. Next, several probes are placed in the flow and the flow parameters are mon-

itored during the run. The most critical parameter for convergence turns out to be k, its probe convergence is shown in Fig. 72. Again, a rather unstable start-up phase is observed, but then k goes to a constant value at each probe, indicating convergence. Convergence is also verified for the other meshes using the same check.

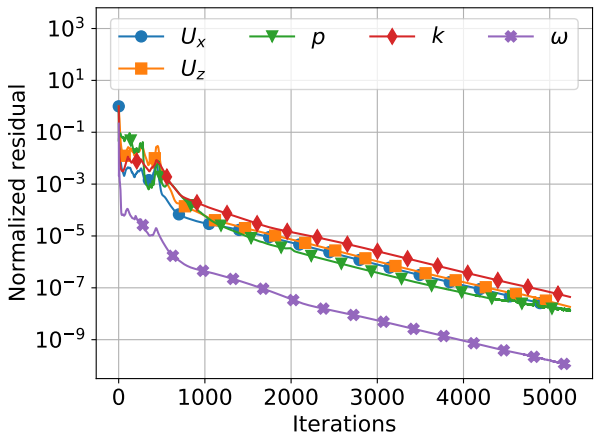


Figure 71: Outer residual versus iteration for various flow parameters, LES domain hump case with 2x coarser mesh.

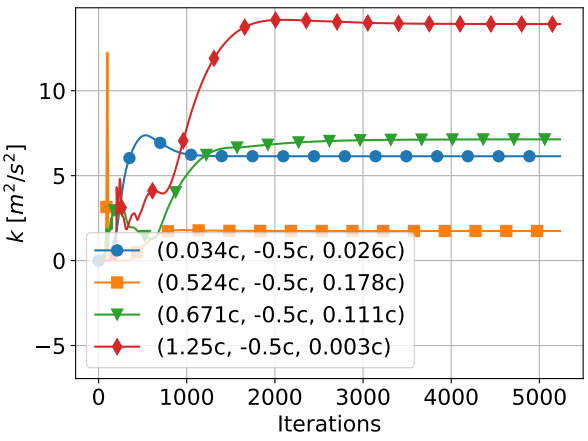


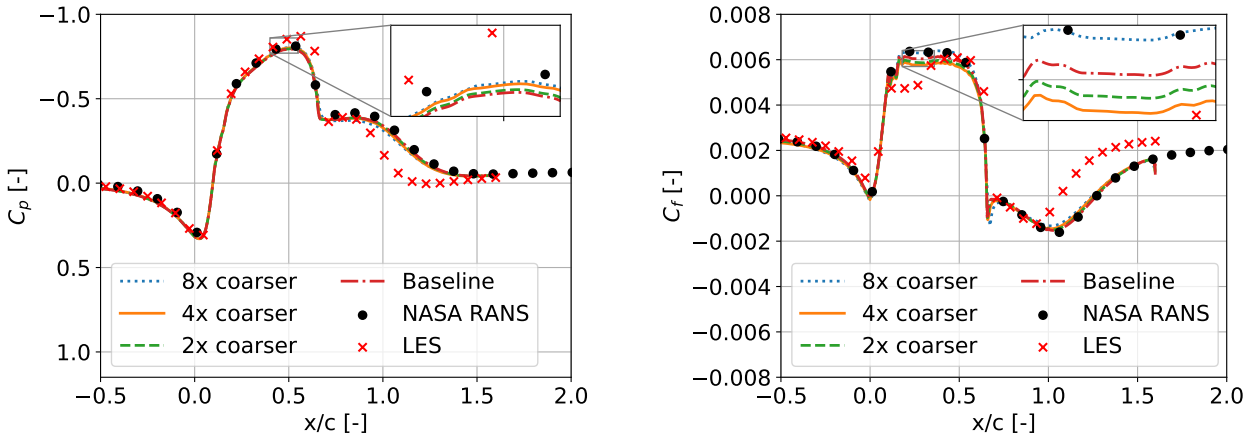
Figure 72: Turbulent kinetic energy versus iteration at various probe locations, LES domain hump case with 2x coarser mesh.

10.4.4 Mesh independence study and validation

As described in Sec. 10.2, four meshes of increasing coarseness are made based on the original LES mesh, as this LES mesh is likely too fine. Now, a mesh independence study is performed to determine which of these meshes to use for further frozen and propagation runs. The pressure coefficient C_p and the friction coefficient C_f are plotted for the meshes in Fig. 73a and Fig. 73b respectively, where NASA's RANS and the LES results are added for comparison. The methodology used to compute C_p and C_f is laid out in Sec. 10.3.2. In the C_p plot, all mesh results are extremely close, but still convergence behaviour is observed with mesh refinement. Though the 8x coarser mesh is on top of the 4x coarser mesh in the zoomed-in portion, they deviate significantly at other locations. In the C_f plot, there is a more notable deviation between meshes, especially around $x = 0.25c$, where there is even a notable deviation between the baseline and 2x coarser mesh. However, for $x > 0.5c$ (the most important region), these two meshes are almost right on top of each-other.

Additionally, profiles at $x = 1.1c$ are plotted again, as this location shows a large deviation between RANS and LES. Profiles of x-velocity are shown in Fig. 74a and profiles of $\langle u'w' \rangle$ are shown in Fig. 74b, again the NASA RANS and LES results are added for comparison. In both figures, convergence is observed with mesh refinement, though relative differences are larger for the $\langle u'w' \rangle$ profile. Whereas the RANS domain mesh is only used for model testing, the current mesh is also used for finding and validating the correction fields with k-corrective-frozen. Discretization errors should be as small as possible in the training process, so the 2x coarser mesh is used, despite the 4x coarser mesh already being close to converged.

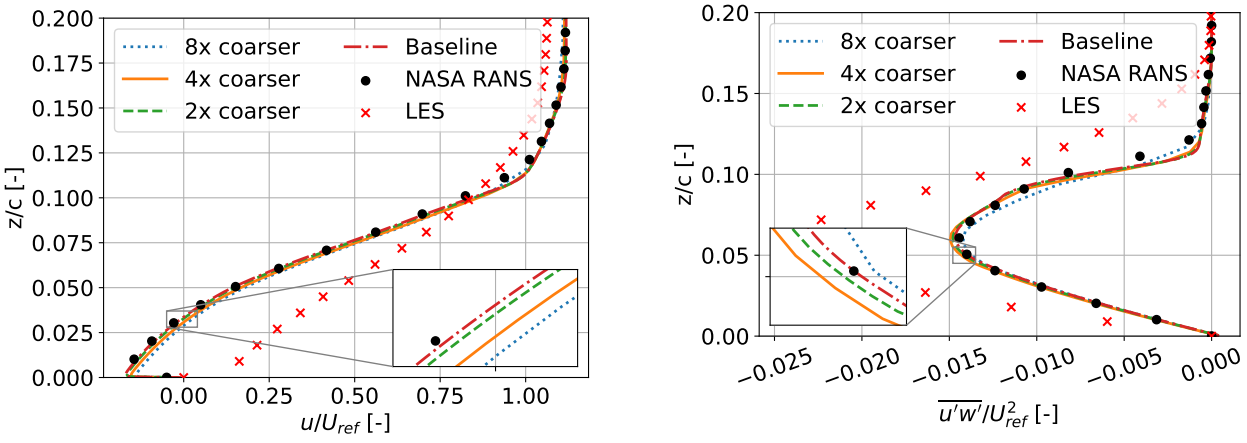
Comparing between the current results and NASA's RANS results in Fig. 73 and Fig. 74, there is a small but notable discrepancy. The magnitude of this discrepancy is similar to that of the discrepancy between the RANS domain results and NASA's RANS results (see Sec. 10.3.3), both of which use the same mesh. In fact, the discrepancy due to the different solver used by NASA seems larger than the discrepancy between the current RANS and LES domain runs. This means that models discovered with the LES domain correction fields should translate well to the larger RANS domain. Also, this provides validation for the discrepancy smoothing and inlet ω approximation used for the LES domain.



(a) C_p vs x/c

(b) C_f vs x/c

Figure 73: Pressure and friction coefficient along the hump wall for LES domain meshes at increasing refinement, plotted together with NASA RANS and LES results for comparison.



(a) z/c vs x-velocity$/U_{ref}$

(b) z/c vs $\langle u'w' \rangle / U^2_{ref}$

Figure 74: Profiles at $x/c = 1.1$ for LES domain meshes at increasing refinement, plotted together with NASA RANS and LES results for comparison.

10.5 Finding correction fields on the LES domain

As laid out in the previous section, there is a significant discrepancy between LES and RANS results of the hump case. To correct the RANS, the fields b_{ij}^{Δ} and R are found for each cell using the k-corrective-frozen approach further explained in Sec. 3.3.3. The custom solver `frozenSimpleFoam` is again used in combination with the custom `frozenkOmegaSST` turbulence model. This solver requires the velocity, Reynolds stress and turbulent kinetic energy of the LES to be available in each cell of the coarsened mesh. Before interpolating the LES data to the coarsened mesh, however, the spline smoothing procedure laid out in Sec. 10.4.1 is applied to the Reynolds stress components and velocity derivatives over the whole domain. This is again to prevent large first and second derivatives originating from the discontinuity at the interface between the near-wall and far-wall mesh regions.

Next, the smoothed LES data is converted from point data to cell data by taking the average of the eight corner nodes of each cell (actually only four nodes are used as the averaged LES data is constant in y-direction). This gives the LES data on the baseline RANS mesh, however, the frozen run is performed on the 2x coarser mesh as explained in the prior section. The mesh is coarsened by merging rectangles with coarsening factor × coarsening factor cells into a single cell. Flow variables for this new cell are calculated as a volume weighted average of the fine mesh cells being merged. Note that at the inlet and top of the domain, a smaller number of cells is merged such that the rest of the domain is evenly divisible by the coarsening factor, this is further explained in Sec. 10.2.

As explained in Sec. 3.3.3, ω is the only parameter being solved for in the frozen run. The inner residual of ω is again set to 10^{-15}, but the outer residual is increased to 5×10^{-10} which is explained in a bit. A lower relaxation factor of 0.5 is used as the case is notably more unstable than prior cases. Contrary to the baseline run, the consistent flag is set to false, as setting it to true gives floating point errors in the pressure solve at the end of the run (this is left for further research). The boundary conditions used for the baseline run are also applied here, they are given in Tab. 25. For the Reynolds stress tensor field (τ_{ij}), a zero gradient condition is used at the inlet, outlet and top, while a zero Dirichlet condition is used at the wall. For p, ω and ν_t, the initial conditions listed in Tab. 25 are used. For U, k and τ_{ij}, the LES value is used as the initial condition (these also remain constant through the run).

As mentioned, a lower relaxation factor is used since the run is rather unstable, which also leads to negative values of ω at the start of the run. Even at the end of the run, there are still negative ω values, but these remain constant. To assess convergence, the outer residual is plotted against iteration in Fig. 75. After some startup, the solution converges in a straight line in the log-plot, as expected. However, after 5000 iterations, the residual plateaus to a value slightly below 5×10^{-10}. This plateauing is thought to originate either from the bounding of ω or noise in the LES data. This plateauing is also the reason for the outer residual of 5×10^{-10}; a lower one is not possible.

To ensure the solution is converged at this outer residual, the flow is again probed at several locations. The evolution of b_{13}^{Δ} is plotted against iteration at these probe locations in Fig. 76. Clearly, b_{13}^{Δ} has reached a constant value at the end of the run, which is also observed for the other components of b_{ij}^{Δ}, R and ω, confirming convergence. The point of convergence seems to be after 2000 iterations at which point the outer residual is at 3×10^{-7}. This is again much lower than the outer residual one would normally use for a run with k-ω SST.

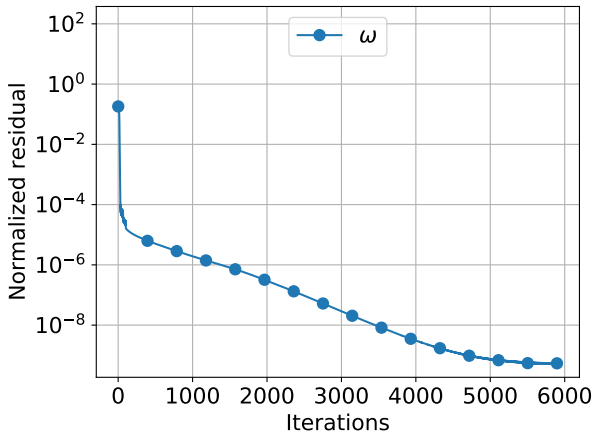


Figure 75: Outer residual of ω versus iteration for the frozen run of the hump on the 2x coarser mesh.

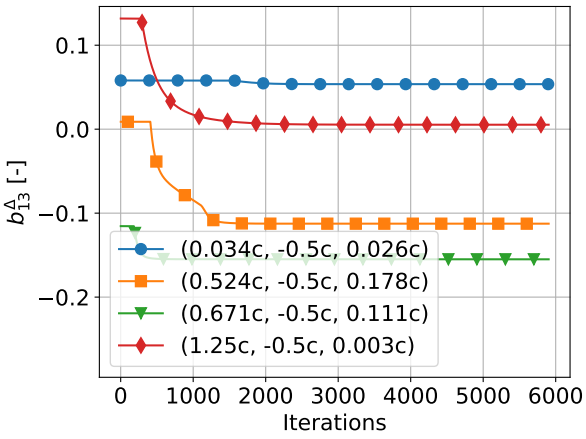


Figure 76: b_{13}^{Δ} versus iteration at various probe locations for the frozen run of the hump on the 2x coarser mesh.

10.6 Validating correction fields on the LES domain

In order to validate the correction fields found in the previous section, they are injected into a full RANS solver where velocity and Reynolds stress are allowed to update (see Sec. 6.4). The `propagationkOmegaSST` turbulence model is used together with the native `simpleFoam` solver, with the consistent flag set to true to improve stability. The boundary conditions used for the baseline run are applied here as well, they are listed in Tab. 25. The baseline solution fields are used as initial conditions; this improves stability and speed of convergence, but also ensures any observed improvements over the baseline come from the correction fields rather than the initial conditions.

The residuals and relaxation factors for the propagation run are provided in Tab. 27. The relaxation factors are significantly lower than for the baseline case due to stability issues. However, the initial condition instabilities encountered for the baseline case, which required a non-orthogonal pressure corrector, are no longer present. This is undoubtedly due to the present initial condition (the baseline solution) serving as a much more physical initial condition than the baseline initial condition (constant fields). As such, no non-orthogonal corrector steps are used for the present propagation run. The inner residuals are adapted from the baseline as well as the outer residuals for U, p and k, however, a higher outer residual is used for ω, which is motivated next.

Table 27: Residuals and relaxation factors for the propagation hump case on the LES domain.

Parameter	Inner residual	Outer residual	Relaxation factor
U	10^{-8}	5×10^{-7}	0.8
p			0.7
k			0.4
ω	10^{-15}	1.5×10^{-8}	0.4

As for the frozen run, negative values of ω are encountered till the end of the run, though they do reach a constant value at the end of the run. Also, all flow parameters converge despite these negative ω values, as shown in Fig. 77, where the outer residuals are plotted against iteration. Up till approximately 4000 iterations, ω shows expected convergence behaviour, but after this point

the convergence starts plateauing which is also observed in the frozen run. It plateaus to an outer residual of approximately 3×10^{-8}, which is why this is used as the outer residual tolerance for ω. To check whether this outer residual is sufficiently low for convergence, probes are placed at several points in the flow and probed flow parameters are plotted against iteration. The most critical parameter for convergence turns out to be k; its probe convergence is shown in Fig. 78. Clearly, k has assumed a constant value at the end of the run, so the solution is considered converged.

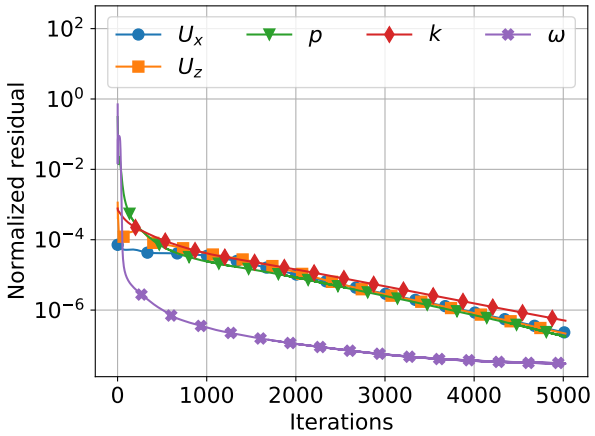


Figure 77: Outer residual versus iteration for various flow parameters, LES domain hump propagation case with 2x coarser mesh.

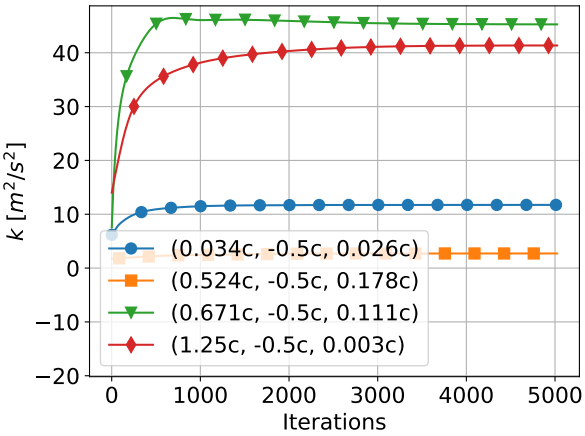
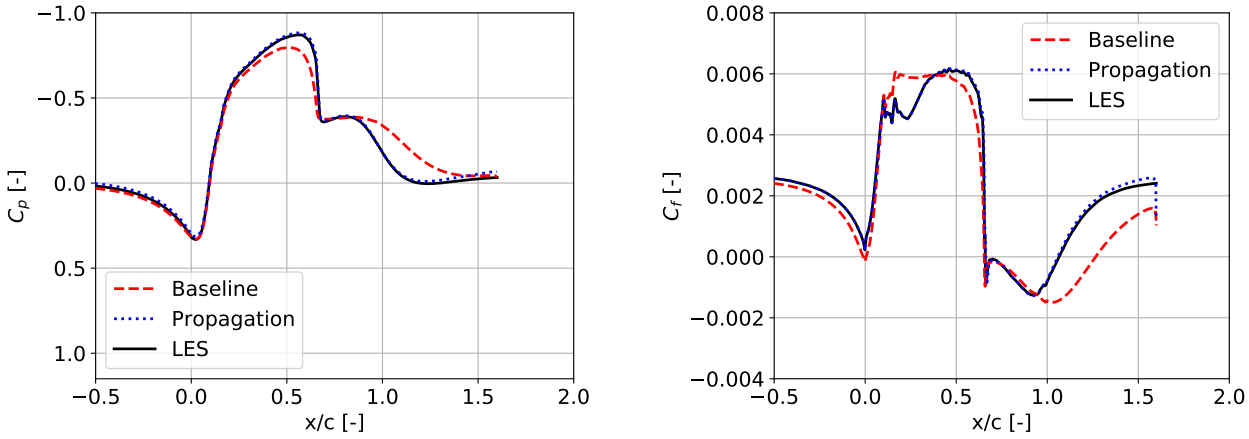


Figure 78: Turbulent kinetic energy versus iteration at various probe locations, LES domain hump propagation case with 2x coarser mesh.

The results of the propagation run are post-processed to obtain derived variables, such as the pressure coefficient C_p, skin friction coefficient C_f and the off-diagonal Reynolds stress $\langle u'w' \rangle$. The methodology to obtain these derived variables is the same as the baseline, it is laid out in Sec. 10.3.2. However, there is one important difference: $2kb_{13}^{\Delta}$ should be added to $\langle u'w' \rangle$ (see Eq. 94), as the Boussinesq eddy viscosity hypothesis is modified in propagation. Validation of the correction fields requires comparing the propagation results with the baseline results and the LES data. These are compared in four plots: C_p and C_f are plotted along the wall in Fig. 79a and Fig. 79b respectively, profiles of U_x and $\langle u'w' \rangle$ are shown at various x/c stations in Fig. 80a and Fig. 80b respectively.

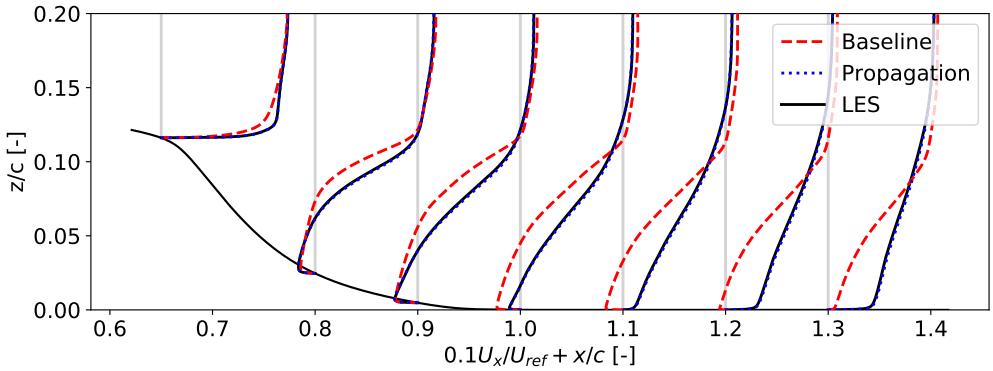
In both the C_p plot and the C_f plot in Fig. 79, the propagation matches the LES extremely well, except for $x > 1.3c$ where there is a small deviation. This deviation is thought to originate from the outlet length being too small for RANS. Nonetheless, the recirculation region behind the hump is captured well by the propagation, including the point of reattachment. The recirculation region is the most important to get right, as it has the largest discrepancy between baseline and LES. The U_x and $\langle u'w' \rangle$ profiles in Fig. 80 are of this recirculation region only. A good match between propagation and LES is also found for these profiles, hence the correction fields are considered validated.



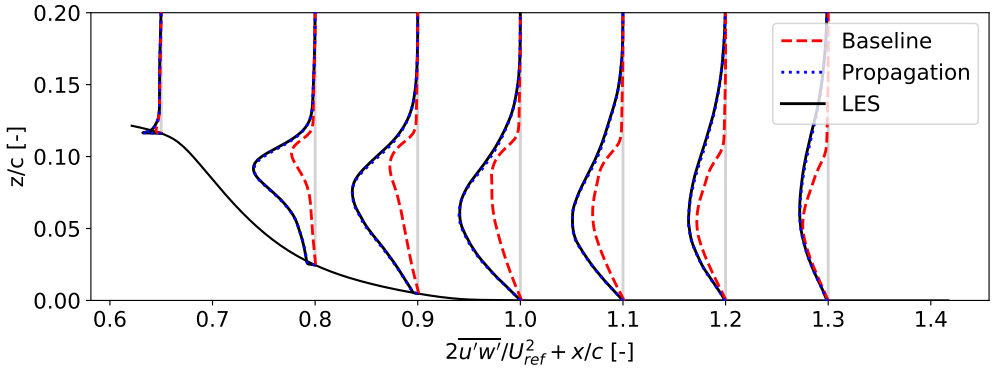
(a) C_p vs x/c

(b) C_f vs x/c

Figure 79: Pressure- and skin friction coefficient along the hump wall, using the 2x coarser LES domain mesh, comparing between baseline (k-ω SST), propagation (k-ω SST with correction terms b_{ij}^{Δ} and R) and LES results.



(a) Profiles of x-velocity



(b) Profiles of $\langle u'w' \rangle$

Figure 80: Profiles of x-velocity and $\langle u'w' \rangle$ behind the hump, using the 2x coarser LES domain mesh, comparing between baseline (k-ω SST), propagation (k-ω SST with correction terms b_{ij}^{Δ} and R) and LES results.

11 Classifier training criterion

The classifier is a boolean function of space, denoted $\sigma(x_i)$. The corrections R and b_{ij}^{Δ} are multiplied by this classifier, such that they are only active if $\sigma = 1$. A proper classifier should only be zero (no corrections applied) in regions where the corrections are not important, such that the propagated solution changes negligibly under the classifier. Following Steiner et al., the idea is to train a symbolic expression for the classifier which is evaluated at each iteration in a RANS solver [54]. Training of this symbolic expression is based on 'exact' fields of $\sigma(x_i)$ for each case, which are obtained using a training criterion. The training criterion used in the NASA challenge (given in Eq. 61) is unsuitable as it is nonlocal, as further explained in Sec. 5.2.1. The criterion used by Steiner et al. in Eq. 43 is local, however, since both R and b_{ij}^{Δ} modify the production of k, it is argued here that they should be considered together, giving the following criterion:

$$\sigma := \begin{cases} 1 \text{ if } & \frac{\left|2kb_{ij}^{\Delta}\left(\partial\langle u_i\rangle/\partial x_j\right)\right|+|R|}{\left|P_{k,LES}\right|+\epsilon} > \xi \\ 0 \text{ otherwise.} & \end{cases} \tag{97}$$

Here, ξ is the classifier threshold and $\epsilon = 10^{-10}$ is used (rather than $\epsilon = 0.01$ used by Steiner et al.). This decrease in ϵ is due it being dimensional; for cases with small k, $\epsilon = 0.01$ may be close to $P_{k,LES}$.

Finding a suitable value for ξ is a tradeoff between accuracy and number of cells being activated. A large ξ would give $\sigma = 0$ in most cells, likely resulting in a significant discrepancy between propagation with and without this classifier. A small ξ, on the other hand, would give $\sigma = 1$ in most cells, likely resulting in no significant discrepancy between propagation with and without this classifier. The goal of this section is to find a value ξ that gives $\sigma = 0$ in as many cells as possible, while retaining a good propagation match with the high-fidelity data. Various values of ξ are tested for the rd1L, hr00 and hump case, resulting contours/profiles are plotted together with baseline and DNS/LES in Fig. 81, Fig. 82 and Fig. 83 respectively.

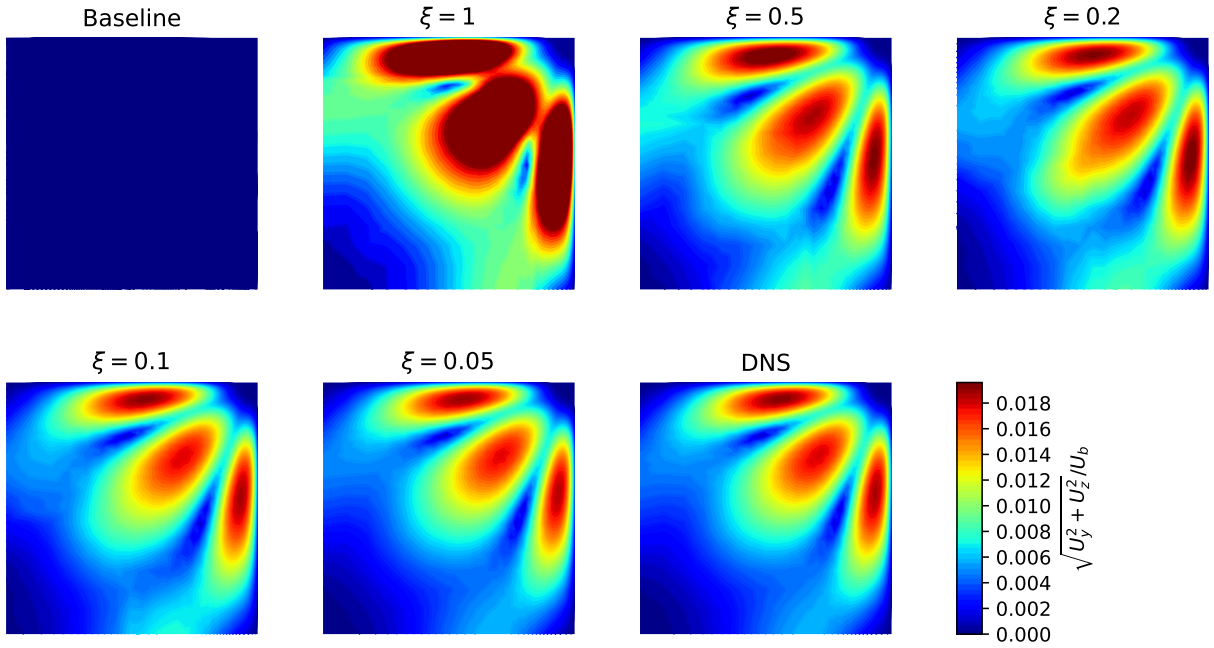


Figure 81: In-plane velocity contours, comparing between baseline (k-ω SST), propagation under various classifier criteria (see Eq. 97) and DNS results, case rd1L.

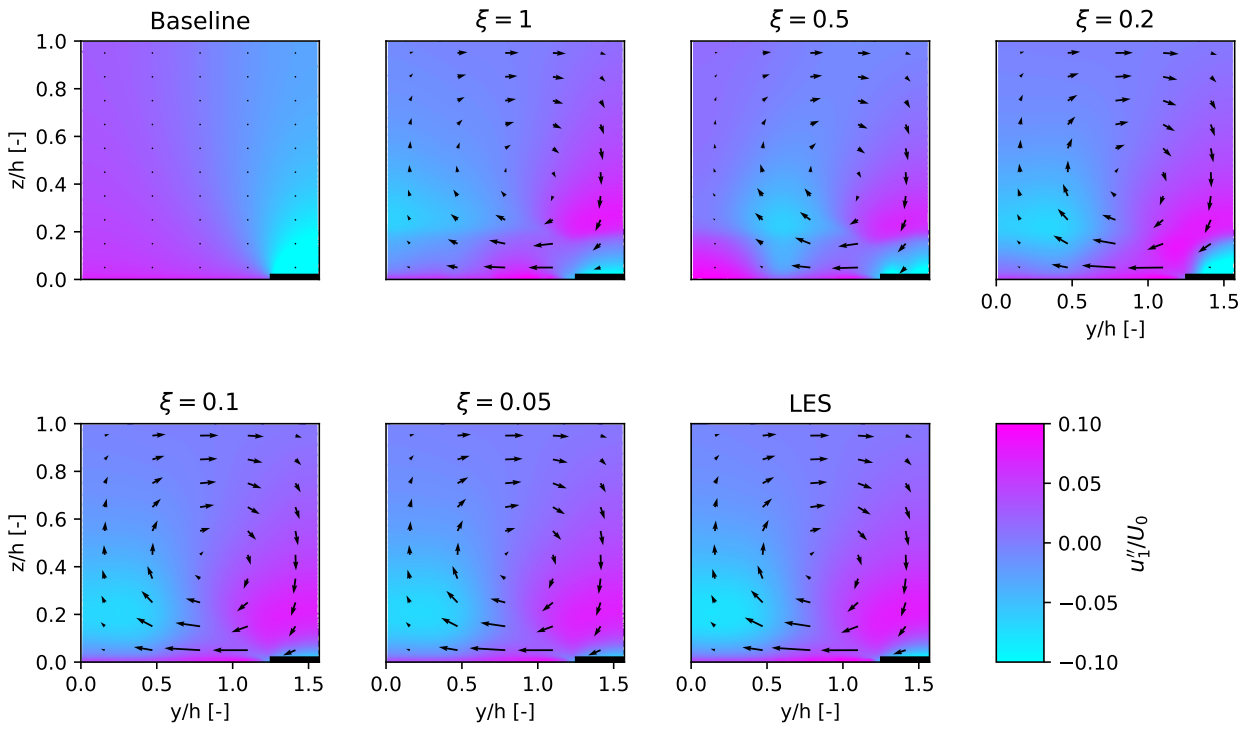


Figure 82: Contours of dispersive x-velocity (see Eq. 74) with the in-plane velocity vector field overlaid, comparing between baseline (k-ω SST), propagation under various classifier criteria (see Eq. 97) and LES results, case hr00.

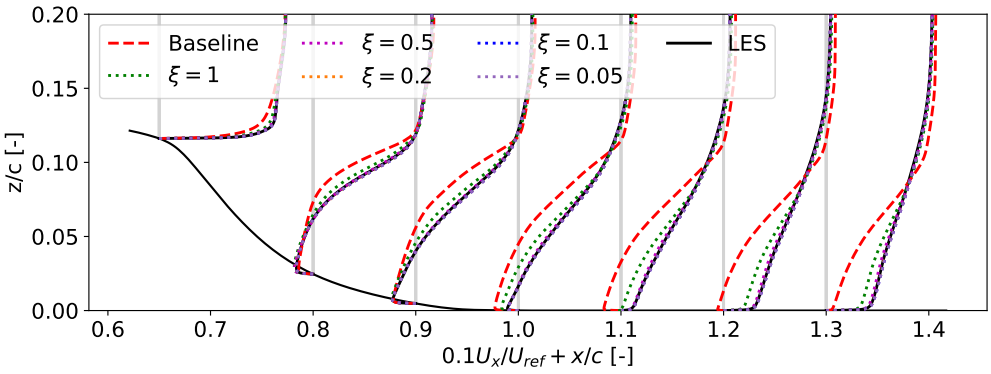


Figure 83: Profiles of x-velocity over the hump, using the 2x coarser LES domain mesh, comparing between baseline (k-ω SST), propagation under various classifier criteria (see Eq. 97) and LES results.

The main effect of the classifier on the rectangular duct's in-plane velocity seems to be on the magnitude rather than the shape, as seen in Fig. 81. The highest ξ that still gives an acceptable flowfield is $\xi = 0.2$, though it is already noticeably different from the DNS. Next, consider the effect of the classifier on the heterogeneous roughness in Fig. 82. The area just above the roughness strip in the lower right corner seems to be most affected. At $\xi = 0.2$, this part is already significantly different from the DNS, but the rest of the domain is still in good agreement. Then at $\xi = 0.5$, also the rest of the domain is significantly different from the DNS. Now, consider the effect of the classifier on the hump in Fig. 83. Even at $\xi = 0.5$, there is an excellent match with the LES data, but at $\xi = 1$ there is a notable discrepancy. In the end, a global classifier criterion is desired to obtain training data. Based on the discussion above, $\xi = 0.2$ is chosen as the highest acceptable value of ξ. The associated classifier activation field is shown for each of the three cases in Fig. 84 below.

(a) Rectangular duct case rd1L

(b) Heterogeneous roughness case hr00

(c) Hump LES domain

(d) Hump zoomed at separation bubble

Figure 84: Activation of the classifier for various cases at $\xi = 0.2$, black indicates $\sigma = 1$; grey indicates $\sigma = 0$.

Consider the classifier activation in Fig. 84, where grey cells indicate $\sigma = 0$ and black cells indicate $\sigma = 1$. For both the duct and the heterogeneous roughness, most of the domain is activated. This is expected as both contain secondary motions throughout their domain, meaning corrections have to be applied almost everywhere to capture these motions. Places of inactivation are also precisely where discrepancies are found with respect to DNS/LES in Fig. 81 (near symmetry) and Fig. 82 (above roughness). Next, the classifier is also active in most of the hump domain as seen in Fig. 84c. This is unexpected, as the inflow of the hump should be similar to a flat plate and the top only acts as a farfield. Corrections are only expected to be relevant over and behind the hump. To further investigate this, the numerator of the fraction in Eq. 97, here called $P_k^{|\Delta|}$, is shown in Fig. 85. Indeed, $P_k^{|\Delta|}$ is only large over- and behind the hump where corrections are expected.

Figure 85: Modification of P_k due to R and b_{ij}^{Δ} $\left(P_k^{|\Delta|} = \left|2kb_{ij}^{\Delta}\left(\partial\langle u_i\rangle/\partial x_j\right)\right| + |R|\right)$ for the hump on the 2x coarser LES domain mesh.

The fact that the classifier activates in regions with small $P_k^{|\Delta|}$ indicates that the denominator in Eq. 97 is also small. This denominator, $\left|P_{k,LES}\right|$, is shown on a logarithmic scale in Fig. 86. Indeed, $\left|P_{k,LES}\right|$ is small at the inlet and top of the domain, leading to the unexpected classifier activation. This figure also reveals why erroneous activation is not observed as much by Steiner et al.; $P_k^{|\Delta|}$ is close to or smaller than 0.01 at most points of erroneous activation. Since Steiner et al. use $\epsilon = 0.01$, this quantity which is only meant to prevent division by zero is actually preventing activation.

Based on Fig. 85, one may propose $P_k^{|\Delta|} > 3000\,\mathrm{m^2\,s^{-3}}$ as a better classifier activation criterion. A similar idea is used in the NASA challenge, where the denominator is replaced with the domain averaged $P_{k,LES}$ (see Sec. 5.2.1). While these approaches may provide an *ad-hoc* fix, such criteria go against the spirit of RANS modeling, where global variables should be nondimensional. For example, if a second, much smaller hump is added before the existing one, it would need a different $P_k^{|\Delta|}$ threshold. Similarly, ϵ is a dimensional quantity and should thus not interfere with the classifier activation. Due to the fundamental nature of the current study, such *ad-hoc* fixes are not used to generate classifier training fields. No satisfactory classifier criterion based only on local quantities is known to the author. Following this difficulty in attaining training data combined with the limited success of the classifier found by Steiner et al. and in the NASA challenge, it is decided not to further pursue it.

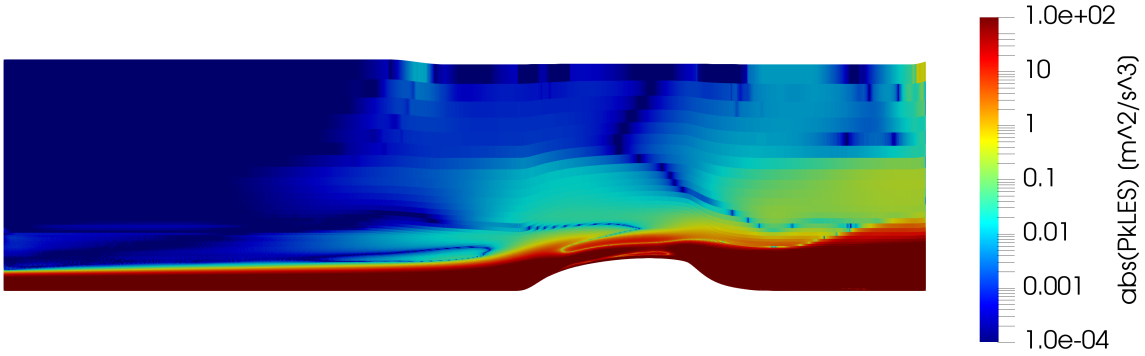


Figure 86: Absolute P_k of the LES results for the hump on the 2x coarser LES domain mesh.

12 Model training and testing results

So far, baseline cases for five flows have been established (duct, roughness, channel, flat plate and hump) and frozen correction fields have been found and validated for the flows requiring corrections (duct, roughness and hump). The next step is to train and validate models for the R and b_{ij}^{Δ} corrections of these cases. Conventionally, model training is performed on the frozen case, however, it is argued here that the propagation case should be used. Propagation provides the upper performance limit of a model, so *a-posteriori* flowfields of a good model are likely closer to propagation than frozen. The section is structured as follows: in Sec. 12.1, models are trained for the R correction, they are tested in isolation by using the exact b_{ij}^{Δ} (the frozen field). Next, b_{ij}^{Δ} models are trained in Sec. 12.2, they are also trained in isolation, now by using the exact R (the frozen field). Finally, in Sec. 12.3, the best isolated R and b_{ij}^{Δ} model are combined into a full model which is tested on each case.

12.1 Isolated R model training and testing

Models for R have the same form as used by Steiner et al., who use (functions of) dimensionless, scalar features multiplied by eleven bases, see Sec. 3.3.4. Ten bases are formed by the P_k modification of each pope base tensor $T_{ij}^{(\lambda)}$. In the SpaRTA framework, these bases are named G_λ (not to be confused with G^λ in Eq. 42), their definition is as follows:

$$G_\lambda = 2kT_{ij}^{(\lambda)}\frac{\partial\langle u_i\rangle}{\partial x_j}. \tag{98}$$

The eleventh basis is simply the dissipation rate ϵ. To avoid confusion, the scalar function by which each basis is multiplied is referred to as $f_\lambda(q)$ in the current work, where q is a scalar feature/invariant (see Sec. 3.3.4). The objective of the current section is to train expressions for $f_\lambda(q)$ and test these *a-posteriori*, striving for simple and general R models. To isolate the effect of the R model, the frozen b_{ij}^{Δ} field is used in testing.

In Sec. 12.1.1, the same model form as used in the challenge (constant times a single basis) is trained and tested. To increase model accuracy, a second basis is added in Sec. 12.1.2, both bases are multiplied by their own constant. These two constants are again trained and tested, but model accuracy is still insufficient. Hence, in Sec. 12.1.3 the SpaRTA framework is used to train a model that also includes features, still only with two bases. While this model has good accuracy, it is Reynolds number dependent, making it non-general. Thus, in Sec. 12.1.4 the simple model form in Sec. 12.1.1 is manually *a-posteriori* optimized to attain an accurate, general R model.

12.1.1 Single basis model

Given the success of the simple R model found for the NASA challenge ($R = 0.079\epsilon$), initial efforts focus on training the same model form of a constant multiplied by a basis. This is further motivated by the fact that the best R models found by Schmelzer et al. also take this form, despite more complex terms being in the library [46]. Next, no weighing was used to regress the challenge R model, but it is argued here that volume weighing should be used. For example, if a mesh independent mesh is refined in a random area, corrections change negligibly. However, without volume weighing, the increased number of cells would result in a higher total weight of

this random area in the regression. Using volume weighted least squares, each basis is regressed for each case with nonzero corrections, ϵ by far provides the best fit for each case. The resulting model form is $R = C_0\epsilon$, the regressed value of C_0 is given in Tab. 28 for each case along with the coefficient of determination (R^2).

Table 28: Fitted C_0 in $R = C_0\epsilon$ and R^2 of the fit.

Rectangular duct			**Heterogeneous roughness**		
Case	C_0	R^2	**Case**	C_0	R^2
rd1L	0.0774	0.981	hr00	0.0317	0.537
rd1H	0.0781	0.984	hr03	0.0309	0.648
rd3L	0.0777	0.984	hr04	0.0318	0.491
rd3H	0.0782	0.986	hr06	0.0360	0.787
rd5L	0.0777	0.985	hr07	0.0411	0.913
rd7L	0.0777	0.985	hr08	0.0390	0.658
rd10L	0.0777	0.985	hr09	0.0316	0.559
rd14L	0.0777	0.985	hr10	0.0311	0.612
Hump			hr13	0.0348	0.746
Case	C_0	R^2	hr15	0.0404	0.920
hump	0.0757	0.948	hr16	0.0420	0.930

In Tab. 28, the C_0 of the hump is $\sim 4\%$ lower than for the challenge model, stemming from the volume weighing and fitting to the whole domain rather than only cells with classifier activation. This also results in a slightly lower R^2, though it is still a good fit. Next, C_0 is similar among duct cases and each case has a high R^2. This generalization over different aspect ratios and Reynolds numbers is evidence that the model has some universal nature (at least *a-priori*). In fact, the model even seems to generalize to the hump, which has a C_0 only 2.6% smaller than the duct average. Now consider the roughness cases, which have a much greater variation in C_0 and a lower R^2. Some do have a high R^2, but these are exactly the cases with small secondary motions. Compared to the hump and duct, the roughness cases have a much lower C_0. This discrepancy does not immediately disprove the universal nature of the current model form. The use of wall models and the non-smooth correction fields (see Sec. 7.3) introduce substantial uncertainty to the roughness, also indicated by the low R^2. Still, more cases should be added to confirm the universality of the model form.

Given the uncertainty in the roughness, a general model is constructed from only the hump and duct C_0s. Equally weighing each duct case gives $C_0 = 0.0778$; weighing this equal to the single hump case gives $C_0 = 0.0767$, resulting in the R model $M_R^{(1)}$:

$$M_R^{(1)} = 0.0767\epsilon. \tag{99}$$

Each duct, hump and roughness case is tested with this R model and the exact frozen b_{ij}^{Δ} field. The same initial conditions, boundary conditions, residuals and relaxation factors as propagation are used. Contours of k and in-plane velocity are shown for the rd1L case in Fig. 87, together with baseline and DNS. Furthermore, profiles of k and U_x are shown for the hump case in Fig. 88, together with baseline and LES. No results are shown for the roughness as not a single case

converges under the model. This is not surprising, as the model's C_0 is significantly higher than the regressed C_0 for any roughness case, giving excessive P_k.

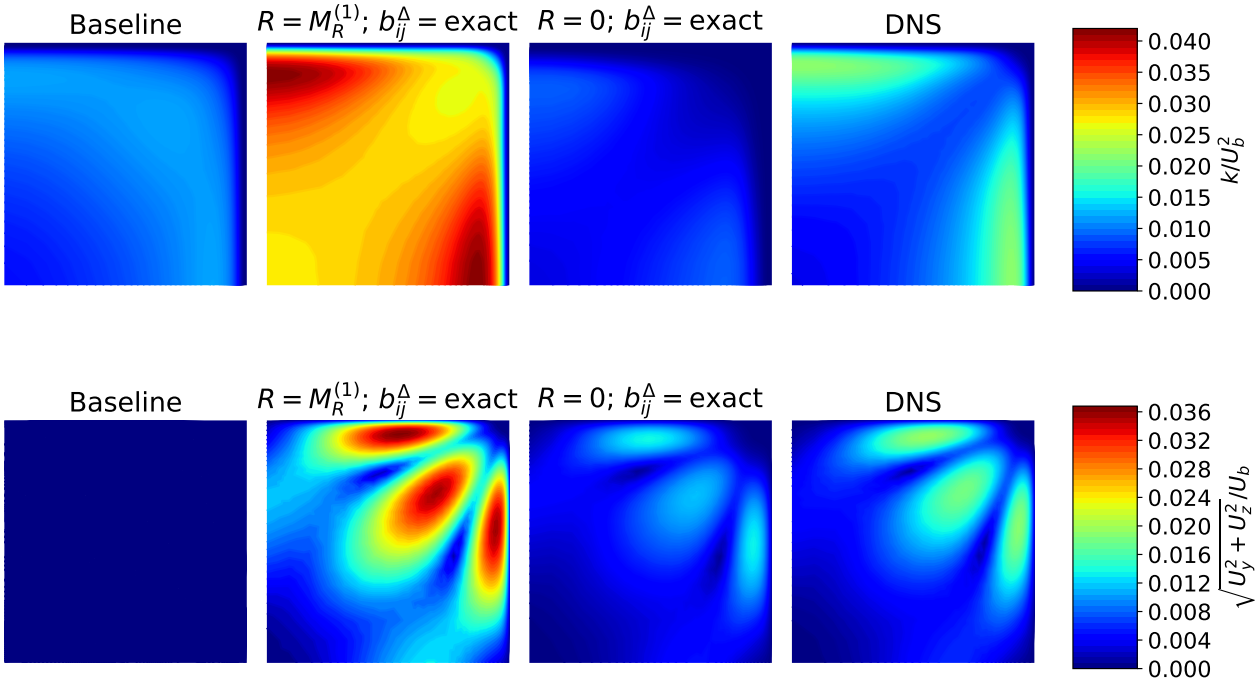


Figure 87: Contours of k and in-plane velocity for the rd1L case for $R = M_R^{(1)}$ (Eq. 99) and $R = 0$ with exact b_{ij}^{Δ}, with baseline (k-ω SST) and DNS added for comparison.

First consider the k profiles in Fig. 87; the model severely overpredicts k, despite its good fit with the training R. This is not necessarily a problem as the velocity is usually of practical interest rather than k. The NASA challenge model also significantly overpredicts k, but still gives a good velocity prediction (see Fig. 12). However, in this case the velocity is adversely affected by the overprediction of k, looking at the in-plane velocity contours. The shape of the in-plane motions is predicted correctly, but the magnitude is severely overpredicted. This overprediction of k and in-plane velocity magnitude is found for all duct cases.

As the duct is dominated by anisotropy, the R correction may not be necessary at all (as was the case for the b_{ij}^{Δ} correction in the challenge). Hence, the case is run again with $R = 0$ and the exact b_{ij}^{Δ} field, the k and in-plane velocity contours are also shown in Fig. 87. Now k is severely underpredicted, resulting in an underprediction of in-plane velocity magnitude. In fact, k is even lower than baseline, meaning b_{ij}^{Δ} has a negative effect on P_k. Hence, a nonzero R model is definitely required to get the proper in-plane velocity magnitude. A C_0 may exist between 0 and 0.0767 that gives the correct in-plane velocity magnitude, but it cannot be found with the SpaRTA framework.

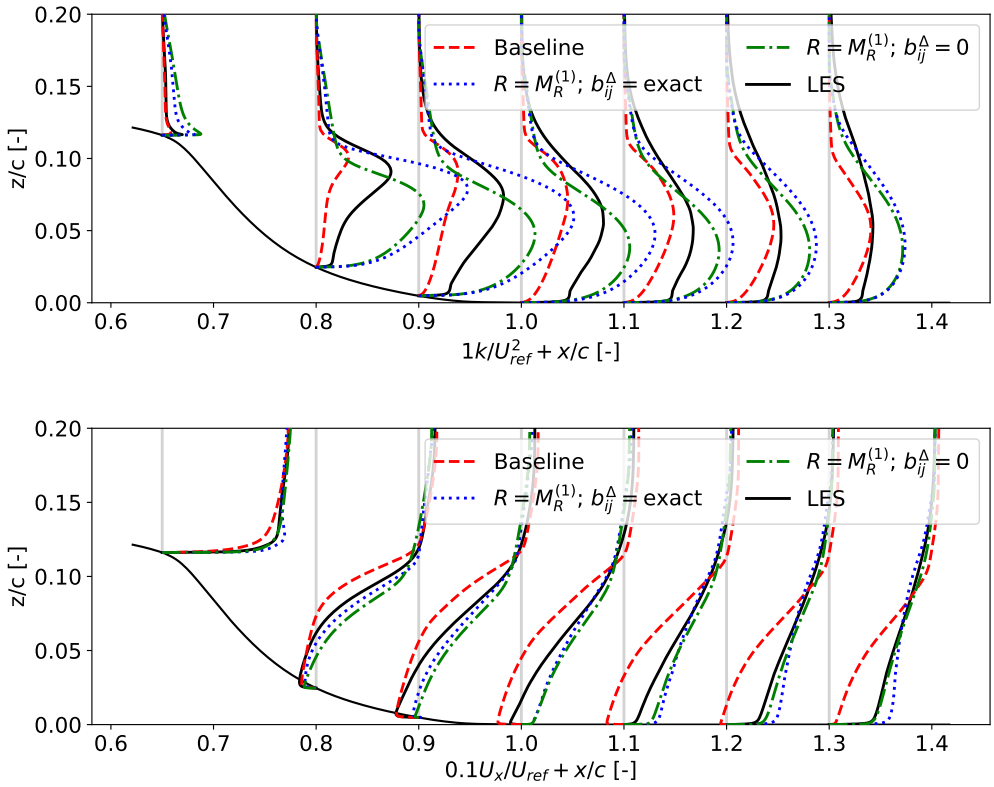


Figure 88: Profiles of k and x-velocity for the hump case for $R = M_R^{(1)}$ (Eq. 99) with exact and zero b_{ij}^{Δ}, with baseline (k-ω SST) and LES added for comparison.

Consider the k profiles of the hump in Fig. 88; the model again overpredicts k, especially behind the point of separation. As for the duct, the correction is in the right direction, but has a too large magnitude. This also results in the overprediction of x-velocity, though it is closer to the LES than baseline. Strangely, the match with LES is worse than for the challenge model, even though the exact b_{ij}^{Δ} is now used. For a better comparison, the current model is run with $b_{ij}^{\Delta} = 0$ (used by the challenge model), the results are also shown in Fig. 88. Surprisingly, $b_{ij}^{\Delta} = 0$ gives better predictions of k and U_x than using the exact b_{ij}^{Δ}. Still, the $M_R^{(1)}$ model performs worse than the challenge model, likely due to the lack of a classifier. Finally, as for the duct, it is speculated that a model exists with C_0 between 0 and 0.0767 that performs better than either model shown here.

12.1.2 Two bases model

Given the *a-posteriori* misfit of the single basis R model laid out in the prior section, more accurate models are trained. The first attempt at increasing accuracy is the addition of a second basis, still only multiplied by a coefficient. Now volume weighted least squares is used to regress each possible pair of bases for each case with nonzero corrections. Every time, ϵ and G_1 provide the best fit, giving the model form $R = C_0\epsilon + C_1 G_1$. The fitted C_0 and C_1 are given for each case in Tab. 29 along with the R^2 of the fit.

Table 29: Fitted C_0 and C_1 in $R = C_0\epsilon + C_1 G_1$ and R^2 of the fit.

Rectangular duct				**Heterogeneous roughness**			
Case	C_0	C_1	R^2	**Case**	C_0	C_1	R^2
rd1L	0.0835	-0.0508	0.988	hr00	0.120	-0.328	0.788
rd1H	0.0841	-0.0584	0.990	hr03	0.122	-0.328	0.811
rd3L	0.0837	-0.0536	0.990	hr04	0.119	-0.320	0.755
rd3H	0.0841	-0.0579	0.992	hr06	0.132	-0.361	0.914
rd5L	0.0836	-0.0527	0.990	hr07	0.132	-0.353	0.956
rd7L	0.0836	-0.0525	0.990	hr08	0.116	-0.284	0.856
rd10L	0.0836	-0.0521	0.990	hr09	0.120	-0.323	0.798
rd14L	0.0836	-0.0526	0.990	hr10	0.119	-0.310	0.865
Hump				hr13	0.130	-0.352	0.908
Case	C_0	C_1	R^2	hr15	0.125	-0.324	0.954
hump	0.0832	-0.0569	0.954	hr16	0.123	-0.314	0.951

In Tab. 29, there is again excellent agreement between duct cases for C_0 and good agreement for C_1. The R^2 is significantly higher than for the single basis R model (see Tab. 28). For the hump on the other hand, R^2 is only slightly better than its single basis R model. Surprisingly, the hump C_0 and C_1 are again close to the averaged C_0 and C_1 of the duct (0.6% and 6% difference respectively). This is evidence that the model form may have a universal nature. Next, the roughness cases again have a greater variation in coefficients and lower R^2 values (especially for cases with significant secondary motions). Following the aforementioned uncertainty in the training data of the roughness, the generalized model is only based on the duct and hump.

The duct and hump are weighed equally to arrive at the generalized R model $M_R^{(2)}$:

$$M_R^{(2)} = 0.0835\epsilon - 0.0559 G_1, \tag{100}$$

it is tested for each case with the exact frozen b_{ij}^{Δ} field. Contours of k and in-plane velocity are shown for the rd1L case in Fig. 89, together with baseline, DNS and $M_R^{(1)}$ (Eq. 99). Profiles of k and U_x are shown for the hump case in Fig. 90, together with baseline, LES and $M_R^{(1)}$ with zero b_{ij}^{Δ}. No roughness results are shown as not a single case converges with this model, likely due to the significant difference in optimal coefficients.

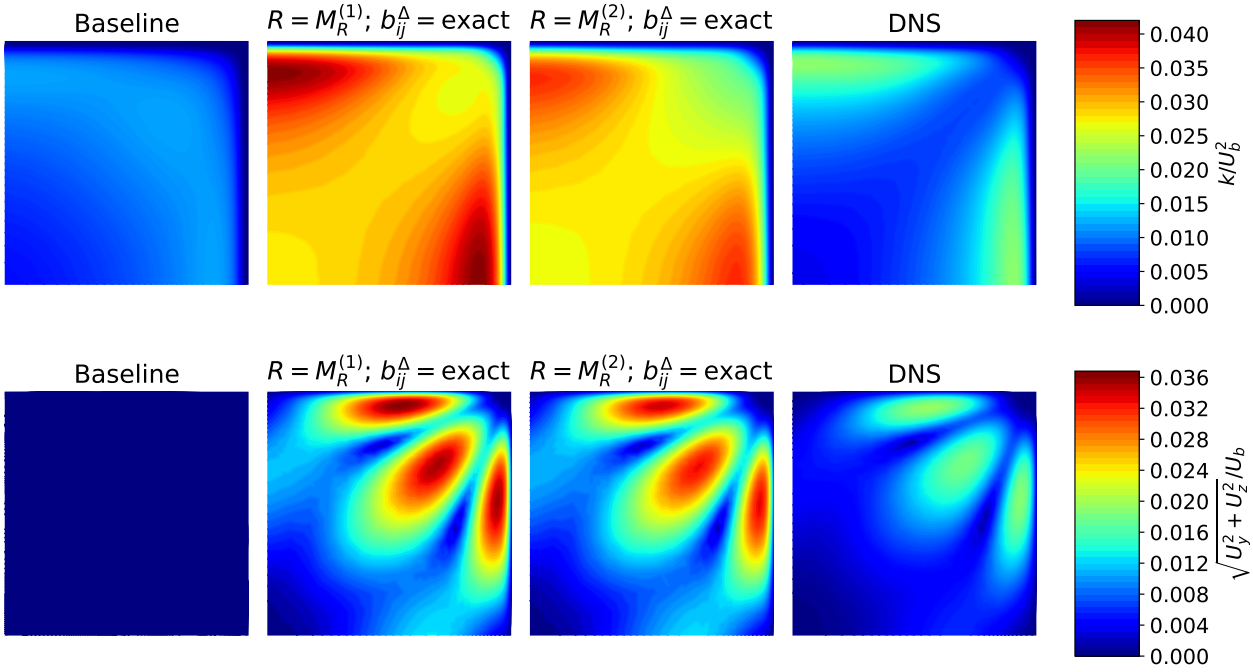


Figure 89: Contours of k and in-plane velocity for the rd1L case for $R = M_R^{(1)}$ (Eq. 99) and $R = M_R^{(2)}$ (Eq. 100) with exact b_{ij}^{Δ}, with baseline (k-ω SST) and DNS added for comparison.

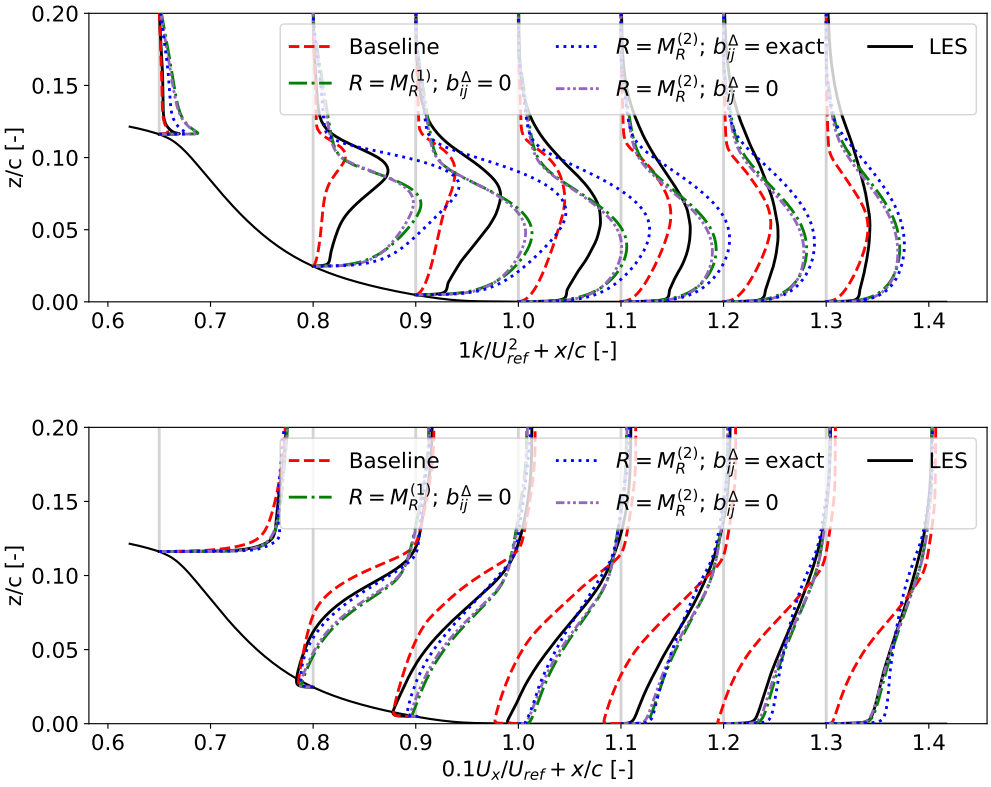


Figure 90: Profiles of k and x-velocity for the hump case for $R = M_R^{(2)}$ (Eq. 100) with exact and zero b_{ij}^{Δ}, with baseline (k-ω SST), LES and $R = M_R^{(1)}$ (Eq. 99) with zero b_{ij}^{Δ} added for comparison.

Consider the duct k contours in Fig. 89; the model still overpredicts k, though slightly less than $M_R^{(1)}$. This suggests that an R model with an even higher R^2 may give a good *a-posteriori* match with DNS. The overprediction of k again results in the overprediction of in-plane velocity magnitude, but slightly less than $M_R^{(1)}$. A similar overprediction of k and in-plane velocity magnitude is observed for the other duct cases. All in all, it is encouraging that the addition of a second basis improves *a-posteriori* results, but the two bases model is still not sufficiently accurate for the duct.

Now consider the hump profiles in Fig. 90; also here $M_R^{(2)}$ with exact b_{ij}^{Δ} overpredicts k and U_x, though predictions are slightly better than those of $M_R^{(1)}$ with exact b_{ij}^{Δ} (see Fig. 88). For $M_R^{(1)}$, significant improvements are found when using zero b_{ij}^{Δ} instead, so profiles of $M_R^{(2)}$ with zero b_{ij}^{Δ} are also shown in Fig. 90. Indeed, also for $M_R^{(2)}$ both k and U_x are predicted significantly better with zero b_{ij}^{Δ}. To better compare $M_R^{(2)}$ with $M_R^{(1)}$, profiles of $M_R^{(1)}$ with zero b_{ij}^{Δ} are also added to Fig. 90. Evidently, $M_R^{(2)}$ consistently predicts k and U_x better than $M_R^{(1)}$, but only slightly. This is not surprising, as the R^2 of $M_R^{(2)}$ is only slightly higher.

The disappointing *a-posteriori* results should not take away from the seeming universal nature of $M_R^{(2)}$. The fact that two different flow types regress the same coefficients potentially means that a physical phenomenon is behind this correction. Unfortunately, the generalized model in its current form does not seem to approximate this underlying phenomenon well *a-posteriori*. In an attempt to improve *a-posteriori* predictions, the strategy of adding more bases is continued. Unfortunately, no agreement for the optimal third basis is found between cases and regressed coefficients vary significantly. Hence, further efforts instead focus on multiplying the bases by (functions of) the dimensionless, scalar features/invariants used by Steiner et al. (see Sec. 3.3.4).

12.1.3 SpaRTA model

As explained in the previous section, an R model with an even better fit is needed for sufficiently accurate *a-posteriori* predictions. The aim of this section is to improve the fit by using the SpaRTA framework to train R models that contain the dimensionless, scalar features/invariants used by Steiner et al. (see Sec. 3.3.4). The vortex stretching feature q_V is added to these, since the duct and roughness are 3D. From experience, models that contain A_k and/or A_p invariants (dependent on $\nabla_i k$ and/or $\nabla_i p$) are highly unstable, so these are excluded from the training. Furthermore, some q-features are not Galilean invariant, contain the pressure gradient or have low mutual information for one or more flows, these are excluded as well. In the end, the five Pope invariants [36] as well as q_γ, q_V, q_Q and q_ν are used for training, their equation is given in Tab. 30. Note that hereafter, these will all be referred to as features (including the Pope invariants). Exponents of these features as well as the tanh, rdiv and exp functions are included in the library. Interestingly, these functions do not appear in any models, likely because the internal function coefficients are not regressed, as further discussed in Sec. 12.2.3.

Table 30: Features used in the model training and their equation.

Feature name	Equation
S^2	$s_{ij}s_{ij}$
ω^2	$\omega_{ij}\omega_{ij}$
S^3	$s_{ij}s_{jk}s_{ki}$
$\omega^2 S$	$\omega_{ij}\omega_{jk}s_{ki}$
$\omega^2 S^2$	$\omega_{ij}\omega_{jk}s_{kl}s_{li}$
q_γ	$\left\|\left\|\frac{\partial u_i}{\partial x_j}\right\|\right\| \Big/ \frac{\epsilon}{k}$
q_ν	$\frac{\nu_t}{100\nu}$
q_Q	$\frac{\omega_{ij}\omega_{ij} - s_{ij}s_{ij}}{2s_{ij}s_{ij}}$
q_V	$\frac{\sqrt{\left(\epsilon_{jlm}\nabla_l u_m\right)\frac{\partial u_i}{\partial x_j}\left(\epsilon_{kno}\nabla_n u_o\right)\frac{\partial u_i}{\partial x_k}}}{s_{pq}s_{pq}}$

For each duct case as well as the hump case, the following model form appears as the best 2-term fit: $C_0\epsilon + C_1 q_\nu^n G_1$, where n is found as either 1 or 2. The difference in exponent does not take away from the generality of the model; $n = 1$ gives a good fit for all cases, so it is used in the generalized model form. Also, the fact that this model form is so close to the model form found in the previous section is encouraging. For the roughness, the best 2-term model form differs significantly per case and no improvement of R^2 is found over the model form in the previous section. The model form for the duct and hump is studied further, C_0 and C_1 are regressed for each case, the results are given in Tab. 31 together with the R^2 of the fit.

Table 31: Fitted C_0 and C_1 in $R = C_0\epsilon + C_1 q_\nu G_1$ and R^2 of the fit.

Rectangular duct				**Heterogeneous roughness**			
Case	C_0	C_1	R^2	**Case**	C_0	C_1	R^2
rd1L	0.0819	-2.26	0.9978	hr00	0.0465	-1.35×10^{-4}	0.616
rd1H	0.0816	-1.86	0.9988	hr03	0.0371	-6.36×10^{-5}	0.676
rd3L	0.0818	-2.19	0.9988	hr04	0.0497	-1.47×10^{-4}	0.585
rd3H	0.0813	-1.63	0.9986	hr06	0.0441	-8.91×10^{-5}	0.812
rd5L	0.0818	-2.16	0.9989	hr07	0.0455	-5.20×10^{-5}	0.923
rd7L	0.0817	-2.15	0.9989	hr08	0.0592	-1.84×10^{-4}	0.754
rd10L	0.0817	-2.15	0.9989	hr09	0.0405	-8.01×10^{-5}	0.603
rd14L	0.0817	-2.15	0.9989	hr10	0.0393	-7.21×10^{-5}	0.653
Hump				hr13	0.0447	-9.91×10^{-5}	0.779
Case	C_0	C_1	R^2	hr15	0.0450	-5.45×10^{-5}	0.931
hump	0.0790	-0.0146	0.972	hr16	0.0461	-4.96×10^{-5}	0.939

In Tab. 31, there is excellent agreement between duct cases for C_0 and C_0 is close to its value for the two bases model (see Tab. 29). Due to the introduction of q_ν to the second term, C_1 is significantly more negative than for the two bases model. Furthermore, C_1 of the high Reynolds number ducts is now significantly less negative than C_1 of the low Reynolds number ducts, likely stemming from the fact that q_ν increases with Reynolds number. This is further confirmed by the fact that C_1 is much less negative for the hump, which has a significantly higher Reynolds number than the ducts. Unfortunately, this makes the model in its current form Reynolds number dependent, so not general. Nonetheless, the simple addition of q_ν to the G_1 term has significantly increased the fit R^2. Meanwhile, models of similar simplicity with other features do not improve the R^2 over the two bases model in Eq. 100.

Given that the q_ν model is the only simple SpaRTA model that gives a significant fit improvement for the ducts, it is further tested despite its Reynolds number dependence. Due to the variance in coefficients, the generalized model is based only on the three highest aspect ratio duct cases. This is because these have the highest R^2 and the largest amount of data points, the resulting R model $M_R^{(3)}$ is:

$$M_R^{(3)} = 0.0817\epsilon - 2.15 q_\nu G_1. \tag{101}$$

The C_1 of this model is likely too negative for the high Reynolds number cases, this is further assessed in testing. The model is tested for all flows, but the hump and roughness cases do not converge, undoubtedly due to the model's Reynolds number dependence. Contours of k and in-plane velocity are shown for the rd1L and rd3H case in Fig. 91 and Fig. 92 respectively, together with baseline and DNS.

With the new model, the rd1L case almost perfectly matches the k of the DNS as seen in Fig. 91. The only discrepancy is a slight overprediction of k near the diagonal. Surprisingly, this small overprediction of k results in a notable overprediction of the magnitude of the secondary motions, this effect is further explained in the next paragraph. Nonetheless, the magnitude is predicted much better than earlier R models (see Fig. 87 and Fig. 89). Similar predictive ability is observed for larger aspect ratio cases with the same Reynolds number, their contour plots are not shown for brevity.

Next, consider the k contours of the higher Reynolds number rd3H case in Fig. 92. Now k is slightly underpredicted in most of the domain, while it is slightly overpredicted in the top-right corner. The underprediction likely stems from a too negative C_1, making the second model term more negative, resulting in a smaller R and thus a too small P_k. Despite k being largely underpredicted, the in-plane velocity magnitude is overpredicted. This is explained by the overprediction of k in the top right corner. This is where streamwise vorticity is generated (see Sec. 6.2), which depends on gradients of the Reynolds stress tensor. The overprediction of k leads to an overprediction of the correction to the Reynolds stress tensor ($k \cdot b_{ij}^{\Delta}$). All in all, the $M_R^{(3)}$ model performs extremely well for the duct. However, its Reynolds number dependence makes it unsuitable as a general model as it does not converge for higher Reynolds number cases.

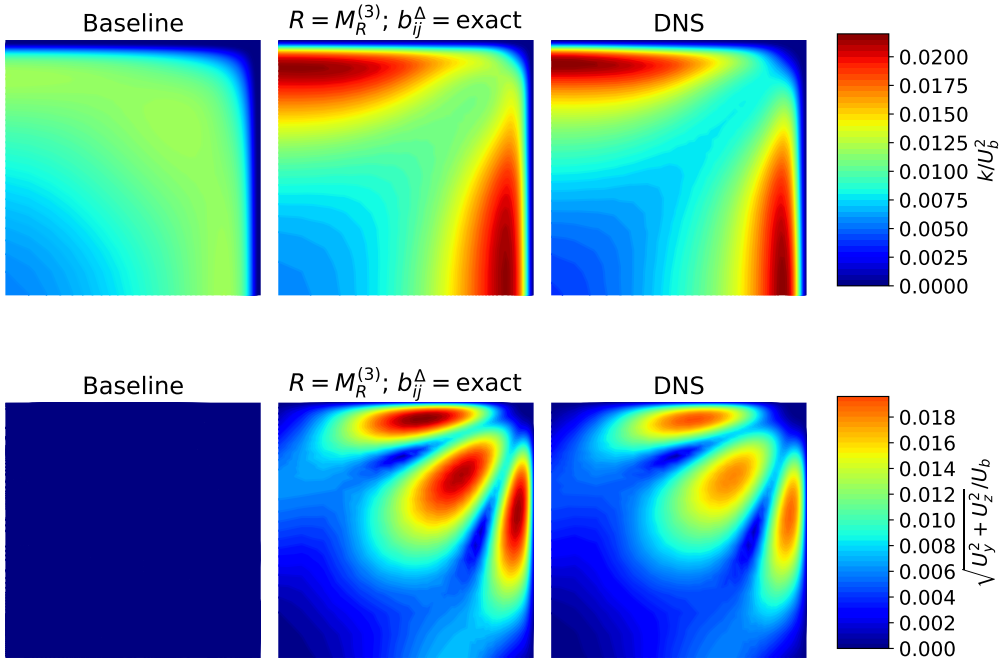


Figure 91: Contours of k and in-plane velocity for the rd1L case for $R = M_R^{(3)}$ (Eq. 101) with exact b_{ij}^{Δ}, with baseline (k-ω SST) and DNS added for comparison.

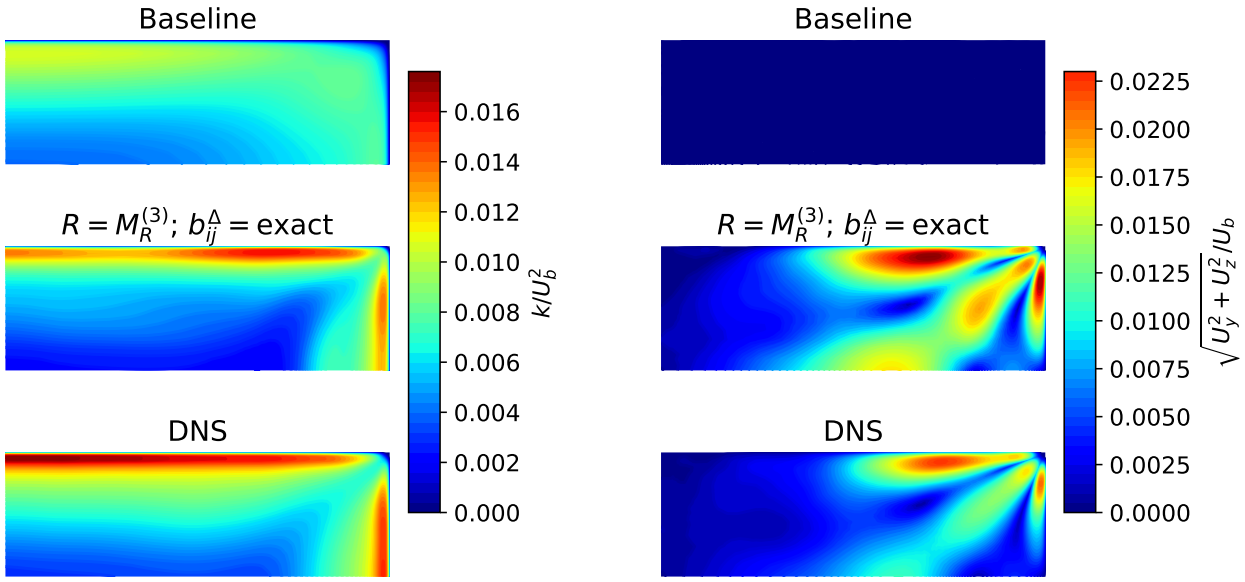


Figure 92: Contours of k and in-plane velocity for the rd3H case for $R = M_R^{(3)}$ (Eq. 101) with exact b_{ij}^{Δ}, with baseline (k-ω SST) and DNS added for comparison.

12.1.4 Manual CFD-driven model refit

So far, efforts to increase *a-posteriori* predictions of R models have focused on increasing the *a-priori* fit with the exact frozen R field. While this has produced an accurate model for the duct, generality with the hump has been lost. The approach in the current section goes back to the $R = C_0\epsilon$ model form discussed in Sec. 12.1.1, for which the model $M_R^{(1)}$ in Eq. 99 was regressed. The error of this model mostly stems from its overprediction of the magnitude of k; the shape of k is predicted relatively well. Thus, it is speculated that a lower value of C_0 exists which yields better predictions of k and in-plane velocity. This alternate C_0 value cannot be found *a-priori* with the current tools, so the CFD solver has to be included in the regression.

Implementing a CFD-driven optimizer is beyond the scope of the current work, so C_0 optimized manually for the rd1L case as a proof-of-concept. Manual optimization is only feasible as there is just one model coefficient, the runtime of the rd1L case is ∼1 minute and a reasonable initial guess is available. As this is just a proof-of-concept, no rigorous minimization criterion is used, nor a proper updating algorithm. Instead, the case is run with various values of C_0 until a value is found that visually gives the optimal match with DNS in terms of the in-plane velocity contours. The manual optimization is only performed on the rd1L case as the other cases have significantly longer run times. The hope is that the *a-posteriori* optimized C_0 generalizes as well as the *a-priori* fit of C_0 to the duct and hump (see Tab. 28).

The manual CFD-driven procedure gives $C_0 \approx 0.043$, resulting in the model $M_R^{(4)}$:

$$M_R^{(4)} = 0.043\epsilon. \tag{102}$$

Encouragingly, this model converges for all cases (including the roughness), likely due to its lower C_0 compared to $M_R^{(1)}$. Contours of k and in-plane velocity magnitude are shown for the the rd1L and rd3H case in Fig. 93 and Fig. 94 respectively. Furthermore, profiles of k and U_x are shown for the hump in Fig. 95, also including a run with $b_{ij}^{\Delta} = 0$ as this previously gave improvements over the exact b_{ij}^{Δ}. The best hump model found so far ($M_R^{(2)}$ with zero b_{ij}^{Δ}) is added as well. Next, results of roughness case hr00 are shown in Fig. 96

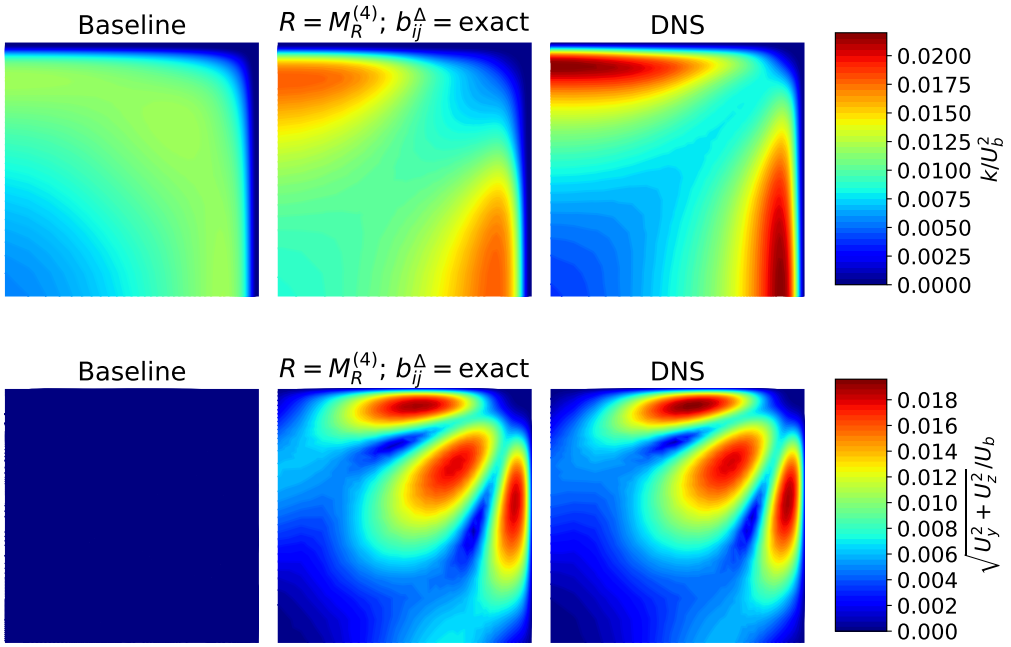


Figure 93: Contours of k and in-plane velocity for the rd1L case for $R = M_R^{(4)}$ (Eq. 102) with exact b_{ij}^{Δ}, with baseline (k-ω SST) and DNS added for comparison.

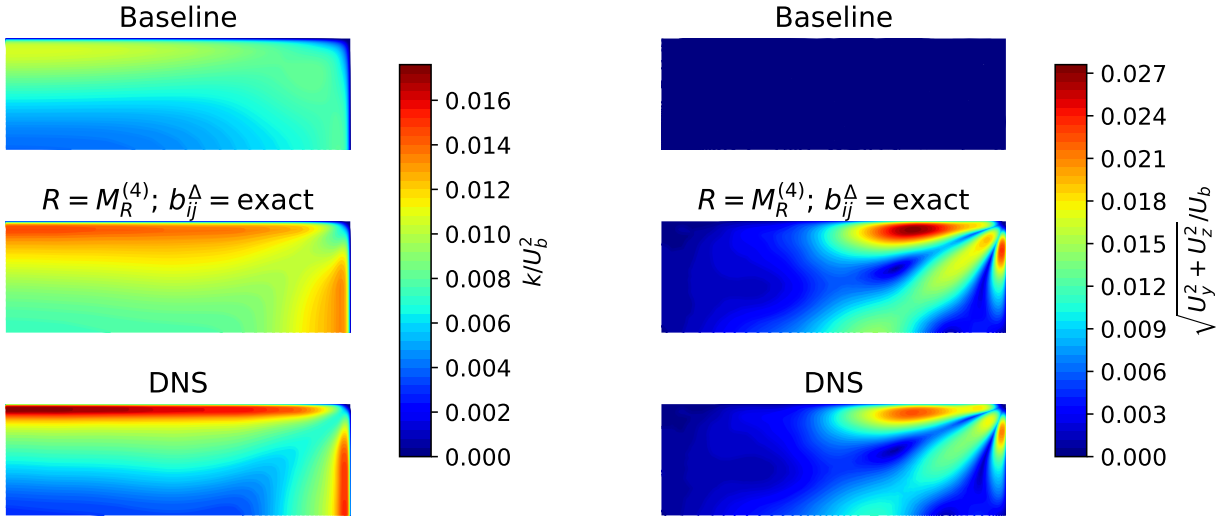


Figure 94: Contours of k and in-plane velocity for the rd3H case for $R = M_R^{(4)}$ (Eq. 102) with exact b_{ij}^{Δ}, with baseline (k-ω SST) and DNS added for comparison.

Consider k contours of the duct cases in Fig. 93 and Fig. 94; peak values of k are underpredicted, while k at the diagonal is overpredicted (this is worse for the rd3H case). Nonetheless, k is predicted approximately as well as by the $M_R^{(3)}$ model and much better than by the $M_R^{(1)}$ and $M_R^{(2)}$ models. As mentioned, optimization of C_0 is based on the in-plane velocity contours rather than k. This is why the overall in-plane velocity magnitude is predicted perfectly for the rd1L case. For the rd3H case, which is not included in the CFD optimization, in-plane velocity magnitude is slightly overpredicted.

This overprediction of in-plane velocity magnitude can be explained by the overprediction of k in the corner. As explained in Sec. 6.2, the secondary motions are generated in the corner by the Reynolds stress tensor (RST). The correction to the RST is overpredicted in the corner as it is the product of b_{ij}^{Δ} (which is exact) and k (which is overpredicted). A similar slight overprediction of in-plane velocity magnitude is observed for all other duct cases (except rd1L). Perhaps a smaller C_0 would give better overall in-plane velocity predictions, however, each case is already predicted well (even better than with $M_R^{(3)}$) and the current section is just a proof-of-concept.

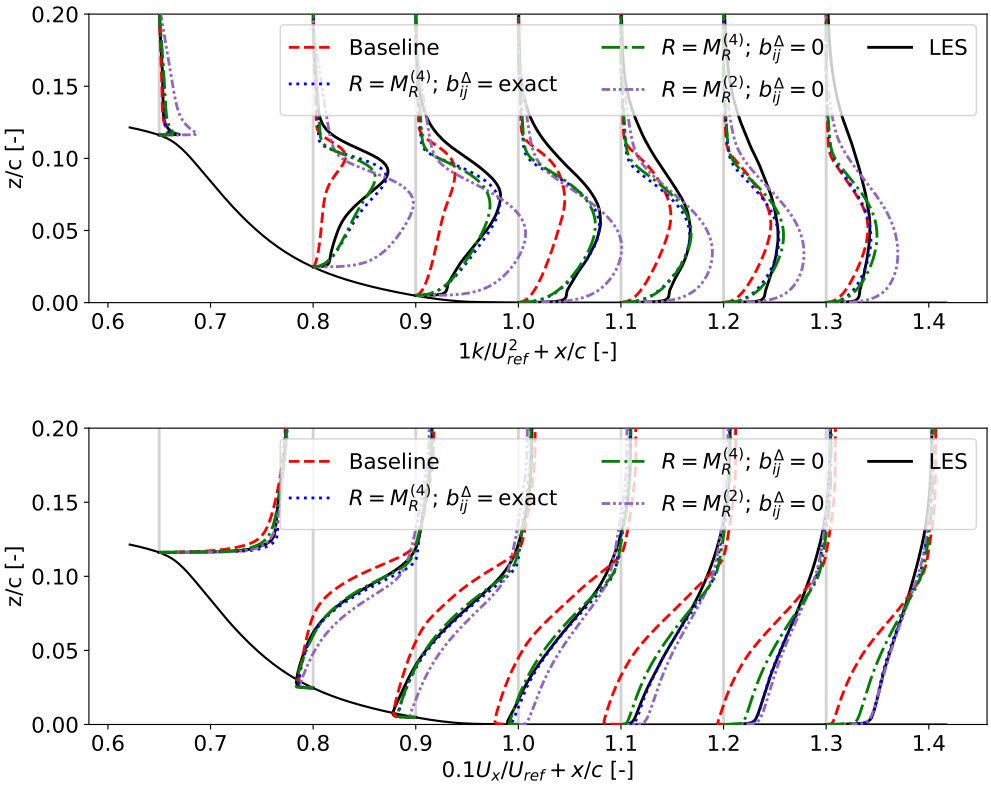


Figure 95: Profiles of k and x-velocity for the hump case for $R = M_R^{(4)}$ (Eq. 102) with exact and zero b_{ij}^{Δ}, with baseline (k-ω SST), LES and $R = M_R^{(2)}$ (Eq. 100) with zero b_{ij}^{Δ} added for comparison.

As can be seen in Fig. 95, k is also slightly underpredicted for the hump. Still, it is predicted much better than both baseline and $M_R^{(2)}$ with zero b_{ij}^{Δ}. Astonishingly, U_x is predicted almost exactly by $M_R^{(4)}$, even though the hump is not considered in the *a-posteriori* optimization of C_0. This points towards a universal nature of the simple $R = C_0\epsilon$ model form, as the hump is a significantly different flow than the duct. For $M_R^{(1)}$ and $M_R^{(2)}$, using $b_{ij}^{\Delta} = 0$ resulted in significantly better predictions of k and U_x. However, for $M_R^{(4)}$ it results in slightly worse prediction of k for $x < 1.0c$ and slightly worse prediction of U_x for $x > 1.0c$. Still, if no better b_{ij}^{Δ} model is found, using $b_{ij}^{\Delta} = 0$ gives good improvements over baseline.

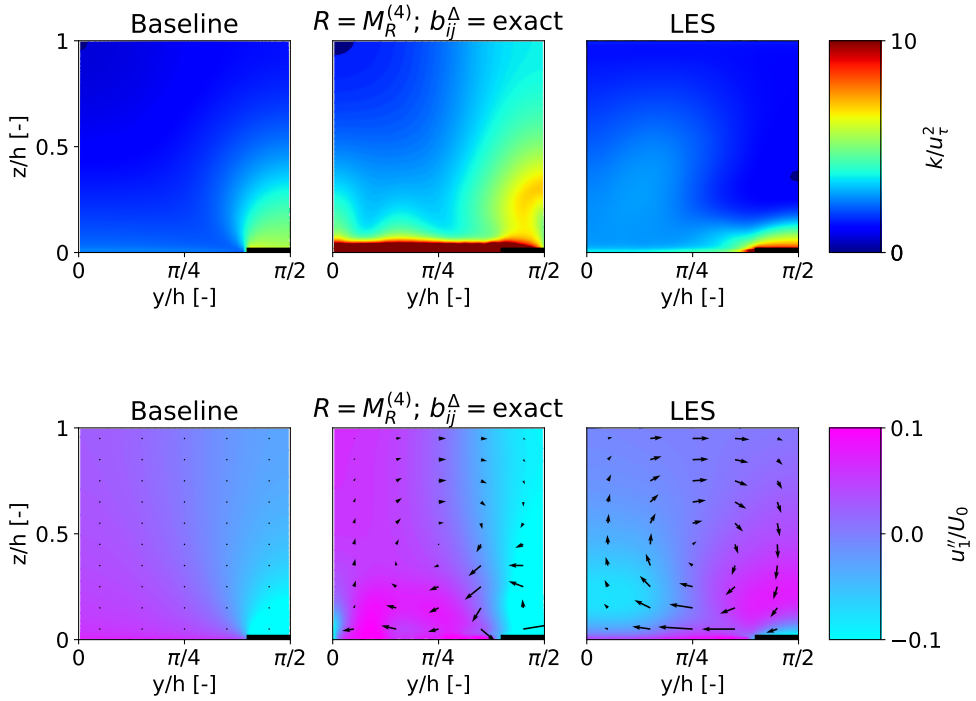


Figure 96: Contours of k and dispersive x-velocity (see Eq. 74) with the in-plane velocity vector field overlaid for the for the hr00 case for $R = M_R^{(4)}$ (Eq. 102) with exact b_{ij}^{Δ}, with baseline (k-ω SST) and LES added for comparison.

Now consider the k contours of the hr00 case in Fig. 96; k is now significantly overpredicted. Surprisingly, also the shape of k is predicted wrongly, as there is far too much k above the smooth part of the wall. As mentioned, wall models are used as boundary conditions for the wall, but these are not modified by SpaRTA. Perhaps these wall models also need to incorporate the corrections, though answering this question is beyond the scope of the current work. Next consider the velocity contours of the roughness; a large in-plane vortex is visible, however, in-plane velocity is misaligned especially near the wall. Furthermore, dispersive x-velocity is mispredicted throughout the domain. Given the uncertainties of the roughness case, these results do not immediately discard the generality of $M_R^{(4)}$. However, it clearly is not ready-to-implement as is and further research is required into its interaction with wall models. Also, more cases should be added to confirm its generality.

All in all, the manual *a-posteriori* optimization of the single basis model has yielded the most simple, general and accurate model so far. An even better model could likely be attained by applying the procedure to $M_R^{(2)}$ (Eq. 100). However, manual optimization of two coefficients is much more complicated than just one. Furthermore, the current single basis model is already sufficiently accurate (at least for the duct and hump). The analysis presented clearly shows that the frozen nature of k-corrective-frozen is limiting model fits. Hence, for future research, it is recommended to implement the *a-posteriori* optimization approach (also known as CFD-driven) in an OpenFOAM solver. This requires a minimization algorithm and criterion; the minimization criterion should ideally be based on the velocity field.

12.2 Isolated b_{ij}^{Δ} model training and testing

Following earlier work, models for b_{ij}^{Δ} are comprised of Pope's ten base tensors $T_{ij}^{(\lambda)}$ (see Sec. 2.6) multiplied by functions of scalar invariants/features q (see Sec. 3.3.4). Contrary to literature, these functions are referred to as $f_\lambda(q)$ to avoid confusion with the bases of R models. The objective of the current section is to train expressions for $f_\lambda(q)$ and test these *a-posteriori*, striving for simple and general b_{ij}^{Δ} models. To isolate the effect of the b_{ij}^{Δ} model, the exact frozen R field is used in testing. The section is structured as follows: A model that is a simple linear combination of bases is trained in Sec. 12.2.1. To improve accuracy, the established SpaRTA framework is then used to train a model that also contains functions of features in Sec. 12.2.2. Unfortunately, accuracy does not improve when using SpaRTA, which is found to come from its limited functional forms. Thus, in Sec. 12.2.3, the newly developed CuRTA framework is used to regress a model with an expanded functional form.

12.2.1 One and two bases models

Inspired by the success of the simple one basis R model laid out in Sec. 12.1.4, a similar form is sought for a b_{ij}^{Δ} model. This means that the model will have the form $b_{ij}^{\Delta} = C_0 T_{ij}^{(n)}$, with C_0 a coefficient and $T_{ij}^{(n)}$ the nth Pope base tensor. To find the base tensor giving the best model, C_0 is fitted for each base tensor for each case. For the regression, volume weighing is again used but now k weighing is also included. This is because b_{ij}^{Δ} is first multiplied by k before being added to the Reynolds stress tensor. The coefficient of determination (R^2) is shown for the regression of each base tensor for each case in Fig. 97. The second, seventh and eighth base tensor provide significantly better fits than the other tensors, but these three perform similar to each other. The second base tensor is chosen for further testing, as it contains the lowest powers of $\nabla_i u_j$, making it more stable.

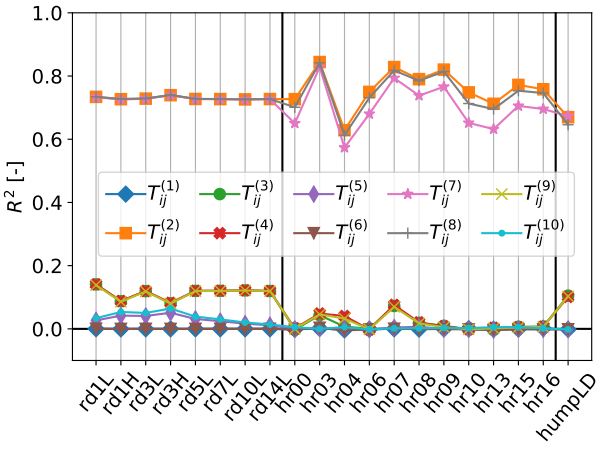


Figure 97: Coefficient of determination of the fit $b_{ij}^{\Delta} = C_0 T_{ij}^{(n)}$ for each base tensor for each case.

The C_0 resulting from fitting $b_{ij}^{\Delta} = C_0 T_{ij}^{(2)}$ is given for each case in Tab. 32, along with the fit R^2. There is a significant variance in C_0 between duct cases and the R^2 is much lower than for the single basis R model (see Tab. 28). Furthermore, there is no generality between the duct and hump as the hump has a much lower C_0 than any duct case. For the roughness, variance in C_0 between cases and fit R^2 is similar to the single basis R model, which performs poorly. Given the variance in C_0, a generalized model is not constructed; each case is first tested with its own optimized C_0. Unfortunately, not a single case converges, despite this tailored C_0 value.

Table 32: Fitted C_0 in $b_{ij}^{\Delta} = C_0 T_{ij}^{(2)}$ and R^2 of fit.

Rectangular duct			**Heterogeneous roughness**		
Case	C_0	R^2	**Case**	C_0	R^2
rd1L	0.601	0.734	hr00	0.419	0.727
rd1H	0.447	0.727	hr03	0.409	0.844
rd3L	0.533	0.729	hr04	0.392	0.629
rd3H	0.413	0.739	hr06	0.422	0.750
rd5L	0.530	0.727	hr07	0.472	0.828
rd7L	0.528	0.727	hr08	0.426	0.790
rd10L	0.531	0.726	hr09	0.469	0.820
rd14L	0.525	0.727	hr10	0.448	0.748
Hump			hr13	0.410	0.712
Case	C_0	R^2	hr15	0.428	0.772
hump	0.307	0.670	hr16	0.424	0.758

The nonconvergence under the single basis b_{ij}^{Δ} model likely comes from the low R^2 of the fits. To address this, the fit is improved by adding another basis, giving the model form $b_{ij}^{\Delta} = C_0 T_{ij}^{(n_1)} + C_1 T_{ij}^{(n_2)}$ with $n_1 \neq n_2$. A similar analysis as in Fig. 97 is performed, but now on pairs of base tensors. Given that there are 45 unique pairs of base tensors, the plot is not shown here. A similar trend is found, where a group of base tensor pairs has a much higher R^2 than the other base tensor pairs. From this high R^2 group, the lowest numbered tensor pair should be chosen, as these tensors contain the lowest powers of $\nabla_i u_j$, giving better stability. The pair $(T_{ij}^{(2)}, T_{ij}^{(3)})$ is the lowest numbered pair that has an R^2 within 5% of the highest R^2 of each case, resulting in the model form $b_{ij}^{\Delta} = C_0 T_{ij}^{(2)} + C_1 T_{ij}^{(3)}$. This model form is regressed for each case, the resulting coefficients are given in Tab. 33 together with the fit R^2.

Table 33: Fitted C_0 and C_1 in $b_{ij}^{\Delta} = C_0 T_{ij}^{(2)} + C_1 T_{ij}^{(3)}$ and R^2 of fit.

Rectangular duct				**Heterogeneous roughness**			
Case	C_0	C_1	R^2	**Case**	C_0	C_1	R^2
rd1L	0.596	0.884	0.863	hr00	0.419	-0.00159	0.727
rd1H	0.444	0.516	0.806	hr03	0.405	0.248	0.869
rd3L	0.529	0.726	0.839	hr04	0.391	-0.130	0.636
rd3H	0.411	0.455	0.813	hr06	0.422	0.0632	0.751
rd5L	0.527	0.727	0.840	hr07	0.467	0.416	0.880
rd7L	0.526	0.731	0.841	hr08	0.424	0.179	0.802
rd10L	0.529	0.743	0.843	hr09	0.469	0.175	0.831
rd14L	0.523	0.731	0.844	hr10	0.448	0.0312	0.749
Hump				hr13	0.410	-0.0382	0.713
Case	C_0	C_1	R^2	hr15	0.427	0.119	0.777
hump	0.307	0.379	0.777	hr16	0.424	0.120	0.764

Surprisingly, the introduction of the second basis has partially generalized the regressed coefficients of the duct, as seen in Tab. 33. Partially, because coefficients only generalize for the rd3L, rd5L, rd7L, rd10L and rd14L case, suggesting the coefficients are a function of the Reynolds number. The different coefficients for the rd1L case are thought to originate from its relatively small proportion of low-b_{ij}^{Δ} cells. This skews the rd1L model to fit high-b_{ij}^{Δ} cells, resulting in the relatively high coefficients. The R^2 of each duct case and the hump significantly increased compared to the single basis model in Tab. 32, though it is still much lower than for the single basis R model in Tab. 28. Next, the duct model does not seem to generalize to the hump given the hump's much lower coefficients, likely from the supposed Reynolds number dependence of the coefficients. For the roughness, adding the second basis did not notably increase R^2. Furthermore, there is no generality at all for C_1 between cases. Hence, it is unlikely that this second basis will improve convergence of the roughness cases.

The two bases model is tested for each case, again the case specific coefficients are used, given the lack of generality and prior convergence issues. Only the rd1L and rd1H case converge, likely due to their aforementioned small proportion of low-b_{ij}^{Δ} cells, their models are formalized as $M_{b^{\Delta}}^{(1)}$ and $M_{b^{\Delta}}^{(2)}$ respectively:

$$M_{b^{\Delta}}^{(1)} = 0.596T_{ij}^{(2)} + 0.884T_{ij}^{(3)}, \qquad (103) \qquad\qquad M_{b^{\Delta}}^{(2)} = 0.444T_{ij}^{(2)} + 0.516T_{ij}^{(3)}. \qquad (104)$$

For the rd1L case, contours of in-plane velocity are shown in Fig. 98, together with baseline and DNS. There is a definite improvement over baseline, as two corner vortices are present. However, they are located far too close to the duct's center. The same mismatch is observed for the rd1H case. All in all, the two bases model is not general, inaccurate and only converges for two cases. Adding more bases does not increase the fit R^2, so further efforts focus on replacing the coefficients by (functions of) the dimensionless, scalar features/invariants used by Steiner et al. (see Sec. 3.3.4). This is further motivated by the fact that the coefficients seem to be Reynolds number dependent.

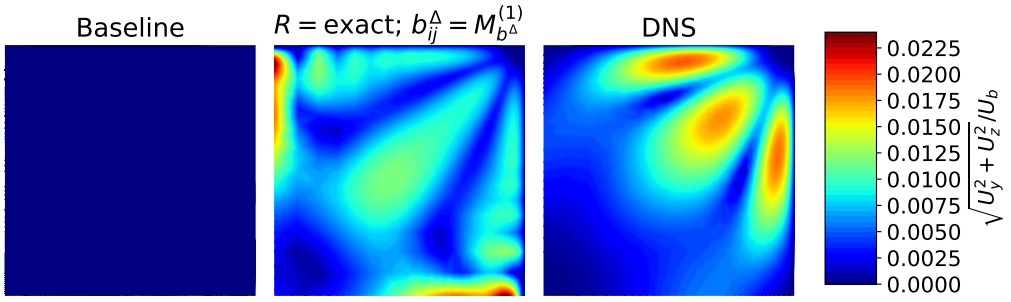


Figure 98: Contours of in-plane velocity for the rd1L case for $b_{ij}^{\Delta} = M_{b^{\Delta}}^{(1)}$ (Eq. 103) with exact R, with baseline (k-ω SST) and DNS added for comparison.

12.2.2 SpaRTA model

The already established SpaRTA framework is used in the first attempt to generate more accurate models. The duct cases are the primary focus here, as they are dominated by anisotropy and a proper R model is established (contrary to the roughness). Weighing by volume and k and using the features in Tab. 30, SpaRTA identifies models of the form $b_{ij}^{\Delta} = f_1(q_\nu)T_{ij}^{(2)} + f_2(q_\nu)T_{ij}^{(3)}$. This is exactly the form desired, as it contains the same base tensors as the model in the prior section, but replaces the coefficients by a function of a feature. It is encouraging that these are functions of q_ν, as this introduces some form of Reynolds number dependence (though not necessarily the correct form). For the low-Reynolds number duct cases, f_1 and f_2 are rdiv functions, defined as:

$$\text{rdiv}(q) = \frac{q}{C + q^2}, \tag{105}$$

where C is a constant. For the high-Reynolds number duct cases, f_2 is also an rdiv function, but f_1 is a fourth root. For the roughness and hump, functions of q_γ mostly appear, though there is little agreement in functional form and other features often appear as well.

In the end, the model form $b_{ij}^{\Delta} = C_0\text{rdiv}(q_\nu)T_{ij}^{(2)} + C_1\text{rdiv}(q_\nu)T_{ij}^{(3)}$ is chosen for the general model, as the duct is of main interest for b_{ij}^{Δ} model training and q_ν may give the right Reynolds number dependence. Furthermore, the rdiv function is used for both terms, as it has asymptotic behaviour (contrary to the fourth root). Usually, the constant C in the rdiv function in Eq. 105 would be regressed, however, SpaRTA is linear meaning it can only regress the coefficient before each term. As an informed guess for C, SpaRTA uses the case variance of q (in this case of q_ν). For the general model, $C = 0.00327$ is used for all cases, which is based on the rd14L case given its coefficient generality observed in Tab. 33. Whereas SpaRTA uses Ridge regression to get small coefficients when refitting models, this is not considered necessary for the current two-term model, so ordinary weighted least squares is used. The model coefficients C_0 and C_1 are regressed for each case, they are given in Tab. 34 along with the fit R^2.

Table 34: Fitted C_0 and C_1 in $b_{ij}^{\Delta} = C_0\frac{q_\nu}{0.00327+q_\nu^2}T_{ij}^{(2)} + C_1\frac{q_\nu}{0.00327+q_\nu^2}T_{ij}^{(3)}$ and R^2 of fit.

Rectangular duct				**Heterogeneous roughness**			
Case	C_0	C_1	R^2	**Case**	C_0	C_1	R^2
rd1L	0.0797	0.118	0.782	hr00	252	-80.9	0.222
rd1H	0.07913	0.0952	0.843	hr03	263	-22.1	0.252
rd3L	0.0768	0.106	0.806	hr04	250	-117	0.214
rd3H	0.0821	0.0942	0.869	hr06	255	-61.4	0.227
rd5L	0.0765	0.106	0.811	hr07	288	33.6	0.206
rd7L	0.0764	0.106	0.814	hr08	247	-32.5	0.235
rd10L	0.0762	0.107	0.813	hr09	283	-24.4	0.192
rd14L	0.0763	0.106	0.819	hr10	269	-67.6	0.203
Hump				hr13	258	-89.0	0.227
Case	C_0	C_1	R^2	hr15	269	-43.0	0.224
hump	0.186	0.136	0.145	hr16	268	-36.7	0.226

Comparing the duct R^2 between Tab. 34 and Tab. 33, it becomes clear that the main goal of increasing the fit R^2 has not been achieved. For the duct, R^2 decreased slightly while for the hump and the roughness it decreased substantially. Nonetheless, the goal of removing Reynolds number and aspect ratio dependence of the coefficients has been partially achieved. For instance, the difference in C_0 between the rd3H and rd3L case went from 22% to 6.5%. Unfortunately, this is only true within duct cases, as the coefficient discrepancy only grew with respect to the hump and roughness cases. Within the roughness cases, a similar spread in C_0 is observed and C_1 again seems to fit mostly noise.

Despite the lack of R^2 improvement, the new model form is tested for each case. Given the better generality between duct coefficients, a single model is chosen. As reasoned before, high aspect ratio duct cases contain a higher proportion of low-b_{ij}^{Δ} cells, resulting in more stable models. Furthermore, the coefficients seem to converge to a single value with increasing aspect ratio. Hence, the general duct model $M_{b^\Delta}^{(3)}$ is based on the coefficients of the rd14L case:

$$M_{b^\Delta}^{(3)} = \frac{q_\nu}{0.00327 + q_\nu^2}\left(0.0763 T_{ij}^{(2)} + 0.106 T_{ij}^{(3)}\right). \tag{106}$$

This model only converges for the rd1L, rd1H and rd3H duct cases, contours of in-plane velocity are shown for the rd1L case in Fig. 99 together with baseline and DNS. The hump case does not converge, however, all roughness cases surprisingly do converge. Results of roughness case hr00 are shown in Fig. 100 together with baseline and LES.

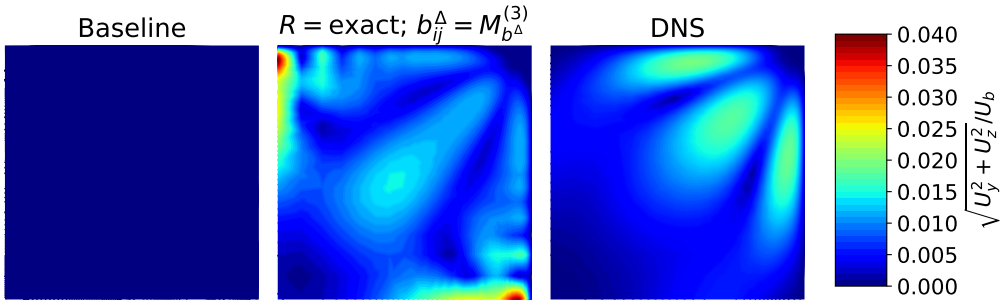


Figure 99: Contours of in-plane velocity for the rd1L case for $b_{ij}^{\Delta} = M_{b^\Delta}^{(3)}$ (Eq. 106) with exact R, with baseline (k-ω SST) and DNS added for comparison.

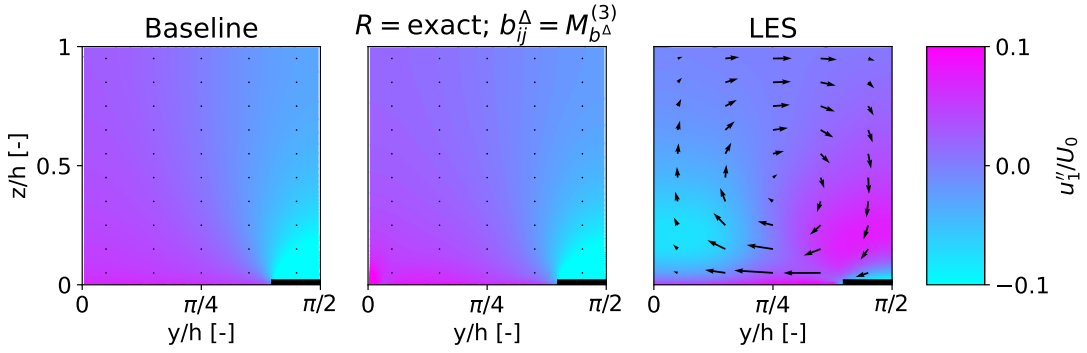


Figure 100: Contours of dispersive x-velocity (see Eq. 74) with the in-plane velocity vector field overlaid for the for the hr00 case for $b_{ij}^{\Delta} = M_{b^\Delta}^{(3)}$ (Eq. 106) with exact R, with baseline (k-ω SST) and LES added for comparison.

Comparing Fig. 99 with Fig. 98, the new SpaRTA $M^{(3)}_{b^\Delta}$ model does not improve duct predictions over $M^{(1)}_{b^\Delta}$, which is not surprising considering the lack of R^2 improvement. Predictions of the new model are even slightly worse, as in-plane velocity at the symmetry is overpredicted even more. The same is observed for the other two converged duct cases (rd1H and rd3H). It should be noted that $M^{(1)}_{b^\Delta}$ is trained on the rd1L case, while $M^{(3)}_{b^\Delta}$ is trained on the rd14L case. The fact that the low aspect ratio cases still manage to predict the two corner vortices using a model trained on a much higher aspect ratio case is encouraging. Still, one would expect R^2 to increase significantly with the more complex model structure enabled by SpaRTA. In Sec. 12.2.3, the limited functional forms in SpaRTA are found to be limiting the fit.

Looking at the hr00 case in Fig. 100, the model b^Δ_{ij} seems to be much smaller than the exact b^Δ_{ij}, as model results are indistinguishable from baseline. This is to be expected, as the roughness has much higher q_ν values than the duct and the rdiv function asymptotes to zero. This is also why the regressed roughness coefficients in Tab. 34 are much larger; the duct coefficients result in a b^Δ_{ij} of almost zero. In principle, this behaviour is acceptable for a generalized model, as one can not expect to improve predictions for every possible case. Thus, going to baseline for cases outside the training regime is an acceptable property.

In an attempt to improve *a-posteriori* results, SpaRTA models with more terms are also tested. Many of these do not even converge for their training case, let alone other cases. It is concluded that increasing the number of model terms will not yield a universal model, only massive overfits to the training data. Hence, in the next section, the R^2 of the fit is attempted to be improved by expanding the available functional forms using the newly developed CuRTA framework.

12.2.3 Nonlinear symbolic regression (CuRTA) model

The lack of improvement in R^2 when using SpaRTA compared to a simple basis fit is speculated to come from its aforementioned linear nature. It is able to use non-linear functions such as rdiv, however, it is only able to regress the coefficient before each term. Thus, coefficients inside nonlinear functions (such as C in Eq. 105) cannot be regressed, they are guessed based on field statistics instead. Furthermore, many functions are missing coefficients that play an important role in their shape and thus their ability to fit the data. For example, the tanh function has the form $C\tanh(C\cdot q)$, meaning it will always be zero if $q=0$. In theory, SpaRTA is able to regress an additional $+C$ term to allow the function to be shifted vertically. However, in practice this term is never observed for simple b^Δ_{ij} models. A much more general functional form would be $C(\tanh(C\cdot q+C)+C)$, such that the function can be shifted vertically and horizontally.

A new framework is developed that is able to regress the coefficients inside non-linear functions using non-linear least-squares. Similar to SpaRTA, it constructs expressions from a library of terms, though it constructs the expression term-by-term. The framework is named CuRTA and is further described in Sec. 4.2. When all base tensors are available to CuRTA, $T^{(8)}_{ij}$ and $T^{(4)}_{ij}$ are selected for the duct models. However, these only give a slight improvement in fit over $T^{(2)}_{ij}$ and $T^{(3)}_{ij}$, so the latter pair is chosen as they contain lower powers of $\nabla_i u_j$, improving stability. Adding a third basis gives a negligible increase in R^2, so only two bases are used. Weighing again by volume and k, CuRTA finds the same model form for each duct case; $b^\Delta_{ij} = C_0(\tanh(C_1 q_\nu + C_2)+C_3)T^{(2)}_{ij} + C_4\left(\tanh(C_5 q_\nu + C_6)+C_7\right)T^{(3)}_{ij}$, the regressed coefficients and R^2 are given in Tab. 35.

For the roughness and hump, slightly different optimal model forms are found by CuRTA. However, these only provide a small increase in R^2 compared to the duct model form; $R^2 = 0.765$ versus $R^2 = 0.744$ for the roughness (case hr00) and $R^2 = 0.791$ versus $R^2 = 0.782$ for the hump. Also for these cases, adding a third basis gives a negligible increase in R^2. Given that a general model is desired, the duct model form is also fitted to the roughness and hump cases. The regressed coefficients are also given in Tab. 35 along with the R^2.

Table 35: Fitted $C_0 - C_7$ in $b_{ij}^{\Delta} = C_0 \left(\tanh\left(C_1 q_\nu + C_2\right) + C_3\right) T_{ij}^{(2)} + C_4 \left(\tanh\left(C_5 q_\nu + C_6\right) + C_7\right) T_{ij}^{(3)}$ and R^2 of fit.

Rectangular duct									
Case	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	R^2
rd1L	0.400	-14.6	1.10	1.51	0.631	-20.5	1.47	1.47	0.986
rd1H	0.477	-8.73	0.383	1.53	0.608	-14.5	0.927	1.42	0.985
rd3L	0.437	-11.0	0.765	1.45	0.605	-18.2	1.25	1.50	0.989
rd3H	0.778	-6.08	-0.154	1.32	0.554	-15.1	0.972	1.50	0.986
rd5L	0.454	-10.5	0.746	1.40	0.591	-19.0	1.31	1.55	0.990
rd7L	0.463	-10.2	0.738	1.36	0.582	-19.5	1.34	1.58	0.991
rd10L	0.465	-10.3	0.772	1.33	0.576	-20.1	1.39	1.61	0.992
rd14L	0.457	-10.3	0.756	1.38	0.567	-20.1	1.38	1.63	0.992
Heterogeneous roughness									
Case	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	R^2
hr00	16.5	-8.93×10^{-4}	13.2	-0.974	0.196	-0.0736	720	-0.198	0.744
hr03	10.3	-1.56×10^{-4}	-2.29	1.03	0.00338	-0.0137	120	75.3	0.883
hr04	0.157	-0.821×10^{-4}	0.676	3.31	0.186	-3.04×10^{-3}	28.4	-0.793	0.663
hr06	0.0309	-3.49×10^{-3}	38.0	13.1	0.193	-3.93×10^{-3}	39.8	0.0494	0.766
hr07	0.0861	-1.71×10^{-3}	1.42	6.29	-21.5	-3.62×10^{-4}	-1.86	0.976	0.896
hr08	9.67	-4.90×10^{-4}	-2.21	1.04	0.104	-0.0677	672	1.22	0.813
hr09	0.0202	-3.70×10^{-3}	43.1	22.6	0.229	-6.01×10^{-3}	65.8	0.518	0.848
hr10	0.0244	-4.71×10^{-3}	47.3	17.7	0.231	-0.0157	145	0.0432	0.770
hr13	0.115	-1.37×10^{-3}	1.29	4.40	0.198	-0.0421	439	-0.343	0.739
hr15	11.9	-5.73×10^{-4}	10.2	-0.963	0.172	-7.13×10^{-3}	81.3	0.383	0.792
hr16	16.3	-3.12×10^{-4}	-2.49	1.02	0.202	-5.97×10^{-3}	67.8	0.344	0.788
Hump									
Case	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	R^2
hump	4.56	-2.32	-1.43	1.07	0.0987	-90.6	6.63	4.84	0.782

Comparing the R^2 of the CuRTA duct models in Tab. 35 with the R^2 of the SpaRTA duct models in Tab. 34, a significant improvement is found. This is evidence that the linear nature of SpaRTA prevents it from properly regressing the data. As before, the regressed duct coefficients seem to converge to one value as aspect ratio increases, fit R^2 also increases with aspect ratio. Some coefficients are similar for all cases (such as C_4), while others are significantly different for the low aspect ratio and high Reynolds number cases (such as C_2). Due to the large number of coefficients, it is difficult to asses the level of Reynolds number dependence, however, it is definitely present.

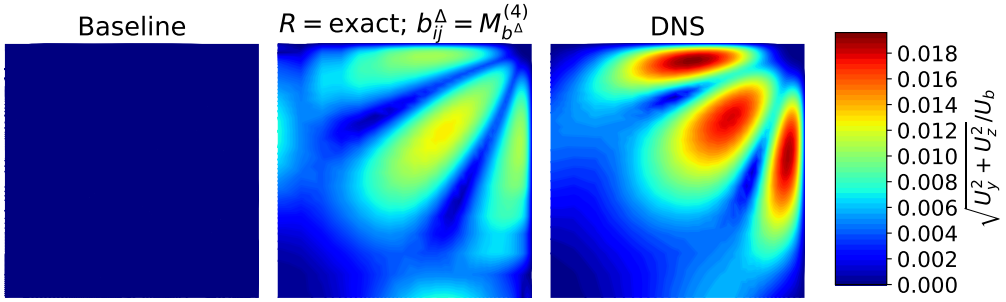
Next, comparing the hump R^2 in Tab. 35 with its two-bases R^2 in Tab. 33, no significant improvement is found. As mentioned, even the best hump model found by CuRTA does not have a much higher R^2. Also, the hump coefficients are quite different from the duct coefficients, indicating no generality between the cases. One explanation for the low hump R^2 is the farfield cells, which have large b_{ij}^{Δ} without influencing the flow, though this should be largely addressed by k weighing. Another option is that the hump's b_{ij}^{Δ} depends on another feature/functional form not in the library. As the main hump error is in P_k rather than b_{ij}^{Δ}, alternative b_{ij}^{Δ} models for the hump are left for further research.

Now compare the roughness R^2 in Tab. 35 with its two bases R^2 in Tab. 33, again no real improvement is found. Furthermore, there seems to be no generality between cases as each coefficient varies orders of magnitude. Also, the coefficients are significantly different from the hump and duct cases. The lack of R^2 improvement is surprising, as similar to the duct, the roughness is dominated by anisotropic effects. It is speculated that the use of wall functions and insufficient mesh refinement near the transition from smooth to rough wall are the cause, as high gradients of b_{ij}^{Δ} are observed near these locations. As explained in Sec. 7.3, the LES is sufficiently accurate to generate frozen fields on a finer RANS mesh, but this is left for further research. Furthermore, applying k-corrective-frozen to the Reynolds force vector also seems to increase accuracy, as observed by Amarloo et al. [2], this is also left for further research.

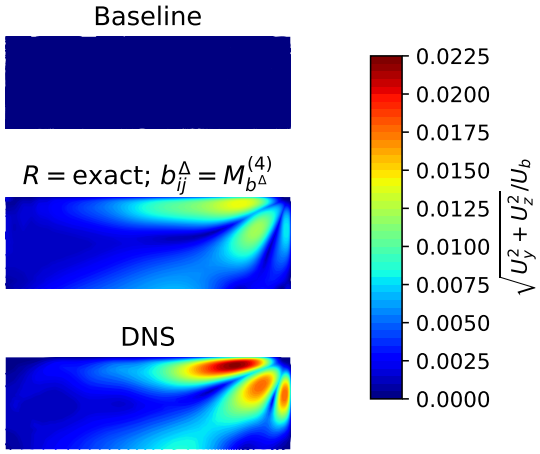
Following the reasoning in Sec. 12.2.2, the rd14L model in Tab. 35 is considered most suitable as a general model and is further tested, it is formalized as $M_{b^{\Delta}}^{(4)}$:

$$\begin{aligned} M_{b^{\Delta}}^{(4)} = 0.457\left(\tanh\left(-10.3q_{\nu}+0.756\right)+1.38\right)T_{ij}^{(2)} + \ldots \\ + 0.567\left(\tanh\left(-20.1q_{\nu}+1.38\right)+1.63\right)T_{ij}^{(3)}. \end{aligned} \tag{107}$$

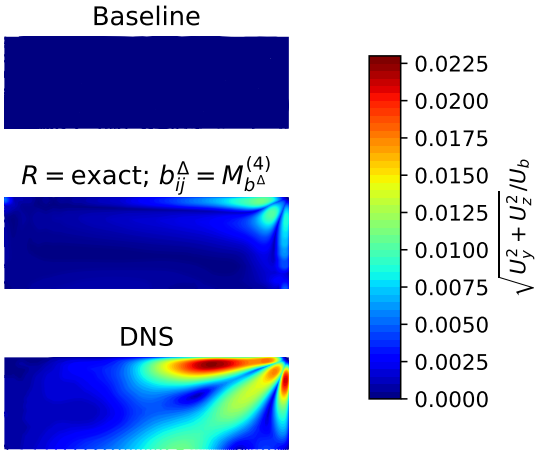
This model is rather difficult to converge (this is also found for other b_{ij}^{Δ} models), so all relaxation factors are set to 0.5. With this simple change, all duct cases converge, though runtime is significantly increased compared to propagation. Even after applying significant under relaxation and using first-order gradient schemes, the hump and roughness cases do not converge. This likely comes from the fact that this model does not fit these cases at all and does not asymptote to zero (which converged the roughness cases in Sec. 12.2.2). Contours of in-plane velocity are shown for the rd1L, rd3L and rd3H case in Fig. 101, together with baseline and DNS.



(a) rd1L



(b) rd3L



(c) rd3H

Figure 101: Contours of in-plane velocity for various duct cases for $b_{ij}^{\Delta} = M_{b^{\Delta}}^{(4)}$ (Eq. 107) with exact R, with baseline (k-ω SST) and DNS added for comparison.

Consider the contours of the rd1L case in Fig. 101a; the shape of the secondary motions is captured extremely well. The main difference with DNS is that the two vortex cores are predicted slightly further from the top right corner, causing a relatively high velocity peak at the symmetry. Though the shape is captured well, the magnitude of the secondary motions is underpredicted. A similar trend is observed for the rd3L case in Fig. 101b; slightly too far out corner vortices and a notable underprediction of in-plane velocity magnitude. Though not shown here for brevity, this trend is also observed for the larger aspect ratio cases. For the rd3H case in Fig. 101c, in-plane velocity magnitude is underpredicted even further. However, now the vortices are actually predicted too close to the top right corner, causing a significant discrepancy also in contour shape. This significant decrease of predictive ability with only a doubling of Reynolds number suggests that the model is strongly Reynolds number dependent.

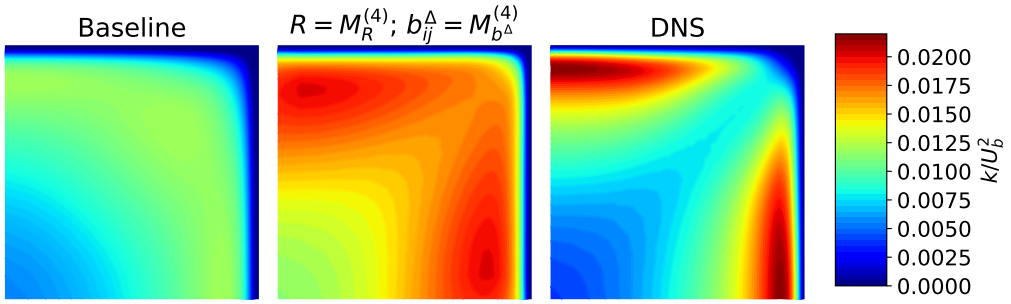
In conclusion, CuRTA has produced a b_{ij}^{Δ} model with satisfactory predictions for the low Reynolds number duct cases. Its main discrepancy is an underprediction of velocity magnitude; this could likely be addressed by an *a-posteriori* coefficient optimization as performed in Sec. 12.1.4. Though the number of coefficients is now too great for a manual optimization. Next, the model is strongly Reynolds number dependent, yielding unsatisfactory predictions for the high Reynolds number ducts, the roughness cases and the hump. This Reynolds number dependence likely comes from training at a single Reynolds number. Training on a set of cases at a range of Reynolds numbers should largely remove Reynolds number dependence. Finally, coupling this model with a model for R rather than exact fields is speculated to significantly impact results, this is further explored in the next section.

12.3 Combined model testing

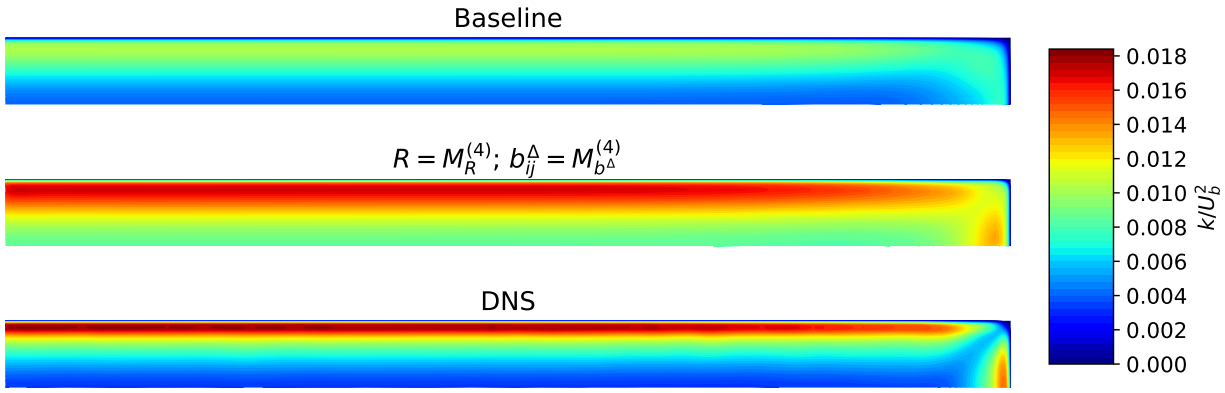
In Sec. 12.1, various R models are trained and tested in isolation by using the exact b_{ij}^{Δ} field. The best model found is $M_R^{(4)}$ (Eq. 102), which performs well for the duct cases and the hump, but performs poorly for the roughness cases. In Sec. 12.2, various b_{ij}^{Δ} models are trained and tested in isolation by using the exact R field. The best model found is $M_{b^{\Delta}}^{(4)}$ (Eq. 107), though it only works well for the low aspect ratio duct cases. In this section, these two models are combined into a full model, which can be tested without requiring exact fields. Thus, the hump can now be tested on its full RANS domain and the channel and plate can finally be tested as well.

As laid out in Sec. 12.2.3, significant convergence difficulties are encountered when testing $M_{b^{\Delta}}^{(4)}$ in isolation. Surprisingly, convergence is improved when combining it with $M_R^{(4)}$ rather than the exact R field, this is observed for all cases. For the duct, only the rd10L and rd14L cases now need stronger underrelaxation than propagation (each variable at 0.5). Contours of k of the rd1L and rd14L duct cases are shown in Fig. 102. Furthermore, in-plane velocity contours are shown for the rd1L, rd3L and rd3H duct cases in Fig. 103. Next, the roughness only converges with strong underrelaxation for U (0.02) and p (0.2), significantly increasing runtime compared to propagation. For roughness case hr00, contours of k and dispersive x-velocity with the in-plane velocity vector field overlaid are shown in Fig. 104.

The hump is run on both the RANS and the LES domain, both domains prove extremely difficult to converge, stemming from small fluctuations in the recirculation bubble. The SIMPLEC algorithm is used with no underrelaxation for p. At the start of the run, little underrelaxation is used for U (0.8), but it is gradually increased during the run (up to 0.0001). While many iterations are required for convergence, all variables eventually reach their outer residual specified in Tab. 27. Profiles of k and U_x are shown in the wake of the hump in Fig. 105, both on the RANS domain (RD) and the LES domain (LD), the run with zero b_{ij}^{Δ} is shown as well. Finally, the channel converges with its baseline settings, while the plate needs additional underrelaxation (each variable at 0.5). The boundary layer velocity profiles of the channel and plate are shown in Fig. 106. The performance of the full model is analyzed next based on these case figures.



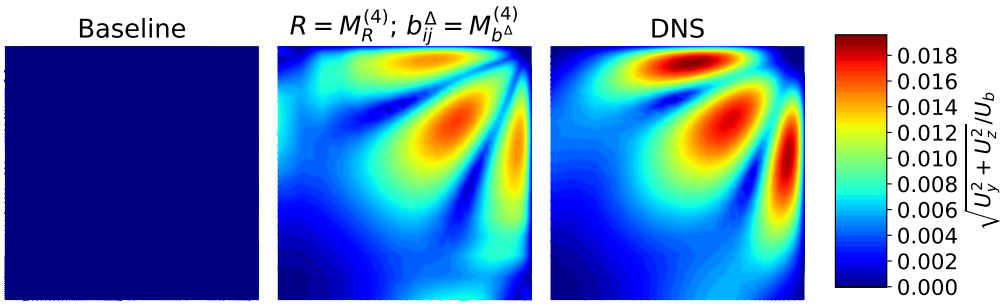
(a) rd1L



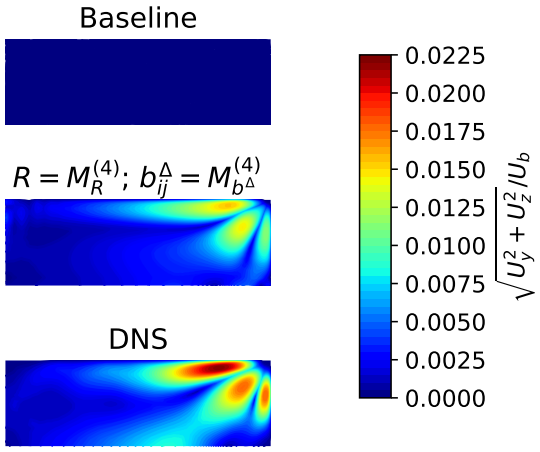
(b) rd14L

Figure 102: Contours of k for various duct cases for $R = M_R^{(4)}$ (Eq. 102) and $b_{ij}^{\Delta} = M_{b^{\Delta}}^{(4)}$ (Eq. 107), with baseline (k-ω SST) and DNS added for comparison.

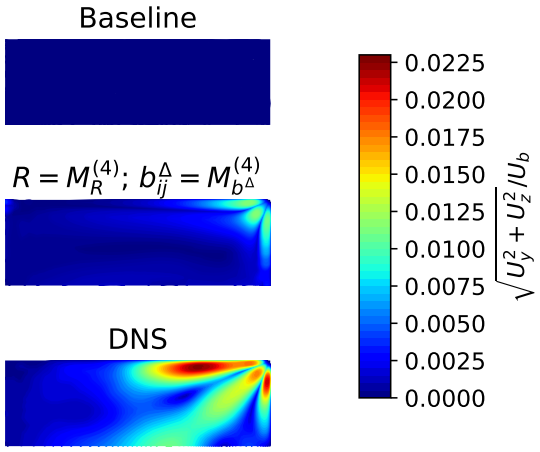
First compare the k contours of the rd1L case between the full model in Fig. 102a and the isolated $M_R^{(4)}$ model in Fig. 93. The full model predicts a larger k everywhere, but especially on the diagonal. This means that the full model improves predictions of k in the lower right and upper left quadrant, but worsens predictions over the diagonal. A similar trend is observed for the k contours of the rd14L case in Fig. 102b, though k is overpredicted slightly less in the corner. This trend is also observed for the other duct cases, where overprediction of k in the corner decreases with aspect ratio. Even the two high Reynolds number duct cases follow this trend (figures omitted for brevity).



(a) rd1L



(b) rd3L



(c) rd3H

Figure 103: Contours of in-plane velocity for various duct cases for $R = M_R^{(4)}$ (Eq. 102) and $b_{ij}^{\Delta} = M_{b^{\Delta}}^{(4)}$ (Eq. 107), with baseline (k-ω SST) and DNS added for comparison.

Now compare the in-plane velocity contours of the rd1L case between the full model in Fig. 103a and the isolated $M_{b^{\Delta}}^{(4)}$ model in Fig. 101a. The full model predicts corner vortices closer to the corner and with a greater strength. This actually makes the full model prediction much closer to DNS than the isolated $M_{b^{\Delta}}^{(4)}$ model, the remaining discrepancy is mostly in terms of magnitude. The same trend is observed when comparing the full model predictions for the rd3L case in Fig. 103b with the predictions of the isolated $M_{b^{\Delta}}^{(4)}$ model in Fig. 101b. However, in-plane velocity magnitude is underpredicted slightly more for this case (though still better than the isolated $M_{b^{\Delta}}^{(4)}$ model). This trend is also observed for the other low Reynolds number duct cases, where the underprediction increases with aspect ratio. As explained in Sec. 12.1.3, k in the corner is largely responsible for the magnitude of the secondary motions. Hence, the decrease of corner k with aspect ratio is thought to be responsible for this increase in underprediction of in-plane velocity magnitude.

Next, compare the in-plane velocity contours of the rd3H case between the full model in Fig. 103c with those of the isolated $M_{b^{\Delta}}^{(4)}$ model in Fig. 101c. Now the full model does not improve the shape of the contours, however, it does predict a higher magnitude. Still, the magnitude is significantly underpredicted, much more so than for the rd3L case. Furthermore, the corner vortices are actually predicted closer to the corner than the rd3L case, while they should be further from the corner. This confirms the Reynolds number dependence of the $M_{b^{\Delta}}^{(4)}$ model once more, though it is still an improvement over baseline for the high Reynolds number duct cases.

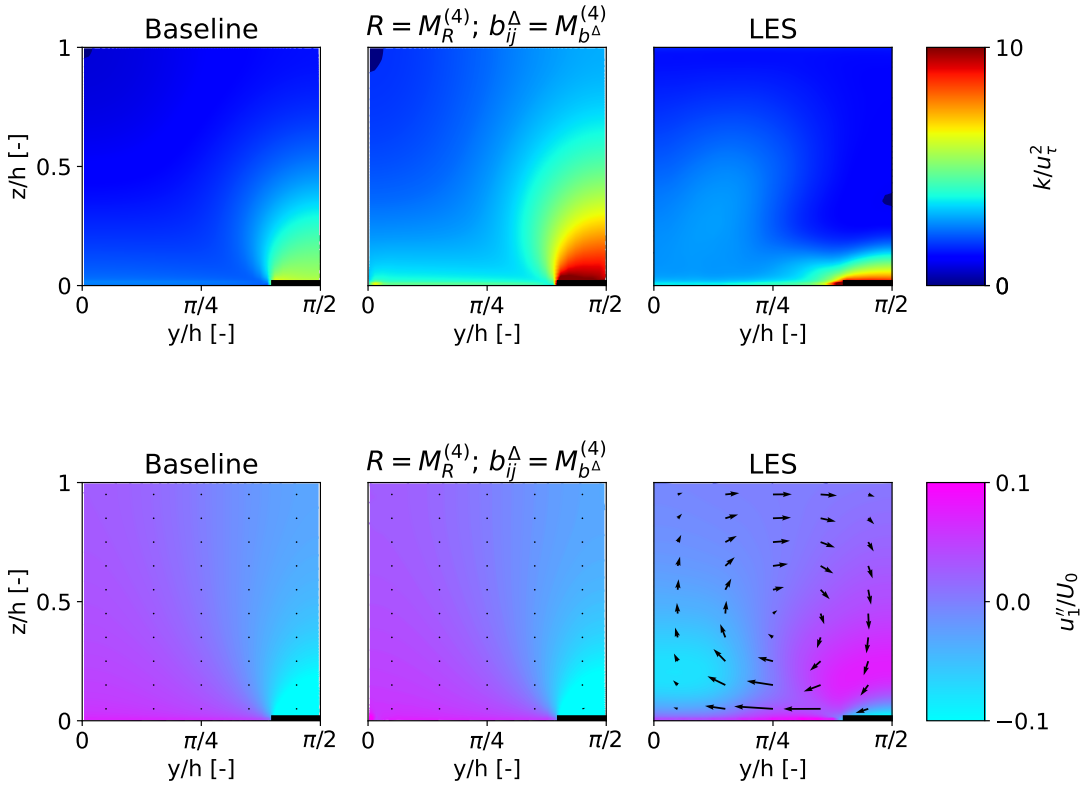


Figure 104: Contours of k and dispersive x-velocity (see Eq. 74) with the in-plane velocity vector field overlaid for the for the hr00 case for $R = M_R^{(4)}$ (Eq. 102) and $b_{ij}^{\Delta} = M_{b^{\Delta}}^{(4)}$ (Eq. 107), with baseline (k-ω SST) and LES added for comparison.

Consider the k contours of the hr00 case in Fig. 104 and compare these with the isolated $M_R^{(4)}$ model for the same case in Fig. 96. While k is still overpredicted for the full model, the shape of the contour is predicted much better. A closer inspection reveals that R is now much lower above the smooth wall, making R above the whole wall much closer to its exact frozen value. This is despite using the exact same R model, indicating that R and b_{ij}^{Δ} are strongly coupled. Next consider the velocity profiles of the full model in Fig. 104, which appear identical to baseline. As mentioned, the roughness has much higher q_ν values than the duct, meaning the b_{ij}^{Δ} model reduces to the following:

$$\lim_{q_\nu \to \infty} M_{b^{\Delta}}^{(4)} = 0.174 T_{ij}^{(2)} + 0.357 T_{ij}^{(3)}. \tag{108}$$

Thus, contrary to baseline, b_{ij}^{Δ} is nonzero and its magnitude turns out to be similar to the magnitude of the exact frozen b_{ij}^{Δ}. The secondary motions are also not exactly zero as in baseline, just much smaller than LES. Upon further inspection, the b_{ij}^{Δ} field appears almost constant, indicating small gradients. As explained in Sec. 7.2, secondary motions are generated by gradients of the Reynolds stress tensor, explaining the almost zero magnitude of the secondary motions.

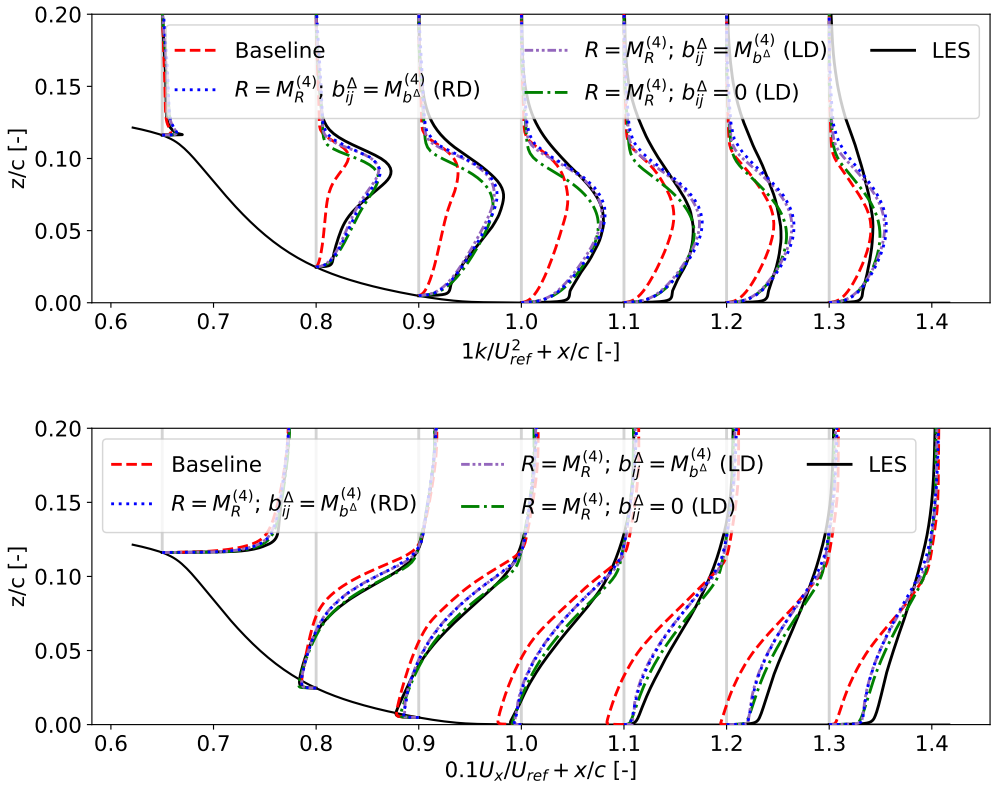
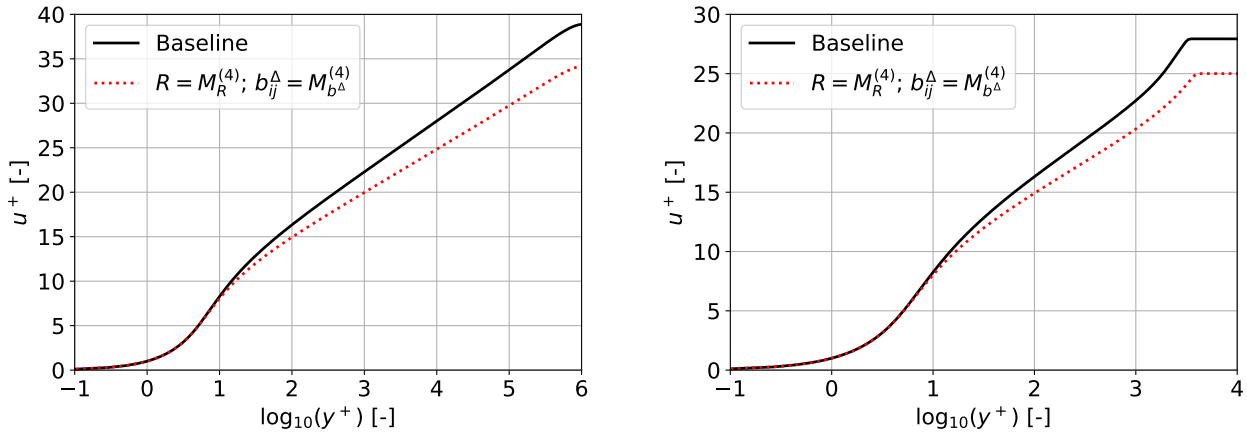


Figure 105: Profiles of k and x-velocity for the hump case for $R = M_R^{(4)}$ (Eq. 102) and $b_{ij}^\Delta = M_{b^\Delta}^{(4)}$ (Eq. 107), run on both the RANS domain (RD) and the LES domain (LD). Baseline (k-ω SST), LES and $R = M_R^{(4)}$ with zero b_{ij}^Δ run on the LES domain are added for comparison.

Compare the k profiles of the hump in Fig. 105 between the LES domain cases with $b_{ij}^\Delta = M_{b^\Delta}^{(4)}$ and $b_{ij}^\Delta = 0$. The $M_{b^\Delta}^{(4)}$ model predicts k slightly better for $x/c \leq 1$ and slightly worse for $x/c > 1$. For the U_x profiles, the $M_{b^\Delta}^{(4)}$ model gives slightly worse predictions everywhere. To assess the influence of using the RANS domain rather than the LES domain, the $M_{b^\Delta}^{(4)}$ model is also run on the RANS domain. No notable difference is found between the results of these domains, confirming again the suitability of the smaller LES domain for training/testing.

The fact that model predictions deteriorate when using $b_{ij}^\Delta = M_{b^\Delta}^{(4)}$ rather than $b_{ij}^\Delta = 0$ indicates that $M_{b^\Delta}^{(4)}$ negatively impacts the hump. As the Reynolds number of the hump is significantly higher than that of the duct, most relevant cells are likely in the asymptotic region of $M_{b^\Delta}^{(4)}$, reducing it to Eq. 108. In Sec. 12.1, a smaller R gave a greater underprediction of U_x at $x/c = 1.3$. Thus, it seems that $M_{b^\Delta}^{(4)}$ erroneously gives a decrease of P_k; the exact b_{ij}^Δ increases P_k as it produces a larger U_x at $x/c = 1.3$ compared to $b_{ij}^\Delta = 0$. Nonetheless, the $M_{b^\Delta}^{(4)}$ model is only slightly worse than $b_{ij}^\Delta = 0$ and it predicts both k and U_x significantly better than baseline.



(a) Channel

(b) Flat plate

Figure 106: Boundary layer velocity profiles for the channel and flat plate case for $R = M_R^{(4)}$ (Eq. 102) and $b_{ij}^{\Delta} = M_{b^{\Delta}}^{(4)}$ (Eq. 107), with baseline (k-ω SST) added for comparison.

Consider the boundary layer velocity profiles of the flat plate and channel case in Fig. 106. The viscous sublayer ($y^+ < 5$) is predicted well for both cases, likely because the model has little influence here (small k and small q_ν). Then, the model starts diverging from baseline in the buffer layer ($5 < y^+ < 30$), presumably due to the model's influence increasing. This discrepancy further grows in the log-layer ($y^+ > 30$), where the wrong slope is predicted. Surprisingly, the model predicts roughly the same erroneous slope for both the channel and the flat plate.

The misprediction of fundamental boundary layer profiles by the full model should not immediately disqualify the R model as universal for several reasons. Firstly, the R and b_{ij}^{Δ} model are heavily intertwined, as observed for the roughness case. The P_k term is modified by b_{ij}^{Δ} as $2k \cdot b_{ij}^{\Delta}\left(\partial u_i / \partial x_j\right)$, meaning the total P_k would be unaffected for the channel and flat plate with a b_{ij}^{Δ} model exactly countering R. Secondly, the baseline k-ω SST model has a number of model coefficients specifically tuned to reproduce the correct boundary layer behaviour [31] [64]. It is not obvious to the author why these coefficients would remain the same with the introduction of the corrections. This is further supported by the fact that other than the slope, the same boundary layer shape is predicted by the model.

All in all, the full model presented in this section greatly improves results of the low-aspect ratio ducts and the hump over baseline. Only a small improvement is found for the high aspect ratio ducts and similar results to baseline are found for the roughness. A deterioration of the channel and flat plate is found, though a coefficient recalibration could potentially resolve this. Thus, in its current form, the model is not yet general, especially as it appears Reynolds number dependent. A further observation is the coupling between R and b_{ij}^{Δ} models, indicating that isolated model testing has its limitations. Improvements are expected when this coupling is taken into consideration in training, for example by accounting for the fit deficit of the b_{ij}^{Δ} model when training the R model.

13 Conclusions

Each research sub-question introduced in the introduction is repeated here, followed by the conclusions pertaining to this sub-question. Finally, the main research question is repeated followed by its conclusions.

Sub-question 1: Is the SpaRTA framework [46] suitable for the symbolic regression of general models?
The SpaRTA framework produces well-fitting, simple R (P_k correction) models for all cases with validated correction fields (duct and hump). A promising single term model is found that generalizes over the duct and hump, but further testcases are required to verify its generality. In contrast, the simple b_{ij}^{Δ} (RST correction) models regressed by SpaRTA provide no better fit than a linear combination of tensor bases. This is the result of the limited functional form of SpaRTA, stemming from its linear regression. The new CuRTA framework, able to represent a much broader range of functional forms due to its non-linear regression, gives a significantly better b_{ij}^{Δ} fit for the same number of terms. Next, inclusion of the model breaks the match with the law of the wall. The baseline model only matches this due to a calibration procedure which appears to be affected by the inclusion of corrections.

Sub-question 2: Does a classifier which applies corrections only in certain areas improve generalizability of models?
A twelve-term classifier is trained on three out of five cases for the NASA challenge entry, based on domain dependent activation data. This classifier improves generalizability, as it preserves the already accurate baseline results for the channel and flat plate case (included in its training). It is even useful for the axisymmetric jet and stalled airfoil case (both not included in its training), as it prevents deterioration by an unsuited model. However, given the large number of terms, it is probable that this classifier is overfitting the data, making generalization unlikely. Further training attempts are halted at the generation of domain independent training data. Thus, the answer to the second sub-question remains inconclusive.

Sub-question 3: How well does the *a-priori* fit to the corrections translate to *a-posteriori* performance of the model?
For both R and b_{ij}^{Δ}, increased *a-posteriori* performance is observed with a better *a-priori* fit (obtained by using a more complex model form). However, the optimal *a-priori* fit to the corrections does not necessarily yield the optimal *a-posteriori* fit to the exact flowfields. For the duct and hump, the *a-posteriori* match is significantly improved after manually optimizing the coefficient of a model produced by SpaRTA. Another important finding is the coupling of R and b_{ij}^{Δ} models, suggesting isolated fitting is not the optimal approach for the best *a-posteriori* performance.

Main question: How suitable is the k-corrective-frozen approach combined with the SpaRTA framework [46] for training a model giving improvements over a range of steady-state flows with respect to k-ω SST?
SpaRTA gives an R model form which generalizes over two flows and possibly more. However, a sub-optimal model coefficient is found, originating from the frozen nature of k-corrective-frozen. For b_{ij}^{Δ}, SpaRTA's linear fit prevents it from properly regressing the data. The CuRTA framework is able to regress a simple b_{ij}^{Δ} model that generalizes over ducts of different aspect ratio. Still, in-plane velocity magnitude is underpredicted, again indicating sub-optimal model coefficients. Furthermore, the model is strongly Reynolds number dependent. In conclusion, neither k-corrective-frozen nor SpaRTA is optimally suited for the discovery of general correction models.

14 Recommendations

Based on the analysis presented in this study, the following recommendations for further research are put forth:

- Following the success of expanding the functional forms available to the symbolic regression, it is recommended to add further functional forms to CuRTA.
- The q_ν feature gives excellent fits, but is highly Reynolds number dependent. The number of input features should be further expanded, with a special focus on using ν_t. Also, new features should be Galilean invariant and ideally not depend on gradients of k/p.
- Many of the optimizations in SpaRTA, such as filtering out features and bases with small variance and cliqueing, are not yet in CuRTA. Adding these should decrease computational cost, requiring less filtering for larger cases.
- For the heterogeneous roughness case, the found correction fields are non-smooth. Due to the rough wall, further mesh refinement is limited. For this case to be of better use, the effect of wall models on k-corrective-frozen should be further studied.
- Much lower outer residuals are needed to converge the frozen cases than expected. Rigorous convergence verification as presented in this work should be in place for future frozen cases.
- There is no constraint in place to ensure fundamental theoretical results such as the law of the wall are matched by a model. As the match of the baseline model with these is simply the result of a calibration procedure, it is recommended to recalibrate coefficients for a given model.
- A highly promising R model is identified which works on all cases with validated correction fields (duct and hump). To confirm its generality, more cases should be added with wall-resolved high fidelity data.
- The best b_{ij}^{Δ} model found is strongly Reynolds number dependent, likely because it is trained at a single Reynolds number. To remove Reynolds number dependence, training should occur over a range of Reynolds numbers simultaneously.
- Designing a proper classifier training criterion based only on local flow quantities proves extremely difficult. A more elaborate study into a proper criterion is warranted, given the good performance of the classifier for the NASA challenge cases.
- A strong coupling is found between the R and b_{ij}^{Δ} model, but they are trained independently. By taking this coupling into account in training, predictions could be improved, for instance by letting R account for the fit deficit of b_{ij}^{Δ}.
- A manual *a-posteriori* optimization of the coefficient in the R model regressed by SpaRTA greatly increases its accuracy. Thus, including the CFD solver in the model regression loop has potential to yield better models. The greatest improvement is expected when it is included in the symbolic regression loop. However, if this proves too expensive, it could also just be used for a coefficient refit of a model regressed on the frozen correction fields.

References

[1] Abramowitz, M., Stegun, I. A., & Romer, R. H. (1988). *Handbook of mathematical functions with formulas, graphs, and mathematical tables.* American Association of Physics Teachers.

[2] Amarloo, A., Forooghi, P., & Abkar, M. (2022a). Frozen propagation of Reynolds force vector from high-fidelity data into Reynolds-averaged simulations of secondary flows. *Physics of Fluids*, *34*(11), 115102. doi: 10.1063/5.0123231

[3] Amarloo, A., Forooghi, P., & Abkar, M. (2022b). Secondary flows in statistically unstable turbulent boundary layers with spanwise heterogeneous roughness. *Theoretical and Applied Mechanics Letters*, *12*(2), 100317. doi: 10.1016/j.taml.2021.100317

[4] Anderson, J. (2017). *Fundamentals of aerodynamics* (6. ed.). McGraw hill.

[5] Anderson, J. D. (1995). *Computational fluid dynamics* (3rd ed.). Springer.

[6] Bridges, J., & Wernet, M. P. (2011). *The nasa subsonic jet particle image velocimetry (PIV) dataset* (Tech. Rep. No. E-17439). Cleveland, Ohio: National Aeronautics and Space Administration (NASA).

[7] Caretto, L. S., Gosman, A. D., Patankar, S. V., & Spalding, D. B. (2007). Two calculation procedures for steady, three-dimensional flows with recirculation. In *Proceedings of the third international conference on numerical methods in fluid mechanics* (pp. 60–68). Springer Berlin Heidelberg. doi: 10.1007/bfb0112677

[8] Cheung, S. H., Oliver, T. A., Prudencio, E. E., Prudhomme, S., & Moser, R. D. (2011). Bayesian uncertainty analysis with applications to turbulence modeling. *Reliability Engineering & System Safety*, *96*(9), 1137–1149. doi: 10.1016/j.ress.2010.09.013

[9] Craft, T., Launder, B., & Suga, K. (1996). Development and application of a cubic eddy-viscosity model of turbulence. *International Journal of Heat and Fluid Flow*, *17*(2), 108–115. doi: 10.1016/0142-727x(95)00079-6

[10] Date, J., & Turnock, S. (1999). *A study into the techniques needed to accurately predict skin friction using RANS solvers with validation against Froude's historical flat plate experimental data* (Tech. Rep. No. 114). Southampton: University of Southampton.

[11] Doormaal, J. P. V., & Raithby, G. D. (1984). Enhancements of the SIMPLE method for predicting incompressible fluid flows. *Numerical Heat Transfer*, *7*(2), 147–163. doi: 10.1080/01495728408961817

[12] Duraisamy, K., Iaccarino, G., & Xiao, H. (2019). Turbulence modeling in the age of data. *Annual Review of Fluid Mechanics*, *51*(1), 357–377. doi: 10.1146/annurev-fluid-010518-040547

[13] Dwight, R., Hoefnagel, K., Tian, R., & Buchanan, T. (2022). *Challenge entry: SpaRTA with classification.* https://turbmodels.larc.nasa.gov/Turb-prs2022/Slide_presentations/Day_2/07_Dwight_challenge.pdf. (Accessed on 05-19-2023)

[14] Edeling, W., Cinnella, P., & Dwight, R. (2014). Predictive RANS simulations via Bayesian model-scenario averaging. *Journal of Computational Physics*, *275*, 65–91. doi: 10.1016/j.jcp.2014.06.052

[15] Gatski, T., & Jongen, T. (2000). Nonlinear eddy viscosity and algebraic stress models for solving complex turbulent flows. *Progress in Aerospace Sciences*, *36*(8), 655–682. doi: 10.1016/s0376-0421(00)00012-9

[16] Gatski, T. B., & Speziale, C. G. (1993). On explicit algebraic stress models for complex turbulent flows. *Journal of Fluid Mechanics*, *254*, 59–78. doi: 10.1017/s0022112093002034

[17] Goderie, M. (2020). *Enhancement of data-driven turbulence models for wind turbine wake applications* (Master's thesis). TU Delft.

[18] Greenblatt, D., Paschal, K., Yao, C., Harris, J., Schaeffler, N., & Washburn, A. (2004). A separation control CFD validation test case. part 1: Baseline & steady suction. In *2nd AIAA flow control conference*. American Institute of Aeronautics and Astronautics. doi: 10.2514/6.2004-2220

[19] Gregory, N., & O'Reilly, C. (1973). *Low-speed aerodynamic characteristics of NACA0012 aerofoil section, including the effects of upper-surface roughness simulating hoar frost* (Tech. Rep. No. R&M 3726). Teddington: National Aeronautics and Space Administration (NASA).

[20] Hemmes, J. (2022). *Data-driven turbulence modelling of algebraic Reynolds-stress models using deep symbolic regression* (Master's thesis). TU Delft.

[21] Huijing, J. P., Dwight, R. P., & Schmelzer, M. (2021, July). Data-driven RANS closures for three-dimensional flows around bluff bodies. *Computers & Fluids*, *225*, 104997. doi: 10.1016/j.compfluid.2021.104997

[22] Jakirlic, S. Z., & Maduta, R. (2014). On "steady" RANS modeling for improved prediction of wall-bounded separation. In *52nd aerospace sciences meeting*. American Institute of Aeronautics and Astronautics. doi: 10.2514/6.2014-0586

[23] Kaandorp, M. L., & Dwight, R. P. (2020). Data-driven modelling of the Reynolds stress tensor using random forests with invariance. *Computers & Fluids*, *202*. doi: 10.1016/j.compfluid.2020.104497

[24] Kolmogorov, A. N. (1941). The local structure of turbulence in incompressible viscous fluid for very large Reynolds numbers. *Doklady Akademii Nauk SSSR*, *30*, 299-303.

[25] Ladson, C. L. (1988). *Effects of independent variation of Mach and Reynolds numbers on the low-speed aerodynamic characteristics of the NACA0012 airfoil section* (Tech. Rep. No. 4074). Hampton, Virginia: National Aeronautics and Space Administration (NASA).

[26] Launder, B., & Sharma, B. (1974). Application of the energy-dissipation model of turbulence to the calculation of flow near a spinning disc. *Letters in Heat and Mass Transfer*, *1*(2), 131–137. doi: 10.1016/0094-4548(74)90150-7

[27] Launder, B. E., Reece, G. J., & Rodi, W. (1975). Progress in the development of a Reynolds-stress turbulence closure. *Journal of Fluid Mechanics*, *68*(3), 537–566. doi: 10.1017/s0022112075001814

[28] Ling, J., Kurzawski, A., & Templeton, J. (2016). Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. *Journal of Fluid Mechanics*, *807*, 155–166. doi: 10.1017/jfm.2016.615

[29] Ling, J., & Templeton, J. (2015). Evaluation of machine learning algorithms for prediction of regions of high Reynolds averaged Navier Stokes uncertainty. *Physics of Fluids*, *27*(8), 085103. doi: 10.1063/1.4927765

[30] Maulik, R., Fytanidis, D. K., Lusch, B., Vishwanath, V., & Patel, S. (2022, July). PythonFOAM: In-situ data analyses with OpenFOAM and Python. *Journal of Computational Science*, *62*, 101750. doi: 10.1016/j.jocs.2022.101750

[31] Menter, F. R. (1994). Two-equation eddy-viscosity turbulence models for engineering applications. *AIAA Journal*, *32*(8), 1598–1605. doi: 10.2514/3.12149

[32] Menter, F. R., Kuntz, M., & Langtry, R. (2003). Ten years of industrial experience with the SST turbulence model. *Turbulence, heat and mass transfer*, *4*(1), 625–632.

[33] Menter, F. R., Garbaruk, A. V., & Egorov, Y. (2012). Explicit algebraic Reynolds stress models for anisotropic wall-bounded flows. In Array (Ed.), *Eucass proceedings series - advances in aerospace sciences* (Vol. 3, p. 89-104). doi: 10.1051/eucass/201203089

[34] Parente, A., Gorlé, C., van Beeck, J., & Benocci, C. (2011). Improved k–ϵ model and wall function formulation for the RANS simulation of ABL flows. *Journal of Wind Engineering and Industrial Aerodynamics*, *99*(4), 267–278. doi: 10.1016/j.jweia.2010.12.017

[35] Perkins, H. J. (1970). The formation of streamwise vorticity in turbulent flow. *Journal of Fluid Mechanics*, *44*(04), 721. doi: 10.1017/s0022112070002112

[36] Pope, S. B. (1975). A more general effective-viscosity hypothesis. *Journal of Fluid Mechanics*, *72*(2), 331–340. doi: 10.1017/S0022112075003382

[37] Pope, S. B. (2000). *Turbulent flows*. Cambridge University Press. doi: 10.1017/cbo9780511840531

[38] Prandtl, L. (1925). Bericht über untersuchungen zur ausgebildeten turbulenz. *ZAMM - Journal of Applied Mathematics and Mechanics*, *5*(2), 136–139. doi: 10.1002/zamm.19250050212

[39] Prandtl, L. (1945). Über ein neues formelsystem für die ausgebildete turbulenz. *Nachrichten der Akademie der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, 6–19.

[40] Reynolds, O. (1895). On the dynamical theory of incompressible viscous fluids and the determination of the criterion. *Philosophical Transactions of the Royal Society of London A.*, *186*, 123-164.

[41] Rodi, W. (1972). *The prediction of free turbulent boundary layers by use of a two equation model of turbulence* (Ph.D. thesis). University of London.

[42] Rumsey, C. (2022). *2022 symposium on turbulence modeling: Roadblocks, and the potential for machine learning*. https://turbmodels.larc.nasa.gov/turb-prs2022.html. (Accessed on 12-22-2022)

[43] Rumsey, C. (2023). *The Menter shear stress transport turbulence model.* https://turbmodels.larc.nasa.gov/sst.html. (Accessed: 2023-29-03)

[44] Rumsey, C., Gatski, T., Sellers, W., Vatsa, V., & Viken, S. (2004). Summary of the 2004 CFD validation workshop on synthetic jets and turbulent separation control. In *2nd AIAA flow control conference*. American Institute of Aeronautics and Astronautics. doi: 10.2514/6.2004-2217

[45] Rumsey, C. L., & Colem, G. N. (2022). *NASA symposium on turbulence modeling: Roadblocks, and the potential for machine learning* (Tech. Rep.). Hampton, Virginia: National Aeronautics and Space Administration (NASA).

[46] Schmelzer, M., Dwight, R. P., & Cinnella, P. (2019). Discovery of algebraic Reynolds-stress models using sparse symbolic regression. *Flow, Turbulence and Combustion*, *104*(2-3), 579–603. doi: 10.1007/s10494-019-00089-x

[47] Schmitt, F. G. (2007). About Boussinesq's turbulent viscosity hypothesis: historical remarks and a direct evaluation of its validity. *Comptes Rendus Mécanique*, *335*(9-10), 617–627. doi: 10.1016/j.crme.2007.08.004

[48] Settles, G. (2009). *Laminar-turbulent transition.* Retrieved from https://commons.wikimedia.org/wiki/File:Laminar-turbulent_transition.jpg

[49] Smith, A., & Cebeci, T. (1967). *Numerical solution of the turbulent boundary layer equations* (Tech. Rep. No. DAC 33735). Long beach, CA: Douglas aircraft company.

[50] Spalart, P. (2000). Strategies for turbulence modelling and simulations. *International Journal of Heat and Fluid Flow*, *21*(3), 252–263. doi: 10.1016/s0142-727x(00)00007-2

[51] Spalart, P. R. (2015). Philosophies and fallacies in turbulence modeling. *Progress in Aerospace Sciences*, *74*, 1–15. doi: 10.1016/j.paerosci.2014.12.004

[52] Speziale, C. G., Sarkar, S., & Gatski, T. B. (1991). Modelling the pressure–strain correlation of turbulence: an invariant dynamical systems approach. *Journal of Fluid Mechanics*, *227*, 245–272. doi: 10.1017/s0022112091000101

[53] Steiner, J., Dwight, R. P., & Viré, A. (2022). Data-driven RANS closures for wind turbine wakes under neutral conditions. *Computers & Fluids*, *233*. doi: 10.1016/j.compfluid.2021.105213

[54] Steiner, J., Viré, A., & Dwight, R. P. (2022). Classifying regions of high model error within a data-driven RANS closure: Application to wind turbine wakes. *Flow, Turbulence and Combustion*, *109*(3), 545–570. doi: 10.1007/s10494-022-00346-6

[55] Thompson, R. L., Sampaio, L. E. B., de Bragança Alves, F. A., Thais, L., & Mompean, G. (2016). A methodology to evaluate statistical errors in DNS data of plane channel flows. *Computers & Fluids*, *130*, 1–7. doi: 10.1016/j.compfluid.2016.01.014

[56] Uzun, A., & Malik, M. R. (2017). Wall-resolved large-eddy simulation of flow separation over NASA wall-mounted hump. In *55th AIAA aerospace sciences meeting*. American Institute of Aeronautics and Astronautics. doi: 10.2514/6.2017-0538

[57] Van Kan, J., Segal, A., Vermolen, F., & Kraaijevanger, H. (2019). *Numerical methods for partial differential equations*. Delft Academic Press.

[58] Vinuesa, R., Schlatter, P., & Nagib, H. M. (2018). Secondary flow in turbulent ducts with increasing aspect ratio. *Physical Review Fluids*, *3*(5). doi: 10.1103/physrevfluids.3.054606

[59] Wang, J.-X., Wu, J., Ling, J., Iaccarino, G., & Xiao, H. (2017). A comprehensive physics-informed machine learning framework for predictive turbulence modeling. *arXiv:1701.07102*. doi: 10.48550/ARXIV.1701.07102

[60] Wang, J.-X., Wu, J.-L., & Xiao, H. (2017). Physics-informed machine learning approach for reconstructing Reynolds stress modeling discrepancies based on DNS data. *Physical Review Fluids*, *2*(3). doi: 10.1103/physrevfluids.2.034603

[61] Weatheritt, J., & Sandberg, R. (2016). A novel evolutionary algorithm applied to algebraic modifications of the RANS stress–strain relationship. *Journal of Computational Physics*, *325*, 22–37. doi: 10.1016/j.jcp.2016.08.015

[62] Weatheritt, J., & Sandberg, R. (2017). The development of algebraic stress models using a novel evolutionary algorithm. *International Journal of Heat and Fluid Flow*, *68*, 298–318. doi: 10.1016/j.ijheatfluidflow.2017.09.017

[63] White, F. M., & Majdalani, J. (1991). *Viscous fluid flow* (Vol. 2). McGraw-Hill New York.

[64] Wilcox, D. C. (1988). Reassessment of the scale-determining equation for advanced turbulence models. *AIAA Journal*, *26*(11), 1299–1310. doi: 10.2514/3.10041

[65] Wilcox, D. C. (2006). *Turbulence modeling for CFD* (3. ed.). DCW industries La Canada, CA.

[66] Xiao, H., Wu, J.-L., Wang, J.-X., Sun, R., & Roy, C. (2016). Quantifying and reducing model-form uncertainties in Reynolds-averaged Navier-Stokes simulations: A data-driven, physics-informed bayesian approach. *Journal of Computational Physics*, *324*, 115–136. doi: 10.1016/j.jcp.2016.07.038

[67] Zhang, Y., Dwight, R. P., Schmelzer, M., Gómez, J. F., hua Han, Z., & Hickel, S. (2021, May). Customized data-driven RANS closures for bi-fidelity LES–RANS optimization. *Journal of Computational Physics*, *432*, 110153. doi: 10.1016/j.jcp.2021.110153

[68] Zhong, W., Tang, H., Wang, T., & Zhu, C. (2018). Accurate RANS simulation of wind turbine stall by turbulence coefficient calibration. *Applied Sciences*, *8*(9), 1444. doi: 10.3390/app8091444

[69] Zhou, L. (2018). *Theory and modeling of dispersed multiphase turbulent reacting flows*. Butterworth-Heinemann.

[70] Zou, H., & Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, *67*(2), 301–320. doi: 10.1111/j.1467-9868.2005.00503.x

A Rectangular duct case blockMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  7
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

//Include case variables (e.g. h).
#include "../../../casedef"

//Include mesh variables (e.g. beta).
#include "../meshdef"

//Calculate the dimensional first cell height.
//The factor two is there since OpenFOAM calculates yPlus at the cell center.
delta_1     #calc "$yPlus * $h / $Re_tau * 2.";

//Calculate the number of cells in y- and z-direction
N_y         #calc "ceil(log($h*($beta-1)/$delta_1+1)/log($beta))";
N_z         #calc "ceil($N_y*$AR)";

//Calculate the ratio between the first and last cell
ratio       #calc "1/pow($beta, $N_y-1)";

//Scale the mesh by the domain height
scale       $h;

//Nondimensional coordinates of the mesh vertices
vertices
(
    (0 0 0) //0
    (1 0 0) //1
    (1 1 0) //2
    (0 1 0) //3
    (0 0 $AR) //4
    (1 0 $AR) //5
    (1 1 $AR) //6
    (0 1 $AR) //7
);

//Create the single block in the mesh, with the calculated number of cells and ratio
    between first and last cell
blocks
(
    hex (0 1 2 3 4 5 6 7)
    (1 $N_y $N_z)
    simpleGrading (1 $ratio $ratio)
);


//Specify the boundary type for each of the six mesh faces
boundary
(
    inflow
    {
        type                cyclic;
        neighbourPatch      outflow;
        faces
        (
            (0 4 7 3)
```

```
        );
    }

    outflow
    {
        type                cyclic;
        neighbourPatch      inflow;
        faces
        (
            (1 2 6 5)
        );
    }

    wallTop
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }

    wallSide
    {
        type wall;
        faces
        (
            (4 5 6 7)
        );
    }

    symmetryBottom
    {
        type symmetry;
        faces
        (
            (0 1 5 4)
        );
    }

    symmetrySide
    {
        type symmetry;
        faces
        (
            (0 3 2 1)
        );
    }
);
```

B Atmospheric wall functions rewritten to OpenFOAM-7

B.1 atmOmegaWallFunction.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2019 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "atmOmegaWallFunctionFvPatchScalarField.H"
#include "nutWallFunctionFvPatchScalarField.H"
#include "turbulenceModel.H"
#include "fvMatrix.H"
#include "addToRunTimeSelectionTable.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

// * * * * * * * * * * * * * Protected Member Functions  * * * * * * * * * * * //


void atmOmegaWallFunctionFvPatchScalarField::calculate
(
    const turbulenceModel& turbModel,
    const List<scalar>& cornerWeights,
    const fvPatch& patch,
    scalarField& G0,
    scalarField& omega0
)
{
    const label patchi = patch.index();

    const tmp<scalarField> tnutw = turbModel.nut(patchi);

    const nutWallFunctionFvPatchScalarField& nutw = nutWallFunctionFvPatchScalarField::nutw
    (turbModel, patchi);

    const scalarField& y = turbModel.y()[patchi];

    const tmp<scalarField> tnuw = turbModel.nu(patchi);
    const scalarField& nuw = tnuw();

    const tmp<volScalarField> tk = turbModel.k();
    const volScalarField& k = tk();
```

```
    const fvPatchVectorField& Uw = turbModel.U().boundaryField()[patchi];

    const scalarField magGradUw(mag(Uw.snGrad()));

    const scalar Cmu25 = pow025(nutw.Cmu());
    const scalar kappa = nutw.kappa();


    // Set omega and G
    forAll(nutw, facei)
    {
        const label celli = patch.faceCells()[facei];
        const scalar w = cornerWeights[facei];


        omega0[celli] += w*sqrt(k[celli])/(Cmu25*kappa*(y[facei] + z0_[facei]));

        G0[celli] += w*(nutw[facei] + nuw[facei])*magGradUw[facei]*Cmu25*sqrt(k[celli])/(
    kappa*(y[facei] + z0_[facei]));


    }
}


// * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * * //

atmOmegaWallFunctionFvPatchScalarField::atmOmegaWallFunctionFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
    omegaWallFunctionFvPatchScalarField(p, iF),
    z0_(p.size(), 0.0)
{}


atmOmegaWallFunctionFvPatchScalarField::atmOmegaWallFunctionFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    omegaWallFunctionFvPatchScalarField(p, iF, dict),
    z0_("z0", dict, p.size())
{
    // apply zero-gradient condition on start-up
    //this->operator==(patchInternalField());
}


atmOmegaWallFunctionFvPatchScalarField::atmOmegaWallFunctionFvPatchScalarField
(
    const atmOmegaWallFunctionFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    omegaWallFunctionFvPatchScalarField(ptf, p, iF, mapper),
    z0_(ptf.z0_)
{}


atmOmegaWallFunctionFvPatchScalarField::atmOmegaWallFunctionFvPatchScalarField
(
    const atmOmegaWallFunctionFvPatchScalarField& aowfpsf
)
:
    omegaWallFunctionFvPatchScalarField(aowfpsf),
```

```
    z0_(aowfpsf.z0_)
{}


atmOmegaWallFunctionFvPatchScalarField::atmOmegaWallFunctionFvPatchScalarField
(
    const atmOmegaWallFunctionFvPatchScalarField& aowfpsf,
    const DimensionedField<scalar, volMesh>& iF
)
:
    omegaWallFunctionFvPatchScalarField(aowfpsf, iF),
    z0_(aowfpsf.z0_)
{}


// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //

void atmOmegaWallFunctionFvPatchScalarField::autoMap
(
    const fvPatchFieldMapper& m
)
{
    atmOmegaWallFunctionFvPatchScalarField::autoMap(m);
    m(z0_, z0_);
}

void atmOmegaWallFunctionFvPatchScalarField::rmap
(
    const fvPatchScalarField& ptf,
    const labelList& addr
)
{
    atmOmegaWallFunctionFvPatchScalarField::rmap(ptf, addr);

    const atmOmegaWallFunctionFvPatchScalarField& aowfpsf =
        refCast<const atmOmegaWallFunctionFvPatchScalarField>(ptf);

    z0_.rmap(aowfpsf.z0_, addr);
}

void atmOmegaWallFunctionFvPatchScalarField::write(Ostream& os) const
{

    fixedValueFvPatchField<scalar>::write(os);
    writeEntry(os, "z0", z0_);
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

makePatchTypeField
(
    fvPatchScalarField,
    atmOmegaWallFunctionFvPatchScalarField
);

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam

// ************************************************************************* //
```

B.2 atmOmegaWallFunction.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2019 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::atmOmegaWallFunctionFvPatchScalarField

Group
    grpAtmWallFunctions

Description
    From https://www.openfoam.com/documentation/guides/latest/api/
    atmOmegaWallFunctionFvPatchScalarField_8H_source.html
    Adapted to OF7 by Kaj Hoefnagel

    This boundary condition provides a wall constraint on the specific
    dissipation rate (i.e. \c omega) and the turbulent kinetic energy
    production contribution (i.e. \c G) for atmospheric boundary
    layer modelling.

    References:
    \verbatim
        Theoretical expressions (tags:PGVB, B):
            Parente, A., Gorlé, C., Van Beeck, J., & Benocci, C. (2011).
            Improved k-ϵ model and wall function formulation
            for the RANS simulation of ABL flows.
            J. of wind engineering and industrial aerodynamics, 99(4), 267-278.
            DOI:10.1016/j.jweia.2010.12.017

            Bredberg, J. (2000).
            On the wall boundary condition for turbulence models.
            Chalmers University of Technology, Depart. of Thermo and Fluid Dyn.
            Internal Report 00/4. Sweden: Göteborg.
    \endverbatim

    Required fields:
    \verbatim
      omega    | Specific dissipation rate    [1/s]
    \endverbatim

Usage
    Example of the boundary condition specification:
    \verbatim
    <patchName>
    {
        // Mandatory entries
        type            atmOmegaWallFunction;
        z0              <PatchFunction1<scalar>>;

        // Inherited entries
        ...
```

```
    }
    \endverbatim

    where the entries mean:
    \table
      Property | Description                        | Type    | Reqd    | Deflt
      type     | Type name: atmOmegaWallFunction    | word    | yes     | -
      z0       | Surface roughness length [m]       | PatchFunction1<scalar> | yes | -
    \endtable

    The inherited entries are elaborated in:
      - \link omegaWallFunctionFvPatchScalarField.H \endlink
      - \link PatchFunction1.H \endlink

SourceFiles
    atmOmegaWallFunctionFvPatchScalarField.C

\*---------------------------------------------------------------------------*/

#ifndef atmOmegaWallFunctionFvPatchScalarField_H
#define atmOmegaWallFunctionFvPatchScalarField_H

#include "fixedValueFvPatchField.H"
#include "omegaWallFunctionFvPatchScalarField.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

/*---------------------------------------------------------------------------*\
           Class atmOmegaWallFunctionFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/

class atmOmegaWallFunctionFvPatchScalarField
:
    public omegaWallFunctionFvPatchScalarField
{
protected:

    // Protected data
        //- Surface roughness length field [m]
        scalarField z0_;


    // Protected Member Functions


        //- Calculate the omega and G
        virtual void calculate
        (
            const turbulenceModel& turbulence,
            const List<scalar>& cornerWeights,
            const fvPatch& patch,
            scalarField& G,
            scalarField& omega
        );



public:

    //- Runtime type information
    TypeName("atmOmegaWallFunction");


    // Constructors

        //- Construct from patch and internal field
        atmOmegaWallFunctionFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&
```

```
        );

        //- Construct from patch, internal field and dictionary
        atmOmegaWallFunctionFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );

        //- Construct by mapping given
        // atmOmegaWallFunctionFvPatchScalarField
        //  onto a new patch
        atmOmegaWallFunctionFvPatchScalarField
        (
            const atmOmegaWallFunctionFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Copy constructor
        atmOmegaWallFunctionFvPatchScalarField
        (
            const atmOmegaWallFunctionFvPatchScalarField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new atmOmegaWallFunctionFvPatchScalarField(*this)
            );
        }

        //- Copy constructor setting internal field reference
        atmOmegaWallFunctionFvPatchScalarField
        (
            const atmOmegaWallFunctionFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new atmOmegaWallFunctionFvPatchScalarField(*this, iF)
            );
        }


    // Member Functions

        // Mapping functions

            //- Map (and resize as needed) from self given a mapping object
            virtual void autoMap(const fvPatchFieldMapper&);

            //- Reverse map the given fvPatchField onto this fvPatchField
            virtual void rmap
            (
                const fvPatchScalarField&,
                const labelList&
            );

        // I-O

            //- Write
```

```
            virtual void write(Ostream&) const;
};


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#endif

// ************************************************************************* //
```

B.3 atmNutUWallFunction.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2019 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "atmNutUWallFunctionFvPatchScalarField.H"
#include "turbulenceModel.H"
#include "fvPatchFieldMapper.H"
#include "volFields.H"
#include "addToRunTimeSelectionTable.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

// * * * * * * * * * * * * * Protected Member Functions  * * * * * * * * * * * //

tmp<scalarField> atmNutUWallFunctionFvPatchScalarField::nut() const
{
    const label patchi = patch().index();

    const turbulenceModel& turbModel = db().lookupObject<turbulenceModel>
    (
        IOobject::groupName
        (
            turbulenceModel::propertiesName,
            internalField().group()
        )
    );


    const scalarField& y = turbModel.y()[patchi];

    const fvPatchVectorField& Uw = turbModel.U().boundaryField()[patchi];
    const vectorField Up(Uw.patchInternalField() - Uw);
    const scalarField magUpn(mag(Up - (Up & patch().nf())*patch().nf()));

    const tmp<scalarField> tnuw = turbModel.nu(patchi);
    const scalarField& nuw = tnuw();



    tmp<scalarField> tnutw(new scalarField(patch().size(), 0.0));
    scalarField& nutw = tnutw.ref();


    forAll(nutw, facei)
    {
```

```
        const scalar Edash = (y[facei] + z0_[facei])/(z0_[facei] + 1e-4);
        const scalar uStar = magUpn[facei]*kappa_/log(max(Edash, 1.0+1e-4));

        nutw[facei] = sqr(uStar)/max(magUpn[facei], 1e-6)*y[facei] - nuw[facei];

    }

    if (boundNut_)
    {
        nutw = max(nutw, scalar(0));
    }


    return tnutw;
}


tmp<scalarField> atmNutUWallFunctionFvPatchScalarField::yPlus
(
    const scalarField& magUp
) const
{
    const label patchi = patch().index();

    const turbulenceModel& turbModel = db().lookupObject<turbulenceModel>
    (
        IOobject::groupName
        (
            turbulenceModel::propertiesName,
            internalField().group()
        )
    );
    const scalarField& y = turbModel.y()[patchi];
    const tmp<scalarField> tnuw = turbModel.nu(patchi);
    const scalarField& nuw = tnuw();

    tmp<scalarField> tyPlus(new scalarField(patch().size(), 0.0));
    scalarField& yPlus = tyPlus.ref();

    forAll(yPlus, facei)
    {
        const scalar Re = magUp[facei]*y[facei]/nuw[facei];
        const scalar ryPlusLam = 1/yPlusLam_;

        int iter = 0;
        scalar yp = yPlusLam_;
        scalar yPlusLast = yp;

        do
        {
            yPlusLast = yp;
            if (yp > yPlusLam_)
            {
                yp = (kappa_*Re + yp)/(1 + log(E_*yp));
            }
            else
            {
                yp = sqrt(Re);
            }
        } while(mag(ryPlusLam*(yp - yPlusLast)) > 0.0001 && ++iter < 20);

        yPlus[facei] = yp;
    }

    return tyPlus;
}


// * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * //

atmNutUWallFunctionFvPatchScalarField::atmNutUWallFunctionFvPatchScalarField
(
    const fvPatch& p,
```

```
    const DimensionedField<scalar, volMesh>& iF
)
:
    nutUWallFunctionFvPatchScalarField(p, iF),
    boundNut_(true),
    z0_(p.size(), 0.0)
{}


atmNutUWallFunctionFvPatchScalarField::atmNutUWallFunctionFvPatchScalarField
(
    const atmNutUWallFunctionFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    nutUWallFunctionFvPatchScalarField(ptf, p, iF, mapper),
    boundNut_(ptf.boundNut_),
    z0_(ptf.z0_)
{}


atmNutUWallFunctionFvPatchScalarField::atmNutUWallFunctionFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    nutUWallFunctionFvPatchScalarField(p, iF, dict),
    boundNut_(dict.lookupOrDefault<bool>("boundNut", true)),
    z0_("z0", dict, p.size())
{}


atmNutUWallFunctionFvPatchScalarField::atmNutUWallFunctionFvPatchScalarField
(
    const atmNutUWallFunctionFvPatchScalarField& snawfpsf
)
:
    nutUWallFunctionFvPatchScalarField(snawfpsf),
    boundNut_(snawfpsf.boundNut_),
    z0_(snawfpsf.z0_)
{}


atmNutUWallFunctionFvPatchScalarField::atmNutUWallFunctionFvPatchScalarField
(
    const atmNutUWallFunctionFvPatchScalarField& snawfpsf,
    const DimensionedField<scalar, volMesh>& iF
)
:
    nutUWallFunctionFvPatchScalarField(snawfpsf, iF),
    boundNut_(snawfpsf.boundNut_),
    z0_(snawfpsf.z0_)
{}


// * * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * * * //

void atmNutUWallFunctionFvPatchScalarField::autoMap
(
    const fvPatchFieldMapper& m
)
{
    atmNutUWallFunctionFvPatchScalarField::autoMap(m);
    m(z0_, z0_);
}

void atmNutUWallFunctionFvPatchScalarField::rmap
(
    const fvPatchScalarField& ptf,
```

```
    const labelList& addr
)
{
    atmNutUWallFunctionFvPatchScalarField::rmap(ptf, addr);

    const atmNutUWallFunctionFvPatchScalarField& snawfpsf =
        refCast<const atmNutUWallFunctionFvPatchScalarField>(ptf);

    z0_.rmap(snawfpsf.z0_, addr);
}


void atmNutUWallFunctionFvPatchScalarField::write(Ostream& os) const
{
    fvPatchField<scalar>::write(os);
    writeLocalEntries(os);
    writeEntry(os, "value", *this);
    writeEntry(os, "z0", z0_);
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

makePatchTypeField
(
    fvPatchScalarField,
    atmNutUWallFunctionFvPatchScalarField
);

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam

// ************************************************************************* //
```

B.4 atmNutUWallFunction.H

```
/*--------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2019 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::atmNutUWallFunctionFvPatchScalarField

Group
    grpAtmWallFunctions

Description
    From: https://www.openfoam.com/documentation/guides/latest/api/
    atmNutUWallFunctionFvPatchScalarField_8H_source.html
    Adapted to OF7 by Kaj Hoefnagel

    This boundary condition provides a wall constraint on the turbulent
    viscosity (i.e. \c nut) based on velocity (i.e. \c U) for atmospheric
    boundary layer modelling.  It is designed to be used in conjunction
    with the \c atmBoundaryLayerInletVelocity boundary condition.

    The governing equation of the boundary condition:

        \f[
            u = \frac{u^*}{\kappa} ln \left(\frac{z + z_0}{z_0}\right)
        \f]

    where
    \vartable
        u^*      | Friction velocity
        \kappa   | von Kármán constant
        z_0      | Surface roughness length [m]
        z        | Ground-normal coordinate
    \endvartable

    Required fields:
    \verbatim
      nut     | Turbulent viscosity         [m2/s]
      U       | Velocity                    [m/s]
    \endverbatim

Usage
    Example of the boundary condition specification:
    \verbatim
    <patchName>
    {
        // Mandatory entries
        type            atmNutUWallFunction;
        z0              <PatchFunction1<scalar>>;

        // Optional entries
        boundNut        true;
```

```
        // Inherited entries
        ...
    }
    \endverbatim

    where the entries mean:
    \table
      Property | Description                         | Type    | Reqd  | Deflt
      type     | Type name: atmNutUWallFunction      | word    | yes   | -
      z0       | Surface roughness length [m] | PatchFunction1<scalar> | yes | -
      boundNut | Flag: zero-bound nut near wall      | bool    | no    | true
    \endtable

    The inherited entries are elaborated in:
      - \link nutUWallFunctionFvPatchScalarField.H \endlink
      - \link PatchFunction1.H \endlink

SourceFiles
    atmNutUWallFunctionFvPatchScalarField.C

\*---------------------------------------------------------------------------*/

#ifndef atmNutUWallFunctionFvPatchScalarField_H
#define atmNutUWallFunctionFvPatchScalarField_H

#include "nutUWallFunctionFvPatchScalarField.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

/*---------------------------------------------------------------------------*\
           Class atmNutUWallFunctionFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/

class atmNutUWallFunctionFvPatchScalarField
:
    public nutUWallFunctionFvPatchScalarField
{

    //- Flag to zero-bound nut to prevent negative nut
    //- at the wall arising from negative heat fluxes
    const bool boundNut_;

    //- Surface roughness length field [m]
    scalarField z0_;

protected:

    // Protected Member Functions

        //- Calculate yPLus
        virtual tmp<scalarField> yPlus(const scalarField& magUp) const;

        //- Calculate the turbulence viscosity
        virtual tmp<scalarField> nut() const;


public:

    //- Runtime type information
    TypeName("atmNutUWallFunction");


    // Constructors

        //- Construct from patch and internal field
        atmNutUWallFunctionFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&
```

```
        );

        //- Construct from patch, internal field and dictionary
        atmNutUWallFunctionFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );

        //- Construct by mapping given
        //  atmNutUWallFunctionFvPatchScalarField
        //  onto a new patch
        atmNutUWallFunctionFvPatchScalarField
        (
            const atmNutUWallFunctionFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Copy constructor
        atmNutUWallFunctionFvPatchScalarField
        (
            const atmNutUWallFunctionFvPatchScalarField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new atmNutUWallFunctionFvPatchScalarField(*this)
            );
        }

        //- Copy constructor setting internal field reference
        atmNutUWallFunctionFvPatchScalarField
        (
            const atmNutUWallFunctionFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new atmNutUWallFunctionFvPatchScalarField(*this, iF)
            );
        }


    // Member Functions

        // Mapping functions

            //- Map (and resize as needed) from self given a mapping object
            virtual void autoMap(const fvPatchFieldMapper&);

            //- Reverse map the given fvPatchField onto this fvPatchField
            virtual void rmap
            (
                const fvPatchScalarField&,
                const labelList&
            );


        // I-O
```

```
            //- Write
            virtual void write(Ostream& os) const;
};


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#endif

// ************************************************************************* //
```

C Python function to generate plot3d files

```
def convertToPlot3d(X, Z, c, outFile):
    '''Function to convert grid coordinates to plot3d format (which is easy to
    convert to OpenFOAM). Assumes the grid is in the xz-plane and copies this
    grid at y=0 and y=-c to give it a single layer of depth.

    Inputs:
    X : 2D array of x-coordinates
    Z : 2D array of z-coordinates
    c : chord length
    outFile : (path) to the output plot3d file in which the mesh points are to
              be saved.'''

    #If the outFile does not yet have the plot3d extension, add it
    if outFile[-7:] != '.p3dfmt':
        outFile += '.p3dfmt'

    #Flatten the arrays; duplicate each value of x- and z and make the
    #y-array alternating between 0 and 1.
    XFlat = np.concatenate([X.reshape(1,-1)]*2, axis=0).flatten(order='f')
    YFlat = np.hstack((np.ones((X.size,1))*-c, np.zeros((X.size,1)))).flatten()
    ZFlat = np.concatenate([Z.reshape(1,-1)]*2, axis=0).flatten(order='f')

    #Header of the plot3d file: first line indicates the number of blocks (1)
    #second line indicates the number of points in y, x and z.
    header = f'1\n2 {X.shape[1]} {X.shape[0]}'

    #Save the flattened arrays (plot3d first lists all x-values, then y-values
    #then z-values).
    np.savetxt(outFile, np.concatenate([XFlat, YFlat, ZFlat]),
               header=header, comments='')
```

D Model propagation infrastructure

D.1 Custom turbulence model for model propagation (modelPropagationkOmegaSST)

D.1.1 modelPropagationkOmegaSST.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2016-2018 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::RASModels::kOmegaSST

Description
    Specialisation for RAS of the generic kOmegaSSTBase base class.
    For more information, see Description of kOmegaSSTBase.H

See also
    Foam::kOmegaSST

SourceFiles
    kOmegaSST.C

\*---------------------------------------------------------------------------*/


/*---------------------------------------------------------------------------*\
Modification of propagationkOmegaSST; now instead of reading kDeficit and
bijDelta, they are computed based on some model. The model calculation goes
via Python, such that no recompilation is necessary everytime a new model is
to be tested. A case is to be run with the turbulence model set to
modelPropagationkOmegaSST and using modelPropagationFoam as the solver. Then,
it will look for model_propagation.py in the folder. Within this file, a
function model should be defined that takes an array of shape
(nCells, num_scalars), with columns corresponding to: gradU, k, omega, gradp,
gradk, nut, U, walldist, nu and curlU. The function should return an array of
shape (nCells, 8) with the first column corresponding to kDeficit, the next
six to the components of bijDelta and the last one to the classifier sigma.
A template for model_definition.py is available in:
inversion/TurbFOAM-7/src/TurbulenceModels/turbulenceModels/
                                          RAS/modelPropagationkOmegaSST
If a classifier is not used, the boolean useSigma should be set to false in
the constant/turbulenceProperties file of the case. Furthermore, for isolated
testing of correction/classifier models, the boolean modelkDeficit/modelRST/
modelSigma should be set to false to use exact fields for this variable
(defined in the 0 directory). T
\*---------------------------------------------------------------------------*/
```

```
#ifndef modelPropagationkOmegaSST_H
#define modelPropagationkOmegaSST_H

#include "kOmegaSSTBase.H"
#include "RASModel.H"
#include "eddyViscosity.H"

/*The following is for Python interoperability*/
#include <Python.h>
#define NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION
#include <numpy/arrayobject.h>

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
namespace RASModels
{

/*---------------------------------------------------------------------------*\
                           Class kOmegaSST Declaration
\*---------------------------------------------------------------------------*/

template<class BasicTurbulenceModel>
class modelPropagationkOmegaSST
:
    public Foam::kOmegaSST
    <
        eddyViscosity<RASModel<BasicTurbulenceModel>>,
        BasicTurbulenceModel
    >
{
protected:

    //=================== Custom model propagation variables ==================
    // See modelPropagationkOmegaSST.C for the definition of these variables
    // and an elaborate description.

    // Scalars
    dimensionedScalar   useRST_;
    dimensionedScalar   usekDeficit_;
    dimensionedScalar   rampStartTime_;
    dimensionedScalar   rampEndTime_;
    dimensionedScalar   xi_;

    // Switches (booleans)
    Switch              useSigma_;
    Switch              modelRST_;
    Switch              modelkDeficit_;
    Switch              modelSigma_;


    // Fields
    volScalarField      kDeficit_;
    volSymmTensorField  bijDelta_;
    volScalarField      sigma_;
    volScalarField      y_;

    // Python interaction variables
    PyObject            *pName;
    PyObject            *pModule;
    PyObject            *model;
    PyObject            *model_args;
    PyObject            *array_2d;
    PyObject            *boolDict;
    PyArrayObject       *pValue;

    // Array size determining variables

    // The number of scalars to send to Python (e.g. for a vector three scalars
    // need to be send).
```

```
    // gradU=9 + k=1 + omega=1 + gradp=3 + gradk=3 + nut=1 + U=3 + walldist=1 +
    // nu=1 + curlU=3 = 26
    int num_scalars = 26;

    // The number of scalars returned by Python
    int num_return = 8; // kDeficit=1 + bijDelta=6 + sigma=1

    // Define number of mesh cells variable (set in constructor).
    int num_cells;

    // Define (num_cells x num_scalars) 1D array, holding flow variables in
    // the mesh; this is passed to Python.
    double* input_vals;


public:

    //============= Existing k-omega SST variables and functions =============

    typedef typename BasicTurbulenceModel::alphaField alphaField;
    typedef typename BasicTurbulenceModel::rhoField rhoField;
    typedef typename BasicTurbulenceModel::transportModel transportModel;


    //- Runtime type information
    TypeName("modelPropagationkOmegaSST");


    // Constructors

        //- Construct from components
        modelPropagationkOmegaSST
        (
            const alphaField& alpha,
            const rhoField& rho,
            const volVectorField& U,
            const surfaceScalarField& alphaRhoPhi,
            const surfaceScalarField& phi,
            const transportModel& transport,
            const word& propertiesName = turbulenceModel::propertiesName,
            const word& type = typeName
        );

    //- Solve the turbulence equations and correct the turbulence viscosity
    virtual void correct();

    //- Destructor
    virtual ~modelPropagationkOmegaSST()
    {
        delete input_vals;
    }

    //Probably not used, but left in just in case -Kaj
    tmp<Foam::fvVectorMatrix> divDevReff(volVectorField& U) const;

    //- Return the modified effective stress tensor
    virtual tmp<volSymmTensorField> devRhoReff() const;

    //- Return the modified source term for the momentum equation
    virtual tmp<fvVectorMatrix> divDevRhoReff(volVectorField& U) const;

    //- Return the modified source term for the momentum equation
    virtual tmp<fvVectorMatrix> divDevRhoReff
    (
        const volScalarField& rho,
        volVectorField& U
    ) const;

};


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
} // End namespace RASModels
} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#ifdef NoRepository
    #include "modelPropagationkOmegaSST.C"
#endif

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#endif

// ************************************************************************* //
```

D.1.2 modelPropagationkOmegaSST.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2016-2018 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*\
Modification of propagationkOmegaSST; now instead of reading kDeficit and
bijDelta, they are computed based on some model. The model calculation goes
via Python, such that no recompilation is necessary everytime a new model is
to be tested. A case is to be run with the turbulence model set to
modelPropagationkOmegaSST and using modelPropagationFoam as the solver. Then,
it will look for model_propagation.py in the folder. Within this file, a
function model should be defined that takes an array of shape
(nCells, num_scalars), with columns corresponding to: gradU, k, omega, gradp,
gradk, nut, U, walldist, nu and curlU. The function should return an array of
shape (nCells, 8) with the first column corresponding to kDeficit, the next
six to the components of bijDelta and the last one to the classifier sigma.
A template for model_definition.py is available in:
inversion/TurbFOAM-7/src/TurbulenceModels/turbulenceModels/
                                          RAS/modelPropagationkOmegaSST
If a classifier is not used, the boolean useSigma should be set to false in
the constant/turbulenceProperties file of the case. Furthermore, for isolated
testing of correction/classifier models, the boolean modelkDeficit/modelRST/
modelSigma should be set to false to use exact fields for this variable
(defined in the 0 directory).
\*---------------------------------------------------------------------------*/

#include "modelPropagationkOmegaSST.H"
#include "error.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
namespace RASModels
{

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * //

template<class BasicTurbulenceModel>
modelPropagationkOmegaSST<BasicTurbulenceModel>::modelPropagationkOmegaSST
(
    const alphaField& alpha,
    const rhoField& rho,
    const volVectorField& U,
    const surfaceScalarField& alphaRhoPhi,
    const surfaceScalarField& phi,
    const transportModel& transport,
    const word& propertiesName,
    const word& type
```

```
)
:
    Foam::kOmegaSST
    <
        eddyViscosity<RASModel<BasicTurbulenceModel>>,
        BasicTurbulenceModel
    >
    (
        type,
        alpha,
        rho,
        U,
        alphaRhoPhi,
        phi,
        transport,
        propertiesName
     ),

    //====================== Model propagation parameters ====================

    // Scalar by which the exact/modeled bijDelta correction is multiplied,
    // typically in the [0,1] range, mostly used for stabilization.
    useRST_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "useRST",
            this->coeffDict_,
            1.0
        )
    ),

    // Scalar by which the exact/modeled kDeficit correction is multiplied,
    // typically in the [0,1] range, mostly used for stabilization.
    usekDeficit_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "usekDeficit",
            this->coeffDict_,
            1.0
        )
    ),

    // Ramping gradually introduce corrections (both kDeficit and bijDelta)
    // to aid solver stability.  Before `rampStartTime` corrections are zero,
    // after `rampEndTime` they are 1.0.  Linear in between.  Default is
    // full correction from beginning.
    rampStartTime_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "rampStartTime",
            this->coeffDict_,
          dimTime,
            -1
        )
    ),
    rampEndTime_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "rampEndTime",
            this->coeffDict_,
          dimTime,
            0
        )
    ),

    // Corrections are multiplied by xi_ to apply ramping; xi_ is 0 before
    // rampStartTime, it linearly goes to 1 between rampStartTime and
    // rampEndTime and it stays 1 after rampEndTime. At each iteration,
    // xi_ is calculated based on the specified rampStartTime and rampEndTime.
```

```
    xi_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "xi_ramp",
            this->coeffDict_,
          dimless,
            1
        )
    ),

    // Boolean to decide whether to use a classifier or not (can be either a
    // model or existing field). Default is false such that no classifier
    // is used.
    useSigma_
    (
        Switch::lookupOrAddToDict
        (
            "useSigma",
            this->coeffDict_,
            false
        )
    ),

    // Switches to decide whether to use a model for a correction or frozen
    // fields in the 0 directory. If the model switch is true, a model is used
    // and the {var}Eq file (e.g. bijDeltaEq) should be present in the case
    // directory with the model equation to use. If the model switch is false,
    // the frozen field should be present in the zero directory. Default is to
    // use models for all corrections/classifier.
    modelRST_
    (
        Switch::lookupOrAddToDict
        (
            "modelRST",
            this->coeffDict_,
            true
        )
    ),
    modelkDeficit_
    (
        Switch::lookupOrAddToDict
        (
            "modelkDeficit",
            this->coeffDict_,
            true
        )
    ),
    modelSigma_
    (
        Switch::lookupOrAddToDict
        (
            "modelSigma",
            this->coeffDict_,
            true
        )
    ),

    //========================== Fields to be modelled ========================
    // The correction/classifier fields are initialized at a bit-specific small
    // value. The READ_IF_PRESENT directive is used to overwrite this value
    // with whatever is read in from the zero directory. If model{var}_ is
    // true, it is later checked whether the field was successfully read in by
    // checking whether the field still has the specific initialized value.
    // If model{var} is false, the initialized value is overwritten by
    // whatever is calculated based on the model equation.

    kDeficit_
    (
        IOobject(
      "kDeficit",
            this->runTime_.timeName(),
            this->mesh_,
```

```
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        this->mesh_,
        dimensionedScalar
        (
            "kDeficit",
            dimensionSet(0,2,-3,0,0,0,0),
            1.20813608515e-37
        )
    ),
    bijDelta_
    (
        IOobject
        (
            "bijDelta",
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        this->mesh_,
        dimensionedSymmTensor
        (
            "bijDelta",
            dimensionSet(0,0,0,0,0,0,0),
            symmTensor(1.20813608515e-37,0,0,0,0,0)
        )
    ),
    sigma_
    (
        IOobject(
      "sigma",
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        this->mesh_,
        dimensionedScalar
        (
            "sigma",
            dimensionSet(0,0,0,0,0,0,0),
            1.20813608515e-37
        )
    ),

    // Since the mesh is stationary, wall distance is only computed once at
    // the beginning of the run.
    y_(
       IOobject
       (
            "walldist",
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
       ),
       wallDist::New(this->mesh_).y()
    )

// Code run at the beginning of the run
{

    // Print the turbulence model coefficients (including the custom ones
    // defined above).
    if (type == typeName)
    {
        this->printCoeffs(type);
    }


    //========================== Python initialization ========================
```

```
    // Start up the Python interpreter
    Py_Initialize();

    // Add the run directory to the python path
    PyRun_SimpleString("import sys");
    PyRun_SimpleString("sys.path.append(\".\")");

    // initialize numpy array library
    import_array1();

    // Load the file model_definition.py which should be in the case directory
    pName = PyUnicode_DecodeFSDefault("model_definition");
    pModule = PyImport_Import(pName);

    // Load the model function inside model_definition.py
    model = PyObject_GetAttrString(pModule, "model");

    // Initialize tuple to be send to Python
    model_args = PyTuple_New(2);


    // Set num_cells equal to the number of cells in the mesh
    num_cells = this->mesh_.cells().size();

    // Initialize the (num_cells x num_scalars) sized input_vals array. It is
    // 1D on purpose so the whole array is contiguous in memory.
    input_vals = new double[num_cells*num_scalars];

}



//======================== Postprocessing functionality ======================
// Various forms of divDevReff/devRhoReff/divDevRhoReff are called by
// postProcessing functions needing a variable derived from the turbulence
// model. Since the turbulence model is modified by the correction terms,
// these functions need to be redefined with the correction terms to give
// correct postProcessing results.

// divDevReff function
template<class BasicTurbulenceModel>
tmp<fvVectorMatrix> modelPropagationkOmegaSST<BasicTurbulenceModel>::divDevReff
(
    volVectorField& U
) const
{
    Info << "In: modelPropagationkOmegaSST::divDevReff()" << endl;
    return
    (
      // Boussinesq part
      - fvc::div((this->alpha_*this->rho_*this->nuEff())*dev2(T(fvc::grad(U))))
      - fvm::laplacian(this->alpha_*this->rho_*this->nuEff(), U)

      // Nonlinear correction part
      + this->sigma_ * fvc::div(dev(2.*this->k_*this->bijDelta_) *
      useRST_ * xi_)
    );
}


// devRhoReff function
template<class BasicTurbulenceModel>
Foam::tmp<Foam::volSymmTensorField>
modelPropagationkOmegaSST<BasicTurbulenceModel>::devRhoReff() const
{
    Info << "In: modelPropagationkOmegaSST::devRhoReff()" << endl;
    return volSymmTensorField::New
    (
        // Boussinesq part
        IOobject::groupName("devRhoReff", this->alphaRhoPhi_.group()),
        (-(this->alpha_*this->rho_*this->nuEff()))
        *dev(twoSymm(fvc::grad(this->U_)))
```

```
        // Nonlinear correction part
        + this->sigma_ * dev(2.*this->k_*this->bijDelta_) * useRST_ * xi_
    );
}

// divDevRhoReff function
template<class BasicTurbulenceModel>
Foam::tmp<Foam::fvVectorMatrix>
modelPropagationkOmegaSST<BasicTurbulenceModel>::divDevRhoReff
(
    volVectorField& U
) const
{
    Info << "In: modelPropagationkOmegaSST::divDevRhoReff()" << endl;
    return
    (
      // Boussinesq part
      - fvc::div((this->alpha_*this->rho_*this->nuEff())*dev2(T(fvc::grad(U))))
      - fvm::laplacian(this->alpha_*this->rho_*this->nuEff(), U)

      // Nonlinear correction part
      + this->sigma_ * fvc::div(dev(2.*this->k_*this->bijDelta_) *
      useRST_ * xi_)
    );
}

// divDevRhoReff function (different input template)
template<class BasicTurbulenceModel>
Foam::tmp<Foam::fvVectorMatrix>
modelPropagationkOmegaSST<BasicTurbulenceModel>::divDevRhoReff
(
    const volScalarField& rho,
    volVectorField& U
) const
{
    Info << "In: modelPropagationkOmegaSST::divDevRhoReff()" << endl;
    return
    (
      // Boussinesq part
      - fvc::div((this->alpha_*rho*this->nuEff())*dev2(T(fvc::grad(U))))
      - fvm::laplacian(this->alpha_*rho*this->nuEff(), U)

      // Nonlinear correction part
      + this->sigma_ * fvc::div(dev(2.*this->k_*this->bijDelta_) *
      useRST_ * xi_)
    );
}




//============================ Main function ================================
// Given below is the ::correct() function of the turbulence model, which is
// called each iteration to obtain a new estimate of the Reynolds stress
// tensor (RST). Besides the normal calculation of the RST using k-omega SST,
// the Python interaction to calculate correction fields is also included
// in this function.

template<class BasicTurbulenceModel>
void modelPropagationkOmegaSST<BasicTurbulenceModel>::correct()
{
    //====================== Existing k-omega SST code ======================

    if (!this->turbulence_)
    {
        return;
    }

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

```
    const volVectorField& U = this->U_;
    volScalarField& nut = this->nut_;
    fv::options& fvOptions(fv::options::New(this->mesh_));

    // Manually load omega_ and k_; in base kOmegaSST this happends
    // automatically as they are defined there.
    volScalarField& omega_ = this->omega_;
    volScalarField& k_ = this->k_;

    BasicTurbulenceModel::correct();

    volScalarField::Internal divU
    (
        fvc::div(fvc::absolute(this->phi(), U))()()
    );

    tmp<volTensorField> tgradU = fvc::grad(U);
    volScalarField S2(2*magSqr(symm(tgradU())));
    volScalarField GbyNu(dev(twoSymm(tgradU())) && tgradU());
    volScalarField::Internal G(this->GName(), nut()*GbyNu);

    volScalarField CDkOmega
    (
        (2*this->alphaOmega2_)*(fvc::grad(k_) & fvc::grad(omega_))/omega_
    );

    volScalarField F1(this->F1(CDkOmega));
    volScalarField F23(this->F23());

    // ================= Custom variable calculation ============================

    // Calculate vorticity (needed for q_V feature)
    tmp<volVectorField> tcurlU = fvc::curl(U);

    Info << "In: modelPropagationkOmegaSST.correct()" << endl;

    // Calculate the ramping variable xi_
    const dimensionedScalar time = this->runTime_;
    xi_ = (time < rampStartTime_)? 0.0:
          (time > rampEndTime_)? 1.0:
          (time - rampStartTime_) / (rampEndTime_ - rampStartTime_);
    Info << "Corrections: xi = " << xi_.value() <<
          ", kDeficit factor = " << (xi_*usekDeficit_).value() <<
          ", bijDelta factor = " << (xi_*useRST_).value() << endl;


    //========================== Python interaction during run ================

    //Load the latest p_ as a variable
    word pName_ = ("p");
    tmp<volScalarField> p_ = U.db().lookupObject<volScalarField>(pName_);

    //Get the gradient fields of p and k
    tmp<volVectorField> tgradp = fvc::grad(p_);
    tmp<volVectorField> tgradk = fvc::grad(k_);

    // Loop over each mesh cell and store relevant variables in the array which
    // is passed to Python.
    forAll(k_.internalField(), id)
    {

        // First nine elements correspond to the components of the gradU tensor
        input_vals[id*num_scalars + 0] = tgradU()[id][0];
        input_vals[id*num_scalars + 1] = tgradU()[id][1];
        input_vals[id*num_scalars + 2] = tgradU()[id][2];
        input_vals[id*num_scalars + 3] = tgradU()[id][3];
        input_vals[id*num_scalars + 4] = tgradU()[id][4];
        input_vals[id*num_scalars + 5] = tgradU()[id][5];
        input_vals[id*num_scalars + 6] = tgradU()[id][6];
        input_vals[id*num_scalars + 7] = tgradU()[id][7];
        input_vals[id*num_scalars + 8] = tgradU()[id][8];

        // Tenth element corresponds to k
```

```
        input_vals[id*num_scalars + 9] = k_[id];

        // Eleventh element corresponds to omega
        input_vals[id*num_scalars + 10] = omega_[id];

        // Twelfth-fourteenth elements correspond to components of the
        // gradp vector.
        input_vals[id*num_scalars + 11] = tgradp()[id][0];
        input_vals[id*num_scalars + 12] = tgradp()[id][1];
        input_vals[id*num_scalars + 13] = tgradp()[id][2];

        // Fifteenth-seventeenth elements correspond to components of the
        // gradk vector.
        input_vals[id*num_scalars + 14] = tgradk()[id][0];
        input_vals[id*num_scalars + 15] = tgradk()[id][1];
        input_vals[id*num_scalars + 16] = tgradk()[id][2];

        // Eighteenth element corresponds to nu_t
        input_vals[id*num_scalars + 17] = nut()[id];

        // Nineteenth-twenty-first elements correspond to components of the
        // U vector.
        input_vals[id*num_scalars + 18] = U()[id][0];
        input_vals[id*num_scalars + 19] = U()[id][1];
        input_vals[id*num_scalars + 20] = U()[id][2];

        // Twenty-second element corresponds to the wall distance
        input_vals[id*num_scalars + 21] = y_[id];

        // Twenty-third element corresponds to the viscosity
        input_vals[id*num_scalars + 22] = this->nu()().internalField()[id];

        // Twenty-fourth-twenty-sixth elements corresponds to components
        // of curlU vector.
        input_vals[id*num_scalars + 23] = tcurlU()[id][0];
        input_vals[id*num_scalars + 24] = tcurlU()[id][1];
        input_vals[id*num_scalars + 25] = tcurlU()[id][2];

    }

    // Clear temporary arrays with gradients/vorticity
    tgradp.clear();
    tgradk.clear();
    tcurlU.clear();

    // Get the array dimensions in a format understood by Numpy
    npy_intp dim[] = {num_cells, num_scalars};

    // Convert the input_vals array to a format understood by Numpy
    array_2d = PyArray_SimpleNewFromData(2, dim, NPY_DOUBLE, &input_vals[0]);

    // Set the first element of the tuple to the array to be send to Python
    PyTuple_SetItem(model_args, 0, array_2d);

    // Create dictionary of booleans relevant for Python and set it as the
    // second element of the tuple to be send to Python.
    boolDict = PyDict_New();
    PyDict_SetItemString(boolDict, "modelkDeficit",
                         PyLong_FromLong(modelkDeficit_));
    PyDict_SetItemString(boolDict, "modelRST", PyLong_FromLong(modelRST_));
    PyDict_SetItemString(boolDict, "useSigma", PyLong_FromLong(useSigma_));
    PyDict_SetItemString(boolDict, "modelSigma", PyLong_FromLong(modelSigma_));
    PyTuple_SetItem(model_args, 1, boolDict);


    // Call the model() function in Python with the input_vals in a tuple as
    // the argument. The array returned by Python is loaded into pReturn,
    // which is initialized here.
    PyObject* pReturn = PyObject_CallObject(model, model_args);

    // Cast pReturn to a PyArrayObject and store it in pValue
    pValue = reinterpret_cast<PyArrayObject*>(pReturn);
```

```
    // Check if the returned array has the expected number of rows
    // (corresponding to number of cells). If not, throw an error.
    if (PyArray_DIMS(pValue)[0] != num_cells){
        FatalError << "Number of rows (corresponding to the number of mesh "
        << "cells) returned by Python does not correspond to the number of "
        << "mesh cells sent to Python." << nl << exit(FatalError);
    }

    // Check if the returned array has the expected number of columns
    // (corresponding to first column kDeficit, next six bijDelta and last
    // column sigma). If not, throw an error.
    if (PyArray_DIMS(pValue)[1] != num_return){
        FatalError << "Number of columns of the array returned by Python does"
        << " not correspond to the expected number of columns (8). The first "
        << "column should be kDeficit, the next six columns components of "
        << "bijDelta (XX, XY, XZ, YY, ZZ) and the last column sigma."
        << nl << exit(FatalError);
    }

    // If kDeficit is modeled, extract it as the first column of the returned
    // array. If it is not modeled, check whether the file was succesfully
    // read in. If not, throw an error.
    if (modelkDeficit_)
    {
        forAll(kDeficit_.internalField(), id)
        {
            kDeficit_[id] = *((double*)PyArray_GETPTR2(pValue, id, 0));
        }
    }
    else if (kDeficit_[0] == 1.20813608515e-37)
    {
        FatalError << "modelkDeficit set to false, but no kDeficit file found"
        << nl << exit(FatalError);
    }

    // If bijDelta is modeled, extract it as the next six columns of the
    // returned array. If it is not modeled, check whether the file was
    // succesfully read in. If not, throw an error.
    if (modelRST_)
    {
        forAll(bijDelta_.internalField(), id)
        {
            bijDelta_[id][0] = *((double*)PyArray_GETPTR2(pValue, id, 1));
            bijDelta_[id][1] = *((double*)PyArray_GETPTR2(pValue, id, 2));
            bijDelta_[id][2] = *((double*)PyArray_GETPTR2(pValue, id, 3));
            bijDelta_[id][3] = *((double*)PyArray_GETPTR2(pValue, id, 4));
            bijDelta_[id][4] = *((double*)PyArray_GETPTR2(pValue, id, 5));
            bijDelta_[id][5] = *((double*)PyArray_GETPTR2(pValue, id, 6));
      }
    }
    else if (bijDelta_[0][0] == 1.20813608515e-37)
    {
        FatalError << "modelbijDelta set to false, but no bijDelta file found"
        << nl << exit(FatalError);
    }

    // If sigma is used, either read it in from a file or calculate it.
    // If sigma is not used, set it to 1. everywhere.
    if (useSigma_)
    {
        // If modelSigma_ is true, extract the last column of pValue as sigma.
        // If useSigma_ is false, check if the file was succesfully read in,
        // otherwise throw an error.
        if (modelSigma_)
        {
            forAll(sigma_.internalField(), id)
            {
                sigma_[id] = *((double*)PyArray_GETPTR2(pValue, id, 7));
            }
        }
        else if (sigma_[0] == 1.20813608515e-37)
        {
            FatalError << "modelSigma set to false, but no sigma file found"
```

```
            << nl << exit(FatalError);
        }
    }
    else
    {
        forAll(sigma_.internalField(), id)
        {
            sigma_[id] = 1.;
        }
    }

    //Free the memory of pReturn to prevent a memory leak
    Py_DECREF(pReturn);


    //======================== Existing k-omega SST code ========================
    // Solving the omega and k equations of the k-omega SST turbulence model,
    // slightly modified to add the corrections just calculated in Python.

    // Modified version of the G variable which also includes the effect of
    // the corrections.
    volScalarField G2
    (
         "G2",
    nut*GbyNu - xi_ * useRST_ * sigma_ * (2*(this->k_)*bijDelta_ && tgradU())
    );

    // Finally clear the temporary gradient of U variable.
    tgradU.clear();

    // --------------------------- omega equation ---------------------------
    {
        volScalarField::Internal gamma(this->gamma(F1));
        volScalarField::Internal beta(this->beta(F1));

        // Turbulent frequency equation
        tmp<fvScalarMatrix> omegaEqn
        (
            fvm::ddt(alpha, rho, omega_)
          + fvm::div(alphaRhoPhi, omega_)
          - fvm::laplacian(alpha*rho*this->DomegaEff(F1), omega_)
         ==
            alpha()*rho()*gamma
           *min
            (
                // Production modified due to RST correction
              G2 / nut(),
                (this->c1_/this->a1_)*this->betaStar_*omega_()
               *max(this->a1_*omega_(), this->b1_*F23()*sqrt(S2()))
            )
          // Production modified due to k-equation correction
        + alpha()*rho()*gamma*sigma_*kDeficit_/nut()*(xi_ * usekDeficit_)
          - fvm::SuSp((2.0/3.0)*alpha()*rho()*gamma*divU, omega_)
          - fvm::Sp(alpha()*rho()*beta*omega_(), omega_)
          - fvm::SuSp
            (
                alpha()*rho()*(F1() - scalar(1))*CDkOmega()/omega_(),
                omega_
            )
          + this->Qsas(S2(), gamma, beta)
          + this->omegaSource()
          + fvOptions(alpha, rho, omega_)
        );

      // Update omega and G at the wall
      omega_.boundaryFieldRef().updateCoeffs();

        omegaEqn.ref().relax();
        fvOptions.constrain(omegaEqn.ref());
        omegaEqn.ref().boundaryManipulate(omega_.boundaryFieldRef());
        solve(omegaEqn);
        fvOptions.correct(omega_);
        bound(omega_, this->omegaMin_);
```

```
    }

    // ------------------- Turbulent kinetic energy equation -------------------
    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(alpha, rho, k_)
      + fvm::div(alphaRhoPhi, k_)
      - fvm::laplacian(alpha*rho*this->DkEff(F1), k_)
     ==
      // Production modified due to RST correction
        alpha()*rho()*this->Pk(G2)
      // Production modified due to k-equation correction
    + alpha()*rho()*sigma_*kDeficit_()*(xi_ * usekDeficit_)
      - fvm::SuSp((2.0/3.0)*alpha()*rho()*divU, k_)
      - fvm::Sp(alpha()*rho()*this->epsilonByk(F1, F23), k_)
      + this->kSource()
      + fvOptions(alpha, rho, k_)
    );

    kEqn.ref().relax();
    fvOptions.constrain(kEqn.ref());
    solve(kEqn);
    fvOptions.correct(k_);
    bound(k_, this->kMin_);

    this->correctNut(S2, F23);
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace RASModels
} // End namespace Foam

// ************************************************************************* //
```

D.2 model_definition.py Python file called by modelPropagationkOmegaSST

```
'''Template for the Python file used by modelPropagationFoam (using
modelPropagation... as a turbulence model). This file should be in the case
directory where the model is propagated. With the current setup, this file looks
for three other files; kDeficitEq, bijDeltaEq, and sigmaEq, defining the
equation for kDeficit, bijDelta and sigma respectively. The equation file for
kDeficit/bijDelta/sigma only needs to be present if modelkDeficit/modelRST/
modelSigma is true.

Note: when there is an error in this Python file, it will not be printed by
OpenFOAM. Rather, a segmentation error will occur. In this case, it is
recommended to uncomment the last line of this file calling the model() function
with a dummy input to check for errors.

Author : Kaj'''

#===============================================================================
#Import required libraries

import numpy as np
import os
from sparta.features import FlowFeatures
from sparta.util import rdiv, rlog, sqrt_abs
from readOFInternalField import readOFInternalField
import contextlib
import gc

#Use magnitude of gradU tensor to nondimensionalize S and W
meanFlowTimeScale = True

#===============================================================================
#Reading in the equations


#Read the contents of the kDeficitEq, bijDeltaEq and sigmaEq files,
#removing enters. If the file is not present, the content
#variable (kDeficitEq/bijDeltaEq/sigmaEq) is set to False.
if os.path.isfile('./kDeficitEq'):
    with open('./kDeficitEq') as fkDeficit:
        kDeficitEq = fkDeficit.read().replace('\n','')
else:
    kDeficitEq = False

if os.path.isfile('./bijDeltaEq'):
    with open('./bijDeltaEq') as fbijDelta:
        bijDeltaEq = fbijDelta.read().replace('\n','')
else:
    bijDeltaEq = False

if os.path.isfile('./sigmaEq'):
    with open('./sigmaEq') as fSigma:
        sigmaEq = fSigma.read().replace('\n','')
else:
    sigmaEq = False


#Read in the 0/kDeficit and 0/bijDelta files if present.
#if modelkDeficit or modelRST are false, they are later updated with their
#calculated value.
kDeficitIn0, bijDeltaIn0 = False, False
if os.path.isfile('./0/kDeficit'):
    kDeficit0 = readOFInternalField('./0/kDeficit')
    kDeficitIn0 = True
if os.path.isfile('./0/bijDelta'):
    bijDelta0 = readOFInternalField('./0/bijDelta')
    bijDeltaIn0 = True


#Dictionary of functions which may appear in the equation strings
```

```
funcDict = {'exp' : np.exp, 'abs' : np.abs, 'tanh' : np.tanh,
            'sqrt' : np.sqrt, 'np' : np,#Numpy
            'rdiv' : rdiv, 'rlog' : rlog, 'sqrt_abs' : sqrt_abs} #Custom


#==============================================================================
#Main function run by OpenFOAM as well as helper functions



def model(InputArray, boolDict):
    '''Main function that is called by OpenFOAM to evaluate kDeficit,
    bijDelta and sigma. Loads the lazy flow.vv dictionary from the
    sparta.features library into the featureDict variable. If modelkDeficit/
    modelRST/modelSigma is true, kDeficit/bijDelta/sigma are calculated by
    evaluating their formatted equation. For consistency, all arrays will be
    three dimensional, with the first axes corresponding to different cells.
    This means that:
        -scalars have shape (nMeshCells, 1, 1)
        -vectors have shape (nMeshCells, 3, 1)
        -tensors have shape (nMeshCells, 3, 3)


    Input:
    InputArray : Array of shape (nMeshCells, 26), the columns corresponding to:
                    -the 9 components of the velocity gradient tensor
                    -turbulent kinetic energy k
                    -specific turbulence dissipation omega
                    -the 3 components of the pressure gradient vector
                    -the 3 components of the turbulent kinetic energy gradient
                        vector
                    -the eddy viscosity nut
                    -the 3 components of the velocity vector
                    -the wall distance
                    -the molecular viscosity nu
                    -the 3 components of the vorticity vector
    boolDict : Dictionary holding user defined booleans of whether to use and
               model the variables. Should have the following keys with either
               True or False as the corresponding entry:
                   -useSigma : whether to use a classifier
                   -modelSigma : whether to use a model for the classifier
                   -modelkDeficit : whether to use a model for kDeficit
                   -modelRST : whether to use a model for bijDelta

    Output:
    OutputArray : Array of shape (nMeshCells, 8), the columns corresponding to:
                    -kDeficit
                    -the six components of the symmetric bijDelta tensor
                    -sigma'''

    #Get the number of cells
    NCells = InputArray.shape[0]

    #Setup a FlowFeatures object from InputArray, this does not have the features
    #set up yet.
    flow = FlowFeatures.from_inputarray(InputArray)

    #Set up the features for the FlowFeatures object. If the program is not run
    #directly (but presumably by OpenFOAM), suppress the print output for better
    #performance.
    if __name__ != "__main__":
        with contextlib.redirect_stdout(None):
            flow.setup_features(meanFlowTimeScale=meanFlowTimeScale)
    else:
        flow.setup_features(meanFlowTimeScale=meanFlowTimeScale)

    #Extract the lazy feature dictionary from the FlowFeatures object
    featureDict = flow.vv

    #Add "const" to the featureDict as a variable; this was used in some old
    #equations, it is simply 1 everywhere.
    featureDict['const'] = lambda : np.ones(NCells)
```

```
#------------------------------------------------------------------------------
#Evaluating the equations and returning kDeficit and bijDelta back to
#OpenFOAM.

#Evaluate the read in kDeficit equation if modelkDeficit is True.
#Else, set kDeficit to nan everywhere as it is not used anyway.
if boolDict['modelkDeficit']:

    if not kDeficitEq:
        raise Exception(('modelkDeficit is set to True, but no kDeficitEq '
                         'is provided in the case directory.'))
    featureDict['kDeficit'] = eval(kDeficitEq, funcDict, featureDict)
else:

    featureDict['kDeficit'] = np.ones(NCells)*np.nan

#Evaluate the read in bijDelta equation if modelRST is True.
#Else, set kDeficit to nan everywhere as it is not used anyway.
if boolDict['modelRST']:

    if not bijDeltaEq:
        raise Exception(('modelRST is set to True, but no bijDeltaEq '
                         'is provided in the case directory.'))

    featureDict['bijDelta'] = eval(bijDeltaEq, funcDict, featureDict)
else:

    featureDict['bijDelta'] = np.ones((6, NCells))*np.nan



#Dot product between bijDelta and gradU tensor, could be used by a
#classifier criterion.
featureDict['bijDeltaGradU'] = lambda :\
            ((featureDict['bijDelta'].T)[:,0]*featureDict['gradU'][:,0,0] +\
             (featureDict['bijDelta'].T)[:,1]*featureDict['gradU'][:,0,1] +\
             (featureDict['bijDelta'].T)[:,2]*featureDict['gradU'][:,0,2] +\
             (featureDict['bijDelta'].T)[:,1]*featureDict['gradU'][:,1,0] +\
             (featureDict['bijDelta'].T)[:,3]*featureDict['gradU'][:,1,1] +\
             (featureDict['bijDelta'].T)[:,4]*featureDict['gradU'][:,1,2] +\
             (featureDict['bijDelta'].T)[:,2]*featureDict['gradU'][:,2,0] +\
             (featureDict['bijDelta'].T)[:,4]*featureDict['gradU'][:,2,1] +\
             (featureDict['bijDelta'].T)[:,5]*featureDict['gradU'][:,2,2])

#Boussinesq production of k, could be used by a classifier criterion.
featureDict['PkBoussinesq'] = lambda : 2*featureDict['nut']*np.sum(\
            ((featureDict['gradU'] + featureDict['gradU'].swapaxes(-1, -2))*\
             featureDict['gradU']), axis=(-1,-2))


#Evaluate the read in sigma equation if useSigma and modelSigma are True.
#Otherwise set sigma to nan as it is not used anyway.
if boolDict['useSigma'] and boolDict['modelSigma']:

    #Check if the sigmaEq file was successfully read in.
    if not sigmaEq:
        raise Exception(('useSigma and modelSigma are set to True, but no'
                         'sigmaEq is provided in the case directory.'))

    #If kDeficit appears in the sigma equation and is not modeled, check
    #if it was successfully read in from the 0 directory and add it to
    #featureDict.
    if 'kDeficit' in sigmaEq:
        if not boolDict['modelkDeficit']:
            if not kDeficitIn0:
                raise Exception(('modelkDeficit is set to False, but no '
                                 'kDeficit file provided in the 0 directory.'))
            featureDict['kDeficit'] = kDeficit0

    #If bijDelta appears in the sigma equation and is not modeled, check
    #if it was successfully read in from the 0 directory and add it to
    #featureDict.
    if 'bijDelta' in sigmaEq:
```

```
            if not boolDict['modelRST']:
                if not bijDeltaIn0:
                    raise Exception(('modelbijDelta is set to False, but no '
                                     'bijDelta file provided in the 0 directory.'))
                featureDict['bijDelta'] = bijDelta0.T

        sigma = eval(sigmaEq, funcDict, featureDict)

    else:
        sigma = np.ones(NCells)*np.nan


    #Create the return array, with 8 columns and NCells rows.
    #The first column corresponds to kDeficit, the next 6 to the components
    #of bijDelta and the last to sigma.
    ReturnArray = np.concatenate([featureDict['kDeficit'].reshape((-1,1)),
                                  featureDict['bijDelta'].T,
                                  sigma.reshape((-1,1))], axis=-1)

    #Cleanup to prevent memory leaks
    del InputArray, flow, featureDict, sigma, boolDict

    gc.collect()

    #Return the array; first column corresponds to kDeficit, last six columns
    #to the unique components of bijDelta.
    return ReturnArray


#================================================================================
#Debugging switch: uncomment to run the model from Python to check for errors.

'''
boolDict = {'modelkDeficit' : True,
            'modelRST' : False,
            'useSigma' : True,
            'modelSigma' : True}

returnArray = model(np.ones((2209,26)), boolDict)#'''
```

D.3 Custom solver for model propagation (modelPropagationFoam)

D.3.1 modelPropagationFoam.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2018 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Application
    modelPropagationFoam

Description
    Steady-state solver for incompressible, turbulent flow, using the SIMPLE
    algorithm. Modified to prevent a numpy error when evaluating a custom
    turbulence model in Python.

\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulentTransportModel.H"
#include "simpleControl.H"
#include "fvOptions.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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


    //============= Code modified compared to simpleFoam.C ==================

    // simpleFoam.C includes the default postProcess.H file here; see the
    // modifiedPostProcess.H file for the differences.
    #include "modifiedPostProcess.H"

    // The following part is instead of #include setRootCaseLists.H; this is
    // to prevent a numpy error.
    #include "listOptions.H"
    Foam::argList args(argc, argv, true,true,/*initialise=*/false);
    if (!args.checkRootCase())
    {
        Foam::FatalError.exit();
    }
    #include "listOutput.H"


    //================== Existing simpleFoam.C code ========================

    #include "createTime.H"
```

```
    #include "createMesh.H"
    #include "createControl.H"
    #include "createFields.H"
    #include "initContinuityErrs.H"

    turbulence->validate();

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nStarting time loop\n" << endl;

    while (simple.loop(runTime))
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

        // --- Pressure-velocity SIMPLE corrector
        {
            #include "UEqn.H"
            #include "pEqn.H"
        }

        laminarTransport.correct();
        turbulence->correct();

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}


// ************************************************************************* //
```

D.3.2 modifiedPostProcess.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2016-2018 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Global
    postProcess

Description
    Execute application functionObjects to post-process existing results.

    If the "dict" argument is specified the functionObjectList is constructed
    from that dictionary otherwise the functionObjectList is constructed from
    the "functions" sub-dictionary of "system/controlDict"

    Multiple time-steps may be processed and the standard utility time
    controls are provided.

\*---------------------------------------------------------------------------*/

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#ifndef CREATE_TIME
    #define CREATE_TIME createTime.H
#endif

#ifndef CREATE_MESH
    #define CREATE_MESH createMesh.H
#endif

#ifndef CREATE_FIELDS
    #define CREATE_FIELDS createFields.H
#endif

#ifndef CREATE_CONTROL
    #define CREATE_CONTROL createControl.H
#endif

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#define INCLUDE_FILE(X) INCLUDE_FILE2(X)
#define INCLUDE_FILE2(X) #X

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

Foam::argList::addBoolOption
(
    argList::postProcessOptionName,
    "Execute functionObjects only"
);

if (argList::postProcess(argc, argv))
```

```
{
    Foam::timeSelector::addOptions();
    #include "addRegionOption.H"
    #include "addFunctionObjectOptions.H"

    // Set functionObject post-processing mode
    functionObject::postProcess = true;

    //============= Code modified compared to postProcess.H ===================

    // The following part is instead of #include setRootCases.H; this is to
    // prevent a numpy error.
    Foam::argList args(argc, argv, true,true,/*initialise=*/false);
    if (!args.checkRootCase())
    {
        Foam::FatalError.exit();
    }

    //=================== Existing postProcess.H code =========================

    if (args.optionFound("list"))
    {
        functionObjectList::list();
        return 0;
    }

    #include INCLUDE_FILE(CREATE_TIME)
    Foam::instantList timeDirs = Foam::timeSelector::select0(runTime, args);
    #include INCLUDE_FILE(CREATE_MESH)

    #ifndef NO_CONTROL
    #include INCLUDE_FILE(CREATE_CONTROL)
    #endif

    forAll(timeDirs, timei)
    {
        runTime.setTime(timeDirs[timei], timei);

        Info<< "Time = " << runTime.timeName() << endl;

        FatalIOError.throwExceptions();

        try
        {
            #include INCLUDE_FILE(CREATE_FIELDS)

            #ifdef CREATE_FIELDS_2
            #include INCLUDE_FILE(CREATE_FIELDS_2)
            #endif

            #ifdef CREATE_FIELDS_3
            #include INCLUDE_FILE(CREATE_FIELDS_3)
            #endif

            // Externally stored dictionary for functionObjectList
            // if not constructed from runTime
            dictionary functionsControlDict("controlDict");

            HashSet<word> selectedFields;

            // Construct functionObjectList
            autoPtr<functionObjectList> functionsPtr
            (
                functionObjectList::New
                (
                    args,
                    runTime,
                    functionsControlDict,
                    selectedFields
                )
            );

            functionsPtr->execute();
```

```
        }
        catch (IOerror& err)
        {
            Warning<< err << endl;
        }

        // Clear the objects owned by the mesh
        mesh.objectRegistry::clear();

        Info<< endl;
    }

    Info<< "End\n" << endl;

    return 0;
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#undef INCLUDE_FILE
#undef INCLUDE_FILE2

#undef CREATE_MESH
#undef CREATE_FIELDS
#undef CREATE_CONTROL

// ************************************************************************* //
```

E CuRTA Python library

```
'''Program to use non-linear least squares in combination with deficit fitting to
symbolically regress nonlinear functions to predict a target. The main function
to call is regressModel. This function starts by building a library of terms
based on the specified features, functions and bases. Then, the first term is
regressed as the optimal library term. Further terms are found by fitting the
deficit between the target and the established terms. After an additional
optimal term has been found, coefficients of the whole expression are refit.

Author : Kaj Hoefnagel
'''


#==============================================================================
#Libraries to include
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from functools import partial
import time
from sklearn.metrics import r2_score
from sparta.util import LazyDict
from sparta.features import FlowFeatures
import os

#==============================================================================
#Bases names

#kDeficit bases names without mean flow timescale: epsilon, G1, ..., G10
PkScalars = ['epsilon', *[f'G{i}' for i in range(1,11)]]

#bijDelta bases names without mean flow timescale: T1, ..., T10
baseTensors = [f'T{i}' for i in range(1,11)]

#kDeficit bases names with mean flow timescale: epsilon, G1_s, ..., G10_s
PkScalars_s = ['epsilon', *[f'G{i}_s' for i in range(1,11)]]

#bijDelta bases names without mean flow timescale: T1_s, ..., T10_s
baseTensors_s = [f'T{i}_s' for i in range(1,11)]


#==============================================================================
#Goodness of fit functions

def getRMSE(funcTarget, exactTarget, targetFunc=None, weights=None):
    '''Function to calculate the root mean squared error between the functional
    approximation of a target and the exact target.

    Inputs:
    -funcTarget : array_like
            Array of approximations of the target.
    -exactTarget : array_like
            Array of the exact target, must have the same shape as funcTarget.
    -targetFunc : {callable, None}, optional
            Function that takes an equation string as argument and returns a new
            string in which the input string is manipulated. For example, if the
            equation string "q_nu*T2" is inputted (symmetric tensor),
            a targetFunc could be the trace, returning the first, fourth and
            sixth column; targetFunc("q_nu*T2") = "(q_nu*T2)[:,[0,3,5]]".
            This targetFunc will be applied to the funcTarget and exactTarget
            before calculating the RMSE. If the RMSE should be calculated
            directly from funcTarget and exactTarget, targetFunc is set to None.
            The default is None.
    -weights : {array_like, None}, optional
            Array of weights by which to weigh the root mean squared error.
            Should have the same shape as funcTarget evaluated through the
            targetFunc. If each point should be weighed equally, weights should
            be None. The default is None.

    Output:
```

```
    -RMSE : array_like
            (Weighed) root mean square error at each point, same shape as
            targetFunc(funcTarget).'''

    #If the specified targetFunc is not None, evaluate it on funcTarget and
    #exactTarget to apply the targetFunc.
    if targetFunc is not None:
        funcTarget = eval(targetFunc('funcTarget'))
        exactTarget = eval(targetFunc('exactTarget'))

    #Return the square root of the weighted average of squares of differences
    #(weighted RMSE).
    return np.sqrt(np.average((funcTarget-exactTarget)**2, weights=weights))


def getRSquared(funcTarget, exactTarget, targetFunc=None, weights=None):
    '''Function to calculate the (weighted) coefficient of determination (R^2)
    given the functional approximation of a target and the exact target.

    Inputs:
    -funcTarget : array_like
            Array of approximations of the target.
    -exactTarget : array_like
            Array of the exact target, must have the same shape as funcTarget.
    -targetFunc : {callable, None}, optional
            Function that takes an equation string as argument and returns a new
            string in which the input string is manipulated. For example, if the
            equation string "q_nu*T2" is inputted (symmetric tensor),
            a targetFunc could be the trace, returning the first, fourth and
            sixth column; targetFunc("q_nu*T2") = "(q_nu*T2)[:,[0,3,5]]".
            This targetFunc will be applied to the funcTarget and exactTarget
            before calculating the R^2. If the R^2 should be calculated
            directly from funcTarget and exactTarget, targetFunc is set to None.
            The default is None.
    -weights : {array_like, None}, optional
            Array of weights by which to weigh coefficient of determination.
            Should have the same shape as funcTarget evaluated through the
            targetFunc. If each point should be weighed equally, weights should
            be None. The default is None.

    Output:
    -R2 : array_like
            Weighted coefficient of determination (R^2) at each point, same shape
            as targetFunc(funcTarget).'''

    #If the specified targetFunc is not None, evaluate it on funcTarget and
    #exactTarget to apply the targetFunc.
    if targetFunc is not None:
        funcTarget = eval(targetFunc('funcTarget'))
        exactTarget = eval(targetFunc('exactTarget'))

    #Return the weighted R^2 score.
    return r2_score(exactTarget, funcTarget, sample_weight=weights)


#==========================================================================
#Non-linear least squares fit of equation functions


class fitClass:
    '''Class to allow optimization of a function with extra arguments with an
    arbirary number of fit coefficients. Initialized with eqStr and varDict
    (see fitEqStr function for their description). Then, the fitFunc function
    should be called by curve_fit, which only needs to pass the x-data (simply 1)
    and the coefficients. The eqStr and varDict are than accessed within the
    object to return the evaluated eqStr with the specified coefficients.'''

    def __init__(self, eqStr, varDict):
        self.eqStr = eqStr
        self.varDict = varDict

    def fitFunc(self, x, *coeffs):
        return eval(self.eqStr, {'coeffs' : coeffs, 'np' : np, **self.varDict,
```

```
                                'globFuncs' : globFuncs})


def fitEqStr(eqStr, yData, varDict, nCoeffs, maxfev=None, weights=None,
             p0=None, bounds=None):
    '''Function to fit the coefficients in an expression using (weighted)
    non-linear least squares, implemented in the scipy.optimize.curve_fit
    function. This type of regression can be unstable, so possibility to specify
    an initial guess p0 and bounds.

    Inputs:
    -eqStr : str
            String of the equation of which to fit the coefficients. Each
            coefficient to be regressed should be denoted 'coeffs[i]', with
            i from zero to the number of coefficients. For instance, to
            regress C*tanh(C*q_nu), eqStr should be:
                "coeffs[0]*np.tanh(coeffs[1]*q_nu)".
            The output of eqStr should be 1D, so fitting of a tensor expression
            should already have a targetFunc applied to eqStr.
    -yData : (M,) array_like,
            Array of target data to be fitted, e.g. kDeficit. If the target is
            a tensor, a targetFunc should already be applied such that it is 1D.
    -varDict : dict
            Dictionary with variables/features as keys and array as the
            corresponding values. Should at least contain the variables in eqStr.
            For example, if eqStr is a function of q_nu and T2, these should be
            keys of varDict.
    -nCoeffs : int
            Number of coefficients to regress in eqStr. TODO: automatically
            read nCoeffs from eqStr.
    -maxfev : {int, None}, optional
            Maximum number of function evaluations to perform within the
            non-linear least squares optimization. If None is specified, the
            default value is used. The default is None.
    -weights : {array_like, None}, optional
            Array of weights by which to weigh the non-linear least squares.
            Should have the same shape as yData. If each point should be weighed
            equally, weights should be None. The default is None.
    -p0 : {array_like, None}, optional
            Initial guess of the coefficients, should have shape (nCoeffs,).
            Alternatively, if None is specified, 1 is used for the initial guess
            of each coefficient. The default is None.
    -bounds : {2-tuple, None}, optional
            Bounds on the coefficients, given as a 2-tuple containing arrays of
            length nCoeffs with the lower and upper bound of each coefficient.
            In order to use no lower/upper bound, use -np.inf/np.inf. If None
            is specified, no bounds are used for all coefficients.
            The default is None.

    Output:
    coeffs : array_like
            Array of coefficients resulting from the non-linear least squares
            optimization. If the optimization failed, an array of ones is
            returned.'''


    #Check if yData is indeed scalar
    if np.array(yData).ndim > 1:
        raise Exception('yData has more than one dimension (not a scalar).')

    #The curve_fit function does not have a weight argument, but an uncertainty
    #argument sigma. Weighing can be applied by varying sigma; a point with
    #more uncertainty has a relatively small weight. The relation between sigma
    #and weights is sigma=1/sqrt(weights).
    if weights is None:
        sigma = None
    else:
        sigma = 1/np.sqrt(weights)

    #If maxfev is None, use a good guess
    if maxfev is None:
        maxfev = 1000
```

```
    #Usually, p0=None is acceptable for the curve_fit function. However, as a
    #starred input is used for the coefficients, p0 needs to be specified as an
    #array. Hence, if p0 is None, it is specified as an (nCoeffs,) array of ones,
    #mimicking default behaviour.
    if p0 is None:
        p0 = np.ones(nCoeffs)

    #The curve_fit function doesn't accept None as an argument for bounds. Its
    #default value is (-np.inf, np.inf), this is input when the bounds input to
    #the current function are None.
    if bounds is None:
        bounds = (-np.inf, np.inf)

    #Create a fit object, the class of which is defined above this function.
    #This is a hack to pass extra arguments to the fit function while also
    #being able to use an arbitrary number of coefficients. The extra arguments
    #are eqStr and varDict and they are passed to fitClass instead.
    fitObj = fitClass(eqStr, varDict)

    #Call the scipy.optimize.curve_fit function, optimizing the fitobjects
    #fitFunc. The second argument is the initial guess of the dependent
    #variables, no dependent variables are actually passed through curve_fit,
    #so it is just set to one. Dependent variables are passed to fitObj via the
    #varDict.
    try:
        (coeffs), _ = curve_fit(fitObj.fitFunc, 1, yData,
                                p0=p0, sigma=sigma, maxfev=maxfev, bounds=bounds)

    #If the curve_fit function gives an error, first check if either the maximum
    #number of function evaluations was exceeded or the gtol was too small.
    #These errors usually occur for badly conditioned expressions. However, they
    #shouldn't stop the run. Hence, they are printed as errors and the run
    #continues with the regressed coeffs set to all ones. Other errors are raised.
    except RuntimeError as e:
        if 'Number of calls to function has reached maxfev' in str(e)\
           or 'The maximum number of function evaluations is exceeded' in str(e):
            coeffs = np.ones(nCoeffs)
            print(f'maxfev reached by {eqStr}')
        elif 'gtol' in str(e) and 'is too small' in str(e):
            coeffs = np.ones(nCoeffs)
            print(f'Too small gtol for {eqStr}')
        else:
            raise e

    return coeffs



#==========================================================================
#String formatting/evaluation functions

def transposeTensorEqStr(eqStr):
    '''Function to make all T tensors in an equation string transpose and then
    transpose the whole eqStr result. This is to address an issue with shapes of
    features; scalar features need to be multiplied by tensors transposed rather
    than raw tensors.

    Input:
    -eqStr : str
            Tensorial equation string, e.g. "q_nu*T2 + q_gamma*T3".

    Output:
    -eqStrTrans : str
            Same as input, but all tensors transposed, with the whole eqStr
            transposed back again. For the input example, the output would be:
                "(q_nu*T2.T + q_gamma*T3.T).T".'''

    #Loop over each base tensor except the last one (T10(_s)).
    #First replace Ti with Ti.T; in case of mean flow tensors, this would make
    #T2_s into T2.T_s. Fix this by replacing Ti.T_s with Ti_s.T, giving T2_s.T.
    for i in range(1,10):
        eqStr = eqStr.replace(f'T{i}', f'T{i}.T')
        eqStr = eqStr.replace(f'T{i}.T_s', f'T{i}_s.T')
```

```
    #In the above procedure, T10(_s) was also targetted by T1(_s) transpose,
    #giving T1.T0(_s). This is fixed by replacing T1.T0 by T10.T. If mean flow
    #was used, the result would be T10.T_s, which is then replaced by T10_s.T.
    eqStr = eqStr.replace('T1.T0', 'T10.T')
    eqStr = eqStr.replace('T10.T_s', 'T10_s.T')

    #Finally, the whole equation it transposed back to get the correct shape.
    eqStrTrans = f'({eqStr}).T'

    return eqStrTrans


def formatEqStr(eqStrRaw, tensorTarget, targetFunc=None):
    '''Function to format a raw equation string, such that is can be used by
    the fitEqStr function. Most importantly, "{C}" which is used in the raw
    equation string to indicate a coefficient to fit is replaced with coeffs[i],
    where i goes from 0 to the number of "{C}" in the raw equation string.
    Furthermore, if the eqStr is tensorial, it is tranposed to prevent shape
    errors and a targetFunc is applied to get a 1D result.

    Inputs:
    -eqStrRaw : str
            Raw equation string, where coefficients to regress are denoted by
            "{C}".
    -tensorTarget : bool
            Boolean denoting whether eqStrRaw is a tensorial (True) or
            scalar (False) equation.
    -targetFunc : {callable, None}, optional
            Function that takes an equation string as argument and returns a new
            string in which the input string is manipulated. For example, if the
            equation string "q_nu*T2" is inputted (symmetric tensor),
            a targetFunc could be the trace, returning the first, fourth and
            sixth column; targetFunc("q_nu*T2") = "(q_nu*T2)[:,[0,3,5]]".
            This targetFunc will be applied to the raw equation string.

    Outputs:
    -eqStr : str
            Formatted equation string ready to be used by the fitEqStr function.
    -nCoeffs : int
            Number of coefficients to fit in the equation string.'''

    #Initialize the formatted eqStr with eqStrRaw.
    eqStr = eqStrRaw

    #Replace {C} in equation string by coeffs[i], where i goes from zero to the
    #number of coefficients.
    nCoeffs=0
    while '{C}' in eqStr:
        eqStr = eqStr.replace('{C}', f'coeffs[{nCoeffs}]', 1)
        nCoeffs += 1

    #Tranpose the base tensors if the equation is tensorial and transpose the
    #result back (see transposeTensorEqStr for more information).
    if tensorTarget:
        eqStr = transposeTensorEqStr(eqStr)

    #Apply the targetFunc to the eqStr if it is defined.
    if targetFunc is not None:
        eqStr = targetFunc(eqStr)

    return eqStr, nCoeffs


def printEqStrRaw(eqStrRaw, coeffs):
    '''Function to format a raw equation string with its fitted coefficients.

    Inputs:
    -eqStrRaw : str
            Unformatted string of the equation of which the coefficients were
            fitted. Each coefficient that was regressed should be denoted "{C}".
    -coeffs : array_like
            Regressed coefficients of eqStrRaw.
```

```
    Output:
    regressedEqStr : str
            eqStrRaw with the unknown coefficients "{C}" replaced with their
            regressed value.'''

    #Initialize printStr with eqStr raw. Then loop over each coeff and replace
    #the next instance of "{C}" in the eqStr by the regressed coeff.
    regressedEqStr = eqStrRaw
    for coeff in coeffs:
        regressedEqStr = regressedEqStr.replace('{C}', str(coeff), 1)

    return regressedEqStr

def evaluateRegressedEqStr(regressedEqStr, varDict, tensorTarget):
    '''Function to evaluate a regressed equation string, with the coefficients
    already replaced by their regressed value using the printEqStrRaw function.

    Inputs:
    -regressedEqStr : str
            Regressed equation string, meaning its unknown coefficients are
            already replaced with their regressed value.
    -varDict : dict
            Dictionary with variables/features as keys and array as the
            corresponding values. Should at least contain the variables in
            regressedEqStr. For example, if regressedEqStr is a function of
            q_nu and T2, these should be keys of varDict.
    -tensorTarget : bool
            Boolean denoting whether regressedEqStr is a tensorial (True) or
            scalar (False) equation.

    Output:
    -targetFit : array_like,
            regressedEqStr evaluated using the variables in varDict.'''

    #Apply transposing to tensors if regressedEqStr is tensorial, such that
    #shapes are compatible (see the transposeTensorEqStr function for more
    #information).
    if tensorTarget:
        regressedEqStr = transposeTensorEqStr(regressedEqStr)

    #Return the evaluated regressed equation string.
    return eval(regressedEqStr, {'np' : np, **varDict})


#==============================================================================
#Coefficient fitting of a raw equation string

def fitEqStrRaw(eqStrRaw, yData, varDict, tensorTarget, targetFunc=None,
                maxfev=None, weights=None, p0=None, bounds=None):
    '''Function to fit the coefficients in a raw equation string using the
    scipy.optimize.curve_fit function, which uses nonlinear least squares.
    Since this optimization can be unstable, an initial guess and bounds can
    be passed as well.

    Inputs:
    -eqStrRaw : str
            Unformatted string of the equation of which to fit the coefficients.
            Each coefficient to be regressed should be denoted "{C}".
    -yData : array_like,
            Array of target data to be fitted, e.g. kDeficit.
    -varDict : dict
            Dictionary with variables/features as keys and array as the
            corresponding values. Should at least contain the variables in
            eqStrRaw. For example, if eqStr is a function of q_nu and T2, these
            should be keys of varDict.
    -tensorTarget : bool
            Boolean denoting whether eqStrRaw is a tensorial (True) or
            scalar (False) equation.
    -targetFunc : {callable, None}, optional
            Function that takes an equation string as argument and returns a new
            string in which the input string is manipulated. For example, if the
            equation string "q_nu*T2" is inputted (symmetric tensor),
```

```
            a targetFunc could be the trace, returning the first, fourth and
            sixth column; targetFunc("q_nu*T2") = "(q_nu*T2)[:,[0,3,5]]".
            This targetFunc will be applied to the raw equation string.
    -maxfev : {int, None}, optional
            Maximum number of function evaluations to perform within the
            non-linear least squares optimization. If None is specified, the
            default value is used. The default is None.
    -weights : {array_like, None}, optional
            Array of weights by which to weigh the non-linear least squares.
            Should have the same shape as yData. If each point should be weighed
            equally, weights should be None. The default is None.
    -p0 : {array_like, None}, optional
            Initial guess of the coefficients, should have shape (nCoeffs,).
            Alternatively, if None is specified, 1 is used for the initial guess
            of each coefficient. The default is None.
    -bounds : {2-tuple, None}, optional
            Bounds on the coefficients, given as a 2-tuple containing arrays of
            length nCoeffs with the lower and upper bound of each coefficient.
            In order to use no lower/upper bound, use -np.inf/np.inf. If None
            is specified, no bounds are used for all coefficients.
            The default is None.

    Outputs:
    -regressedEqStr : str
            Regressed equation string; inputted eqStrRaw with the coefficients to
            fit replaced by their fitted value.
    -targetFit : array_like
            Target approximation of the regressed eqStr.
    -coeffs : array_like
            Regressed coefficients.'''


    #Get the formatted equation string and number of coefficients.
    eqStr, nCoeffs = formatEqStr(eqStrRaw, tensorTarget, targetFunc)

    #Apply the targetFunc to the yData if it is defined to make yData 1D.
    if targetFunc is not None:
        yData = eval(targetFunc('yData'))

    #Fit the coefficients of the now formatted equation string
    coeffs = fitEqStr(eqStr, yData, varDict, nCoeffs, weights=weights,
                      p0=p0, bounds=bounds, maxfev=maxfev)

    #If there are nan or inf coefficients, the regression also failed and all
    #coefficients are set to 1 to prevent further errors. Also a message is
    #printed with the affected eqStr.
    if np.max(np.isnan(coeffs)) or np.max(np.isinf(coeffs)):
        coeffs = np.ones(nCoeffs)
        print(f'NaN/inf coefficients in {eqStr}')

    #Replace the unknown coefficients in eqStrRaw with their regressed value.
    regressedEqStr = printEqStrRaw(eqStrRaw, coeffs)

    #Find the target predicted by the fitted equation string.
    targetFit = evaluateRegressedEqStr(regressedEqStr, varDict, tensorTarget)

    return regressedEqStr, targetFit, coeffs


#==============================================================================
#Library generation functions

def generateMultiDimList(partialList, loopList, global_degree):
    returnList = []
    if global_degree == 0:
        return [partialList]
    if global_degree == 1:
        for item in loopList:
            returnList.append([*partialList, item])
    else:
        for i, item in enumerate(loopList):
            subList = generateMultiDimList([*partialList, item], loopList[i:],
                                           global_degree - 1)
```

```
            returnList = [*subList, *returnList]

    return returnList



def generateTermLib(bases, features, funcStrings, varDict, target,
                    global_degree):
    '''Function to generate a library of terms, containing each possible
    combination of functions of global degree scalar features multiplied by
    each specified basis. Also adds the variable const, such that lower degree
    functions are regressed as well. The library consists of strings, with
    coefficients to be regressed denoted {C}. As an example, if global degree 2
    is specified, with functions **2 and **3, with feature q_nu and
    bases T1 and T2, the following termLib is generated:
        ["{C}*const*const*T1", "{C}*const*const*T2",
         "{C}*const*q_nu**2*T1", "{C}*const*q_nu**2*T2",
         "{C}*const*q_nu**3*T1", "{C}*const*q_nu**3*T2",
         "{C}*q_nu**2*q_nu**2*T1", "{C}*q_nu**2*q_nu**2*T2",
         "{C}*q_nu**2*q_nu**3*T1", "{C}*q_nu**2*q_nu**3*T2"
         "{C}*q_nu**3*q_nu**3*T1", "{C}*q_nu**3*q_nu**3*T2"]

    Along with this termLib, a p0Lib and boundLib are generated, containing as
    many elements as termLib. The p0Lib contains the initial guesses for each
    term's unknown coefficients and the boundLib contains the bounds for each
    term's unknown coefficients.

    Inputs:
    bases : list
            List of the bases to use, should be strings, e.g. ['T2', 'T3'].
    features : list
            List of features to use, should be strings, e.g. ['q_nu', 'q_gamma'].
    funcStrings : list
            List of funcString objects to use.
    -varDict : dict
            Dictionary with variables/features as keys and array as the
            corresponding values. Should at least contain the specified features
            and bases.
    -target : array_like
            Array with the target that is to be regressed, e.g. kDeficit.
    -global_degree : int
            How many (functions of) features to combine for each term. Can also
            be set to zero to only include bases.

    Outputs:
    -termLib : list
            List of terms, generated from the combinations specified from the
            input.
    -boundLib : list
            List of bounds, same length as termLib. Each bounds is a 2-tuple
            with the array of lower and upper bounds. The number of bounds is
            equal to the number of coefficients to be regressed.
    -p0Lib : list
            List of initial guesses, same length as termLib. Each entry is an
            array with the initial guess for each coefficient to be regressed.'''

    #--------------------------------------------------------------------------
    #First create libraries of 1 dimension (global_degree 1) of only
    #(functions of) features, so no bases yet. For the example in the function
    #description, singleDimTerms would be: ["const", "q_nu**2", "q_nu**3"].

    #Initialize 1D termLib, boundsLib and p0Lib as empty lists.
    singleDimTerms = []
    singleDimBounds = []
    singleDimP0 = []

    #Loop over each specified feature and then each specified functString. Add
    #the expression of each combination to singleDimTerms, each bounds to
    #singleDimBounds and each p0 to singleDimP0.
    for feature in features:
        for funcString in funcStrings:
            singleDimTerms.append(funcString(feature))
            singleDimBounds.append(funcString.get_bounds(\
```

```
                                                    feature=varDict[feature]))
            singleDimP0.append(funcString.get_p0(feature=varDict[feature]))

    #Also add const to the libs. This is not done with any functions, as a
    #function of a constant value is always another constant value. Empty
    #arrays are appended to bounds and p0 such that they are properly intialized
    #as other bounds are later appended to these arrays.
    singleDimTerms.append('const')
    singleDimBounds.append(np.zeros((2, 0)))
    singleDimP0.append(np.zeros(0))


    #-------------------------------------------------------------------------
    #Achieve the specified global degree by taking outer products of the
    #singleDimTerms, singleDimBounds and singleDimP0 lists, global_degree times.
    #This results in expression lists with each combination of the single dim
    #terms/bounds/p0. As an example, the singleDimTerms
    #["const", "q_nu**2", "q_nu**3"] would result in the following multiDimTerms
    #when global degree is 2:
    #["const*const", "const*q_nu**2", "const*q_nu**3", "q_nu**2*q_nu**2,
    # "q_nu**2*q_nu**3", "q_nu**3*q_nu**3"].


    multiDimTerms = ['*'.join(term) for term in\
                     generateMultiDimList([''], singleDimTerms, global_degree)]

    multiDimBounds = [np.concatenate(bounds, axis=-1) for bounds in\
                      generateMultiDimList([np.zeros((2, 0))], singleDimBounds,
                                           global_degree)]

    multiDimP0 = [np.concatenate(p0) for p0 in\
                  generateMultiDimList([np.zeros(0)], singleDimP0,
                                       global_degree)]


    #-------------------------------------------------------------------------
    #Add the bases to generate complete term, bounds and p0 lists.

    #Initialize the final lists with terms, bounds and p0s.
    termLib = []
    boundLib = []
    p0Lib = []

    #Note that up to this point, no coefficient has been added before the
    #expression; only inside functions. This coefficient does not have bounds,
    #so they are specified between -inf and inf. The p0 is found for each basis
    #separately.
    firstTermBounds = np.array([[-np.inf], [np.inf]])

    #Loop over each basis
    for basis in bases:

        #Find the p0 of the coefficient in front of the term. Due to large
        #variation in this coefficient, a reasonable initial guess is
        #constructed from the standard deviation of the target and basis.
        firstTermP0 = np.array([np.std(target)/np.std(varDict[basis])])

        #If the basis has a standard deviation of zero (can happen in 2D
        #cases). Set the p0 of the first term simply to 1 to prevent errors.
        if np.isinf(firstTermP0):
            firstTermP0 = np.ones(1)

        #Loop over each term, bounds and p0 and multiDimTerms, multiTermBounds
        #and multiDimP0. Multipy the term with the basis and add the coefficient
        #in front of the term. Add the bounds and p0 of this first term
        #coefficient to bounds and p0.
        for multiDimTerm, bounds, p0 in\
                              zip(multiDimTerms, multiDimBounds, multiDimP0):
            termLib.append('{C}' + f'{multiDimTerm}*{basis}')
            boundLib.append(np.concatenate([firstTermBounds, bounds], axis=-1))
            p0Lib.append(np.concatenate([firstTermP0, p0]))

    return termLib, boundLib, p0Lib
```

```
#==========================================================================
#Function definitions

class funcString:
    '''Custom class to aid in properly propagating functions in string form,
    including the bounds and p0 of their unknown coefficients. These bounds
    and p0 may depend on the feature in the function, so they can only be
    calculated later. By passing a funcstring object, this is possible.'''

    def __init__(self, funcString, bounds=None, p0=None):
        '''Initialization function of a funcString.

        Inputs:
        funcString : str
                String of the function with possibly a feature and unknown
                coefficient {C}. For example "np.exp({C}*{feature}+{C})".
        bounds : {2-tuple, None}, optional
                Bounds of the coefficient(s) in the funcString. For the example
                funcString, bounds could be ([-np.inf, '-np.std{feature}'],
                                             [0,       'np.std{feature}']).
                The first entry of the tuple are the lower bounds; the second
                entry the upper bounds. Both number as well as strings can be
                used. If strings are used, they are evaluated and {feature} is
                replaced with the feature inputted into the funcString. If None
                is specified, no bounds are used. The default is None.
        p0 : (nCoeffs,) array_like, optional
                Initial guess for each unknown coefficient. Can again be either
                a number or a string to be evaluated for the inputted feature.
                If None is specified, 1 is used for the initial guess.
                The default is None.'''

        #Set object's funcString and nCoeffs property
        self.funcString = funcString
        self.nCoeffs = funcString.count('{C}')

        #Set bounds to -inf to inf for each coefficient if the input is None.
        if bounds is None:
            self.bounds = np.vstack((np.ones(self.nCoeffs)*-np.inf,
                                     np.ones(self.nCoeffs)*np.inf))
        else:
            #If bounds are inputted (not None), check if they are a tuple/list
            #with two elements, both with length nCoeffs. If so, set the
            #object's bounds to the inputted bounds.
            if not isinstance(bounds, list) and not isinstance(bounds, tuple):
                raise Exception(('Inputted bounds are not a list '
                                 'neither a tuple.'))
            if len(bounds) != 2:
                raise Exception('Inputted bounds does not have length two.')
            if len(bounds[0]) != self.nCoeffs and len(bounds[1]) != self.nCoeffs:
                raise Exception(('Inputted bounds does not have length nCoeffs '
                                 'for one of the two internal lists.'))
            self.bounds = bounds


        #Set p0 to ones (equal to the number of coefficients) if no p0 is
        #specified.
        if p0 is None:
            self.p0 = np.ones(self.nCoeffs)

        #If p0 is specified, try to reshape it to a 1D array of length nCoeffs.
        #If this is somehow not possible, throw an error.
        else:
            try:
                self.p0 = np.array(p0).reshape((self.nCoeffs))
            except:
                raise Exception((f'passed p0 to funcstring {funcString} cannot '
                                 'be reshaped to size (self.nCoeffs,).'))

    def __call__(self, feature):
        '''If the object is called with a feature argument, it should return the
        funcString formatted with that feature. For example, if the unformatted
```

```
        funcString is "{feature}**3", it should return "q_nu**3" if called with
        q_nu.

        Input:
        -feature : str
                The feature with which to format the funcString.

        Output:
        -formattedEqStr : str
                The funcString, but with {feature} replaced with the inputted
                feature name.'''

        return self.funcString.replace('{feature}', str(feature))


    def get_p0(self, feature=None):
        '''Function to get the array of initial guesses as numbers, if strings
        with feature dependent functions were inputted, these are evaluated
        using the optional feature argument.

        Input:
        feature : array_like, optional
                Array of feature values.

        Output:
        returnp0 : (nCoeffs,) array_like
                Array of initial guesses for each coefficient in the funcString.
                Always consists of numbers; if the funcString object was
                initialized with strings (e.g. "np.std({feature})"), these are
                evaluated using the inputted array of feature values.'''

        #Initialize the returned p0 as ones
        returnp0 = np.zeros(self.nCoeffs)

        #Loop over each coefficient in the funcString
        for i in range(self.nCoeffs):

            #If the p0 value at this coefficient index is not a string, simply
            #set the returnp0 value at this index to the value in the
            #funcString's p0. Directly continue to the next coefficient index.
            if not isinstance(self.p0[i], str):
                returnp0[i] = self.p0[i]
                continue

            #If no feature is specified, check if the string belonging to the
            #current coefficient contains '{feature}'. If so, throw an error.
            #Else, simply evaluate the string to attain the p0 of the current
            #coefficient as a number. Directly continue to the next coefficient
            #index.
            if feature is None:
                if '{feature}' in self.p0[i]:
                    raise Exception('No feature provided, but needed to get p0')

                returnp0[i] = eval(self.p0[i])
                continue

            #If a feature is specified, evaluate the p0 string for the current
            #coefficient with this feature.
            returnp0[i] = eval(self.p0[i].replace('{feature}', 'feature'))

        return returnp0

    def get_bounds(self, feature=None):
        '''Function to get the bounds arrays as numbers, if strings
        with feature dependent functions were inputted, these are evaluated
        using the optional feature argument.

        Input:
        feature : array_like, optional
                Array of feature values.

        Output:
        returnBounds : (2, nCoeffs) list
```

```
                List of bounds for each coefficient in the funcString. Always
                consists of numbers; if the funcString object was initialized
                with strings as bounds
                (e.g. [["-np.std({feature})"], ["np.std({feature})"]]), these are
                evaluated using the inputted array of feature values.'''

        #Initialize the returnBounds as a list containing two empty lists;
        #the lower and upper bounds. These are appended to within this function
        #to fill both with nCoeffs values.
        returnBounds = [[],[]]

        #Loop over the lower and upper bound.
        for i in range(2):

            #Loop over each coefficient entry in the lower/upper bound list.
            for j in range(len(self.bounds[i])):

                #If the bound is not a string, simply copy the value to
                #returnBounds and continue directly to the next coefficient.
                if not isinstance(self.bounds[i][j], str):
                    returnBounds[i].append(self.bounds[i][j])
                    continue

                #If no feature is specified, but the current bound is a string
                #containing "{feature}", raise an error. Else, evaluate the
                #string and continue directly to the next coefficient.
                if feature is None:
                    if '{feature}' in self.bounds[i][j]:
                        raise Exception(('No feature provided, but needed to '
                                         'get a bound'))

                    returnBounds[i].append(eval(self.bounds[i][j]))
                    continue

                #If the current bound is a string and feature is specified,
                #replace any instances of "{feature}" in the bounds string with
                #the inputted value and evaluate the bound string.
                returnBounds[i].append(eval(self.bounds[i][j].replace(\
                                            '{feature}', 'feature')))

        return returnBounds



def getFuncStringDict():
    '''Function to get a dictionary of possible funcStrings. The keys are names
    of functions and the corresponding values funcString objects of these
    values with appropriate bounds and initial guesses.'''

    #Initialize as empty dict.
    funcStringDict = {}

    #Linear function: y=x
    funcStringDict['linear'] = funcString('{feature}')

    #Tanh function: y=tanh(C*x+C)+C
    funcStringDict['tanh'] = funcString('(np.tanh({C}*{feature} + {C})+{C})',
                                        p0=['1/(50*np.std({feature}))',
                                            'np.std({feature})',
                                            'np.std({feature})'])

    #Normal distribution function: y=exp(-C*(x-C)**2)
    funcStringDict['std'] = funcString('np.exp(-{C}*({feature}-{C})**2)',
                                       p0=['1/(2*np.std({feature})**2)',
                                           'np.mean({feature})'],
                                bounds=[[0, -np.inf],
                                        ['10/(np.std({feature})**2)', np.inf]])

    #Regularized log function: y=log(C*x+1)
    funcStringDict['rlog'] = funcString('np.log({C}*np.abs({feature}) + 1)',
                                        p0='1/np.std({feature})',
                                        bounds=[[0],['10/np.std({feature})']])
```

```
    #Regularized division function: y=x/(C*x^2+1)
    funcStringDict['rdiv'] = funcString('{feature}/({C}*{feature}**2+1)',
                        p0='np.sign(np.average({feature}))/np.std({feature})',
                                    bounds=[['-10/np.std({feature})'],
                                            ['10/np.std({feature})']])

    #Absolute square root function: y=sqrt(abs(x))
    funcStringDict['sqrtabs'] = funcString('np.sqrt(np.abs({feature}))')

    #Regularized square root division function: y=x/(C*abs(x)**1.5+1)
    funcStringDict['rdivsqrt'] = funcString(('{feature}/({C}*'
                                             'np.abs({feature})**1.5+1)'),
                        p0='np.sign(np.average({feature}))/np.std({feature})',
                                    bounds=[['-10/np.std({feature})'],
                                            ['10/np.std({feature})']])

    #Regularized cube root division function: y=x/(C*abs(x)**1.25+1)
    funcStringDict['rdivquart'] = funcString(('{feature}/({C}*'
                                             'np.abs({feature})**1.25+1)'),
                        p0='np.sign(np.average({feature}))/np.std({feature})',
                                    bounds=[['-10/np.std({feature})'],
                                            ['10/np.std({feature})']])

    #Power function: y=x**C
    funcStringDict['pow'] = funcString('np.abs({feature})**{C}',
                                       p0=1.2, bounds=[[1.01],[4]])

    return funcStringDict


#==============================================================================
#Target function definitions


class globFuncs:
    '''Class holding several functions to go from a symmetric tensor (6 columns)
    to a scalar. Functions inside can later be called using globFuncs.func().'''

    #Trace function
    def symmTensI1(S):
        #0  1  2  3  4  5
        #XX XY XZ YY YZ ZZ

        return S[:,0] + S[:,3] + S[:,5]

    #I2 function
    def symmTensI2(S):
        #0  1  2  3  4  5
        #XX XY XZ YY YZ ZZ

        return S[:,0]*S[:,3] + S[:,3]*S[:,5] + S[:,0]*S[:,5] -\
               S[:,1]**2 - S[:,2]**2 - S[:,4]**2

    #Determinant function
    def symmTensI3(S):
        #0  1  2  3  4  5
        #XX XY XZ YY YZ ZZ

        return S[:,0]*S[:,3]*S[:,5] + 2*S[:,1]*S[:,2]*S[:,4] -\
               S[:,1]**2*S[:,5] - S[:,2]**2*S[:,3] - S[:,4]**2*S[:,0]


#Definitions of various targetFuncs; these have eqStr as an argument
#and a new string with a manipulated version of the eqStr as output.
I1TargetFunc = lambda eqStr : f'globFuncs.symmTensI1({eqStr})'
I2TargetFunc = lambda eqStr : f'globFuncs.symmTensI2({eqStr})'
I3TargetFunc = lambda eqStr : f'globFuncs.symmTensI3({eqStr})'
allCompTargetFunc = lambda eqStr : f'{eqStr}.flatten()'


#Generate a dictionary of targetFuncs, with names as keys and a two-element list
#as the corresponding value. The first element of this list is the targetFunc.
#The second element is a string defining what to do with the weights to reshape
```

```
#them similar as the targetFunc.
targetFuncDict = {'I1' : [I1TargetFunc, 'weights'],
                  'I2' : [I2TargetFunc, 'weights'],
                  'I3' : [I3TargetFunc, 'weights'],
                  'all' : [allCompTargetFunc,
                           ('np.tile(weights[:,None], (1, 6)).flatten() '
                            'if weights.ndim==1 else weights.flatten()')]
                  }


#============================================================================
#Plotting functionss


def plotkDeficit(kDeficitFit, kDeficitExact, fignum=1, title=None, weights=None):
    '''Function to scatter the error in kDeficit against the exact kDeficit to
    visualize which parts are fitted well/bad. Scatter points have a size
    dependent on their weight.

    Inputs:
    kDeficitFit : (M,) array_like
            Array of kDeficit values of the fit.
    kDeficitExact : (M,) array_like
            Array of exact kDeficit values.
    fignum : int, optional
            Figure number to plot in, the default is 1.
    title : {int, None}, optional
            Title to put above the figure. If None, no title is used.
            The default is None.
    weights : {(M,) array_like, None}, optional
            Either array of weights with the same shape as kDeficitFit or None.
            If an array is specified, the scatter points size is made weight
            dependent. Else, all scatter points have the same size.
            The default is None.'''


    #If weights=None, give all points the same size, but make the size dependent
    #on the number of points so the plot doesn't get cluttered if there are too
    #many points.
    if weights is None:
        scaledWeights = np.ones(kDeficitFit.shape)/kDeficitFit.size

    #If weights are specified, normalize them.
    else:
        scaledWeights = weights/np.sum(weights)

    plt.figure(num=fignum)
    plt.scatter(kDeficitExact, (kDeficitFit-kDeficitExact),
                c='r', alpha=0.9, s=1500*scaledWeights, zorder=10)
    plt.axhline(c='k', lw=2)
    plt.xlabel(r'kDeficit$_{exact}$ [$m^2/s^3$]')
    plt.ylabel(r'kDeficit$_{fit}$ - kDeficit$_{exact}$ [$m^2/s^3$]')
    plt.title(title)
    plt.tight_layout()

    try:
        os.mkdir('kDeficitOutput')
    except:
        pass
    plt.savefig(f'kDeficitOutput/kDeficit{fignum}')



def plotBijDeltaComponents(bijDeltaFit, bijDeltaExact, fignum=1, title=None,
                           weights=None):
    '''Function to scatter the error in bijDelta against the exact bijDelta to
    visualize which parts are fitted well/bad. Scatter points have a size
    dependent on their weight.

    Inputs:
    bijDeltaFit : (M,6) array_like
            Array of bijDelta values of the fit.
    bijDeltaExact : (M,6) array_like
```

```
            Array of exact bijDelta values.
    fignum : int, optional
            Figure number to plot in, the default is 1.
    title : {int, None}, optional
            Title to put above the figure. If None, no title is used.
            The default is None.
    weights : {(M,) array_like, None}, optional
            Either array of weights with the same number of rows as bijDeltaFit
            or None. If an array is specified, the scatter points size is made
            weight dependent. Else, all scatter points have the same size.
            The default is None.'''


    #If weights=None, give all points the same size, but make the size dependent
    #on the number of points so the plot doesn't get cluttered if there are too
    #many points.
    if weights is None:
        scaledWeights = np.ones(bijDeltaFit.shape[0])/bijDeltaFit.shape[0]

    #If weights are specified, normalize them.
    else:
        scaledWeights = weights/np.sum(weights)

    #Plot in six separate subplots (one for each component).
    fig, axs = plt.subplots(nrows=2, ncols=3, sharex=True, sharey=True)
    tt = axs
    axsf = axs.flatten()
    for i, comp in enumerate(['xx', 'xy', 'xz', 'yy', 'yz', 'zz']):
        axsf[i].scatter(bijDeltaExact[:,i],
                        (bijDeltaFit[:,i] - bijDeltaExact[:,i]),
                        c='r', s=1500*scaledWeights, zorder=10)
        axsf[i].axhline(c='k', lw=2)
        axsf[i].set_title(comp)

    #xlabel
    fig.text(0.5, 0.02, r'$b_{ij,exact}$ [-]', ha='center')

    #ylabel
    fig.text(0.02, 0.5, r'$b_{ij,fit} - b_{ij,exact}$ [-]', va='center',
             rotation='vertical')
    fig.suptitle(title)
    plt.tight_layout()
    plt.subplots_adjust(left=0.126, bottom=0.1)

    try:
        os.mkdir('bijDeltaOutput')
    except:
        pass
    plt.savefig(f'bijDeltaOutput/bijDelta{fignum}')


#==========================================================================
#Creation of the varDict of a case

def createVarDict(flow, inds=None, meanFlowTimeScale=False):
    '''Function to generate the varDict given a FlowFeatures object of a case.

    Inputs:
    flow : FlowFeatures object
            The FlowFeatures object for a case, with the setup_features()
            function already called such that all variables are available in
            flow.vv.
    inds : {array_like, None}, optional
            Either array of indices of the points to add to varDict or None.
            If None is specified, all points in flow.vv are used.
            The default is None.
    meanFlowTimeScale : bool, optional
            Whether to use (True) or not use (False) the mean flow time scale
            nondimensionalization. See sparta.features for more information.
            The default is False.

    Output:
    varDict : dict
```

```
            Dictionary with all bases, features, targets and rest variables as
            keys. The corresponding values are arrays, corresponding to values
            at the inputted inds.'''

    features = [*flow.invariant_names, *flow.scalarfeature_names]
    features.remove('q_T')
    targets = ['kDeficit', 'bijDelta']
    others = ['V', 'k', 'gradU']
    varDict = {} #TODO: make lazy

    #Use the subscript _s if meanFlowTimeScale is true.
    if meanFlowTimeScale:
        allVar = [*features, *PkScalars_s, *baseTensors_s, *targets, *others]
    else:
        allVar = [*features, *PkScalars, *baseTensors, *targets, *others]

    #Copy each specified variable from flow.vv to varDict. If inds are not None,
    #only copy the points at inds.s
    for var in allVar:
        if inds is None:
            varDict[var] = flow.vv[var]
        else:
            varDict[var] = flow.vv[var][inds]

    #Add const as a variable as well.
    varDict['const'] = np.ones(flow.N()) if inds is None else np.ones(inds.size)

    return varDict


#================================================================================
#Main function

def regressModel(targetVar, casePath, global_degree = 1, NTerms = None,
                 allInds = None, NAll = None, NTrain = None, funcs = None,
                 bases = None, fitFeatures = None, targetFuncName = 'all',
                 weights = None, meanFlowTimeScale = False):
    '''
    Regress a model for either kDeficit or bijDelta, based on a certain case.
    Regression is performed using the scipy.optimize.curve_fit function, which
    relies on nonlinear least squares to fit an expression. A library of possible
    expressions with unknown coefficients is built beforehand, these coefficients
    are then regressed using curve_fit. Libraries are only built for single
    terms, meaning multi-term expressions are regressed term-by-term. After a
    new term has been added, the coefficients of the full expression are
    regressed again.

    Inputs:
    -targetVar : {'kDeficit', 'bijDelta'}
            Variable to regress a model for.
    -casePath : str
            Path to the case which data is used to regress the model. This
            case should contain all relevant variables (such as gradU,
            gradk, etc...) to construct the features in its last solution
            directory.
    -global_degree : int, optional
            Maximum degree of each term in terms of number of variables.
            For instance, global_degree 2 uses a maximum of two features in
            each term. global_degree 0 can also be specified such that no
            no features are used in the terms, only scalars multiplied by
            bases. The default is 1.
    -NTerms : {int, None}, optional
            Number of terms to fit in the expression. If set to None, the program
            will keep adding terms indefinitely. The default is None.
    -allInds : {array_like, None}, optional
            Array of indices to use in the regression. This is useful when for
            instance only using a certain region in a case. When None is
            specified, all cells are assumed available for regression. Note that
            a smaller number of cells may still be used for the regression, in
            case NAll or NTrain are not None. The default is None.
    -NAll : {int, None}, optional
            Number of cells to use in the entire regression. This is useful in
            case of memory issues. Note that discovery of the expressions may use
```

```
        a smaller number of cells in case NTrain is not None. However,
        the final regressed expression is reevaluated using NAll cells.
        In case NAll is set to None, all cells in allInds are used for this
        reevaluation. The default is None.
-NTrain : {int, None}, optional
        Number of cells to use in the training. A large library of terms
        is built, each of which has its coefficients fitted by the
        scipy.optimize.curve_fit function. Only NTrain cells are used for
        this 'training' evaluation of each term, which may significantly
        reduce run time. The final expression found is evaluated using
        NAll cells, to give a more accurate fite. If NTrain is specified as
        None, NAll points will also be used for the training. The default
        is None.
-funcs : {list, None}, optional
        List of function names to use in the terms, see the
        getFuncStringDict function for the available names. In the term
        library, each combination of function and feature is used. If
        funcs is set to None, all functions in getFuncStringDict are used.
        The default is None.
-bases : {list, None}, optional
        List of bases to use for the regression (e.g. ['epsilon', 'G1'] for
        a kDeficit model or ['T1', 'T3'] for a bijDelta model). If None is
        specified, all appropriate bases are used. The default is None.
-fitFeatures : {list, None}, optional
        List of features to use in the regression (e.g. ['q_nu', 'W2']).
        If None, then all available features are used. The default is None.
-targetFuncName : {str, None}, optional
        Name of the bijDelta target function, only used when fitting a
        bijDelta model. See targetFuncDict for available target function
        names. The target function specifies how to calculate an objective
        variable from the symmetric bijDelta tensor. If None is specified
        and one is trying to regress bijDelta, an error is thrown.
        The default is 'all' to fit all components of bijDelta.
-weights : {str, None}, optional
        String expression which is evaluated to get the weighing function,
        e.g. 'V*k' to weight by both cell volume and k. When None is
        specified, no weighing is used. The default is None.
-meanFlowTimeScale : bool, optional
        Whether to use the mean flow time scale to nondimensionalize S and W
        or simply omega as was done in the past. Set to True to use gradU
        and False to use omega. The default is False.-
'''

#-----------------------------------------------------------------------------
#Variables derived from targetVar

#Check if the specified target variable is either kDeficit or bijDelta,
#otherwise throw an error. Sigma model fitting is not implemented (yet).
if targetVar not in ['kDeficit', 'bijDelta']:
    raise Exception("targetVar should be either 'kDeficit' or 'bijDelta'.")

#Set the tensorTarget variable; True for bijDelta, false for kDeficit
tensorTarget = True if targetVar == 'bijDelta' else False


#-----------------------------------------------------------------------------
#Get the flowFeatures object for the specified case path and
#set up the features.
flow = FlowFeatures.from_openfoam(casePath)
flow.setup_features(meanFlowTimeScale=meanFlowTimeScale)
N = flow.N() #Total number of cells

#-----------------------------------------------------------------------------
#Generation of varDict with NAll points.

#Generate allInds; either use the specified ones or generate array
#from 0 to N-1.
if allInds is not None:
    allInds = np.array(allInds)
else:
    allInds = np.arange(N)

#If the total number of points is restricted, randomly choose NAll points
```

```
    #from the allInds array generated above.
    if NAll is not None:
        allInds = np.random.choice(allInds, NAll, replace=False)

    #Create varDict with NAll points.
    allVarDict = createVarDict(flow, inds=allInds,
                               meanFlowTimeScale=meanFlowTimeScale)

    #-------------------------------------------------------------------------
    #Generation of varDict with NTrain points.

    #Set trainInds equal to allInds if no NTrain specified. Else, randomly
    #choose NTrain points from allInds.
    trainInds = allInds if NTrain is None else np.random.choice(allInds, NTrain,
                                                                replace=False)

    #Create a varDict with NTrain points.
    trainVarDict = createVarDict(flow, inds=trainInds,
                                 meanFlowTimeScale=meanFlowTimeScale)

    #-------------------------------------------------------------------------
    #Setting of bases, fitFeatures and NTerms if they are None.

    #If no list of bases is specified, simply use all appropriate bases.
    #E.g. T1, ..., T10 if tensorTarget is True and meanFlowTimeScale is False.
    if bases is None:
        if meanFlowTimeScale:
            bases = baseTensors_s if tensorTarget else PkScalars_s
        else:
            bases = baseTensors if tensorTarget else PkScalars

    #If no list of fitFeatures is specified, simply use all available fitFeatures
    #except q_T, as it is simply sqrt(S2).
    if fitFeatures is None:
        fitFeatures = [*flow.invariant_names, *flow.scalarfeature_names]
        fitFeatures.remove('q_T')

    #If no maximum number of terms to regress are specified, set the number to
    #infinite such that it keeps regressing extra terms.
    if NTerms is None:
        NTerms = np.inf

    #-------------------------------------------------------------------------
    #Generating a list of funcStrings and arrays of weights.

    #If list of funcs are specified, use all funcStrings available in
    #funcStringDict. If a list of funcs is specified, map this list to a
    #corresponding list of funcStrings.
    funcStringDict = getFuncStringDict()
    if funcs is None:
        funcStrings = funcStringDict.values()
    else:
        funcStrings = list(map(funcStringDict.__getitem__, funcs))

    #If specified weights are not None, assume its a string to evaluate.
    #Evaluate the weight string with allVarDict and trainVarDict to get the
    #weights at all points and the train poinst respectively. Note that these
    #are later overwritten by the targetFunc modification of the weights.
    if weights is not None:
        allWeights = eval(weights, {'np' : np, **allVarDict})
        trainWeights = eval(weights, {'np' : np, **trainVarDict})

    #If weights are None, the allWeights and trainWeights should also be None.
    else:
        allWeights, trainWeights = None, None

    #TODO: also allow a custom weights list to be passed (with numbers)

    #-------------------------------------------------------------------------
    #Reading in the targetFunc and applying it to the weights.

    #A targetFuncName must be specified if the target is tensorial. If this is
    #not the case, throw an error.
```

```
if targetFuncName == None and tensorTarget is True:
    raise Exception('Please specify a targetFunc when tensorTarget is True.')

#Only use the specified targetFuncName if tensorTarget is True.
#It is assumed no targetFunc is needed for scalar targets.
if tensorTarget:

    #The first element in the targetFuncDict is the actual targetFunc.
    targetFunc = targetFuncDict[targetFuncName][0]

    #The second element in the targetFuncDict is the weightFunc. If weights
    #are not None, apply this weightFunc to allWeights and trainWeights.
    if weights is not None:
        allWeights = eval(targetFuncDict[targetFuncName][1],
                          {'weights' : allWeights, 'np' : np})
        trainWeights = eval(targetFuncDict[targetFuncName][1],
                            {'weights' : trainWeights, 'np' : np})
else:
    targetFunc = None

#-------------------------------------------------------------------------
#Generation of the term, bounds and p0 libraries.

#Get target arrays with NAll points and NTrain points, by extracting the
#targetVar entry from their respective dictionaries.
allTarget = allVarDict[targetVar]
trainTarget = trainVarDict[targetVar]

#Generate the library of terms, bounds and p0s based on the specified bases,
#fitFeatures, funcs and global degree. The varDicts and targets are needed
#for some statistics for initial guesses. NAll points are used for these
#to get the most accurate statistics.
termLib, boundsLib, p0Lib = generateTermLib(bases, fitFeatures, funcStrings,
                                        allVarDict, allTarget, global_degree)

#-------------------------------------------------------------------------
#Initializations before main loop

#Find the number of terms in the termLib. Get an array of printInds to
#print each percentage of the termLib during the main loop. E.g., print
#20% when 20% of the terms in termLib have been regressed.
NExpr = len(termLib)
printInds = np.linspace(0, NExpr, 100).astype(int)

#Initialization of total lists of the expression
totRawExpressionList = []
totRegressedExpressionList = []
totBoundsList = []
totRegressedCoeffsList = []

#-------------------------------------------------------------------------
#Main loop regressing new terms using deficit fitting.

#The first term is fitted as usual; each term in termLib has its coefficients
#regressed given the target. Then, the term that best regresses the target
#is chosen as the first term. Then, the first term is frozen, and the
#second term regresses the deficit between the first term and the target.
#Again, each term in the termLib is considered for this second term, the
#best one is picked. Then, the coefficients of both the second and first
#term are regressed again to optimally fit the target. This deficit fitting
#continues for each new term.

#Initialize the trainDeficit as the target, as the first term should regress
#the target.
trainDeficit = np.array(trainTarget)

t = 1 #Index of the current term
while t <= NTerms:

    #Minimum RMSE initialized as large value, later overwritten if smaller
    #RMSE is found.
    RMSEMin = np.inf
```

```
        #Loop over each term, bounds and p0 in termLib, boundsLib and p0Lib.
        for i, (term, bounds, p0) in enumerate(zip(termLib, boundsLib, p0Lib)):

            #If the current term index in printInds, print the progress as a %.
            if i in printInds:
                print((f'Regression of term {t} at '
                       f'{np.where(printInds==i)[0][0]}%'))

            #Regress the
            regressedEqStr, targetFit, coeffs= \
                           fitEqStrRaw(term, trainDeficit, trainVarDict,
                                       tensorTarget, targetFunc=targetFunc,
                                       weights=trainWeights, p0=p0,
                                       bounds=bounds)

            #Calculate the RSME of the current term
            RMSE = getRMSE(targetFit, trainDeficit, weights=trainWeights,
                           targetFunc=targetFunc)

            #If the term has a lower RMSE than the lowest RMSE found so far,
            #set the best term values to those of the current term. In the end,
            #the best term values are those of the term with the lowest RMSE.
            if RMSE < RMSEMin:
                RMSEMin = RMSE
                termMin = term
                boundsMin = bounds
                coeffsMin = coeffs
                regressedEqStrMin = regressedEqStr

        #Add the minimum term and its bounds to the total lists.
        totRawExpressionList.append(termMin)
        totBoundsList.append(boundsMin)

        #Find the raw total expression so far (e.g. three term expression if
        #t=3). Also find the bounds of this total expression.
        totRawExpression = ' + '.join(totRawExpressionList)
        totBounds = np.concatenate(totBoundsList, axis=-1)

        #Find the regressed coefficients of the total expression.
        totCoeffs = [*totRegressedCoeffsList, *coeffsMin]

        #Find the RMSE for the raw expression given its separately regressed
        #coefficients.
        totEqStr = printEqStrRaw(totRawExpression, totCoeffs)
        allTargetRawFit = evaluateRegressedEqStr(totEqStr, allVarDict,
                                                 tensorTarget)
        RMSERaw = getRMSE(allTargetRawFit, allTarget, weights=allWeights,
                          targetFunc=targetFunc)

        #Refit the total expression to find coefficients regressed for the whole
        #expression.
        refitTotExpression, allTargetRefit, coeffsRefit = \
                                fitEqStrRaw(totRawExpression, allTarget,
                                            allVarDict, tensorTarget,
                                            targetFunc=targetFunc,
                                            weights=allWeights, p0=totCoeffs,
                                            bounds=totBounds, maxfev=1000000)

        #Get the RMSE of the refit of all coefficients in the total expression.
        RMSERefit = getRMSE(allTargetRefit, allTarget, weights=allWeights,
                            targetFunc=targetFunc)

        #For some reason, the refit of all coefficients sometimes gives a higher
        #RSME than isolated coefficients (TODO: find out why). If this happens,
        #simply use the isolated coefficients for the final expression.
        if RMSERaw <= RMSERefit:
            TermRMSE = RMSERaw
            allTargetTerm = allTargetRawFit
            totRegressedCoeffsList = totCoeffs
            totExpression = totEqStr
        else:
            TermRMSE = RMSERefit
            allTargetTerm = allTargetRefit
```

```
            totRegressedCoeffsList = list(coeffsRefit)
            totExpression = refitTotExpression

        #Calcualte the R^2 of the total expression
        TermR2 = getRSquared(allTargetTerm, allTarget, weights=allWeights,
                             targetFunc=targetFunc)

        #Print the total expression and its R^2
        print('\n\n\n')
        print((f"Fitted the following expression with {t} "
               f"term{'' if t==1 else 's'}:"))
        print(totExpression)
        print()
        print((f'It has an R^2 value of {TermR2:.5} and an '
               f'RMSE of {TermRMSE:.6}.'))

        #Calculate the new trainDeficit based on the regressed total expression.
        #The next term should regress the difference between the target
        #prediction of this total expression and the exact target.
        trainDeficit = (allTarget - allTargetTerm)[\
                                        np.where(np.in1d(allInds, trainInds))[0]]

        #Plot the fit error
        if targetVar == 'kDeficit':
            plotkDeficit(allTargetTerm, allTarget, fignum=t,
                         title=f'{t} term model', weights=allWeights)

        elif targetVar == 'bijDelta':
            if targetFuncName == 'all':
                weights=allWeights[::6]
            else:
                weights=allWeights
            plotBijDeltaComponents(allTargetTerm, allTarget, fignum=t,
                                   title=f'{t} term model',
                                   weights=weights)
        t += 1

    return totExpression
```