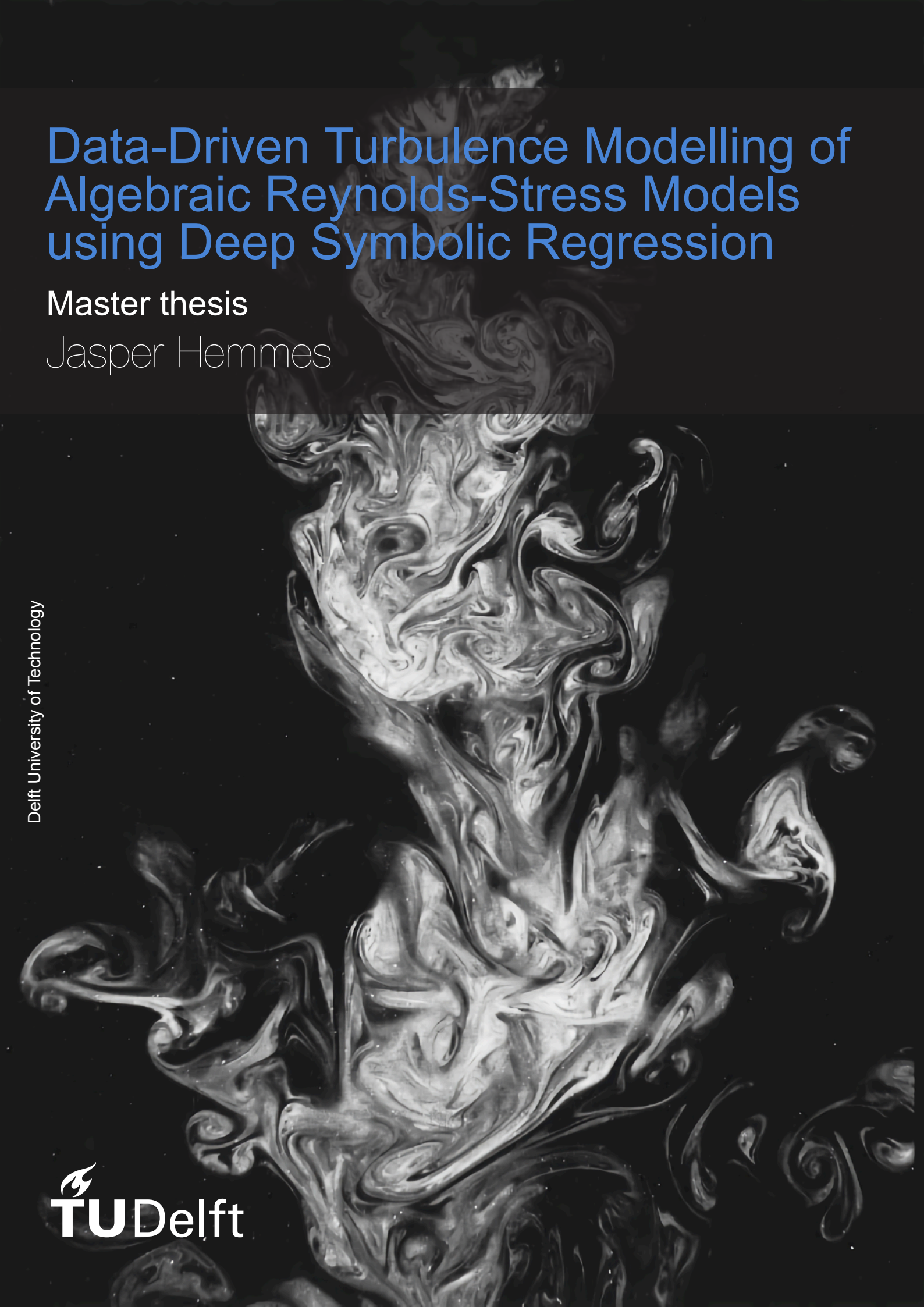
Data-Driven Turbulence Modelling of Algebraic Reynolds-Stress Models using Deep Symbolic Regression

Master thesis

Jasper Hemmes



Delft University of Technology

TU Delft

Delft University of Technology

Data-Driven Turbulence Modelling of Algebraic Reynolds-Stress Models using Deep Symbolic Regression

by

Jasper Hemmes

To obtain the degree of Master of Science in Aerodynamics.
Publicly defended on Wednesday the 18th of May, 2022 at 10:00.

Student number:	4226216	
Project duration:	Januari 2021 - May 2022	
Defence committee:	Dr. R.P. Dwight	TU Delft, supervisor
	Dr. N.A.K. Doan	TU Delft
	Dr. M.B. Zaaijer	TU Delft

Abstract

When simulating fluids the industry standard is Reynolds averaged Navier-Stokes (RANS). However, the results for certain flows are inaccurate. The main source of error in popular RANS turbulence models is the Boussinesq approximation, assuming a linear relationship between the Reynolds stress anisotropy and the mean rate of strain. Experiments show this is simply not correct. Most state of the art research in data driven turbulence modelling is focused on replacing or augmenting this linear relationship.

The research presented in this report implements two corrections, one to the Reynolds stress anisotropy and another to account for the modified production of turbulent kinetic energy by the modified anisotropy tensor. The magnitude of the required corrections is found by comparing RANS simulations to more detailed CFD algorithms such as large eddy simulation and direct numerical simulation.

A state of the art symbolic regressions framework named deep symbolic regression (DSR) is used to find explicit algebraic Reynolds stress models. DSR uses a recurrent neural network to create expressions and is able to find complex expressions that fit the data very well.

The expressions found with DSR are implemented in a custom $k - \omega$ SST turbulence model and validated in CFD. Large improvements over the standard turbulence model are achieved. The results are compared to results of the SpaRTA framework where the same method of corrections is applied. DSR is able to produce better fitting expressions and these result in improved flow fields over the best models found with SpaRTA.

The best model is also tested in a true prediction of a flow at a Reynolds number that is roughly three times as large as values encountered during training. The results are good, showing generalisability of the model outside training conditions.

Contents

List of Figures

List of Symbols

Greek alphabet

Symbol	Definition	Unit
α	Learning rate	-
ϵ	Dissipation rate	m^2/s^3
ε	Flow field error to high fidelity data	-
ε_{sum}	Sum of flow field error ε of all three test cases	-
δ_{ij}	Kronecker delta	-
θ_n	Tensor series invariant	-
λ	Cost function	-
λ_H	Entropy weight	-
μ	Dynamic viscosity	Kg/ms
μ_t	Dynamic eddy viscosity	Kg/ms
ν	Kinematic viscosity	m^2/s
ν_t	Kinematic eddy viscosity	m^2/s
ρ	Density	Kg/m^3
ρ_c	Pearson correlation coefficient	-
σ_y	Standard deviation of data set y	-
τ	Expression traversal	-
τ_{ij}	Viscous stress tensor	N/m^2
ω	Specific rate of dissipation	$1/s$

Roman alphabet

Symbol	Definition	Unit
a_{ij}	Reynolds stress anisotropy	m^2/s^2
b_{ij}	Normalised reynolds stress anisotropy	m^2/s^2
b_{ij}^{Δ}	Reynolds stress anisotropy correction	m^2/s^2
b	Policy gradient objective baseline function	-
a_{max}	Maximum arity of token	-
$\mathbf{b}$	Bias vector for LSTM network	-
C_f	Skin friction coefficient	-
$\mathbf{C}_t$	State vector at time t	-
$\tilde{\mathbf{C}}_t$	Candidate state vector at time t	-
f	Generic machine learning model	-
$\mathbf{f}_t$	Forget gate activation vector at time t	-

Roman alphabet continued

Symbol	Definition	Unit
F_1	Blending function in $k-\omega$ SST model	-
g	Activation function	-
$g^{(m)}$	Scaling function for base tensor m	-
$\mathbf{h}_t$	LSTM cell output at time t	-
H	Entropy	-
$\mathbf{i}_t$	Input gate activation vector at time t	-
J_{risk}	Risk seeking policy gradient objective	-
J_{std}	Standard policy gradient objective	-
k	Turbulent kinetic energy	m^2/s^2
$\mathcal{L}$	Library of tokens	-
l_h	Length of GEP chromosome head	-
l_t	Length of GEP chromosome tail	-
M	DSR or SpaRTA model	-
n_{layers}	Number of layers in neural network	-
n_{units}	Number of LSTM units in neural network layer	-
N	Batch size	-
$\mathbf{o}_t$	Output gate activation vector at time t	-
p	Pressure	N/m^2
$\mathcal{P}$	Turbulent kinetic energy production rate	m^2/s^3
$\mathcal{P}_k^{\Delta}$	Correction to turbulent kinetic energy production rate	m^2/s^3
q	Risk factor	-
r	Expression reward function (INRMSE)	-
Re	Reynolds number	-
R_{ij}	Rotation rate	$1/s$
S_{ij}	Strain rate	$1/s$
t	Time	s
$T^{(m)}$	Base tensor m	-
u_i	Velocity vector	m/s
w	Neural network weights	-
$\mathbf{W}$	Weight matrix for LSTM network	-
X	Training input data set	m
x_i	Spatial coordinate vector (cartesian)	m
x_m	Machine learning input point m in data set X	-
x_t	LSTM cell input at time t	-
y	Training output data set	-
y_{bm}	Benchmark data set target output	-
y_m	Machine learning output point m in data set y	-
$\hat{y}_m$	Predicted machine learning output point m	-

List of Abbreviations

Abbreviation	Definition
ADF	Algorithmic differentiation
ADF	Automatically defined function
BSL	Baseline
C-ADF	Complex automatically defined function
CBFS	Curved backward-facing step
CD	Converging-diverging channel
CFD	Computational fluid dynamics
DNS	Direct numerical simulation
DSR	Deep symbolic regression
EARSM	Explicit algebraic Reynolds stress model
ERC	Ephemeral random constant
ET	Expression tree
FNN	Feedforward neural network
GEP	Gene expression programming
GP	Genetic programming
INRMSE	Inverse normalised root mean squared error
LES	Large-eddy simulation
LEVM	Linear eddy viscosity model
LSTM	Long short-term memory
MLP	Multilayer perceptron
NLEVM	Nonlinear eddy viscosity model
NN	Neural network
NRMSE	Normalised root mean squared error
PH	Periodic hills
RANS	Reynolds averaged Navier-Stokes
ReLU	Rectified linear unit
RF	Random forest
RMS	Root mean square
RNN	Recurrent neural network
SGD	Stochastic gradient descent
SL-GEP	Self learning gene expression programming
SpaRTA	Sparse regression of turbulent stress anisotropy
SST	Shear stress transport
TBDSR	Tensor basis deep symbolic regression
TBNN	Tensor basis neural network
TBRF	Tensor basis random forest

1 Introduction

Simulating reality in a controlled environment has been a field of research since the birth of engineering. In an aerodynamic design process, being able to accurately analyse the flow as early as possible in the design process is of great importance. The fastest and cheapest way to analyse the flow is by computational fluid dynamics (CFD). Within CFD there are several algorithms, some are computationally cheap but give relatively inaccurate results, while others can be much more accurate at the cost of increased computational expense. The research introduced in this document is to improve the accuracy of one of the most widely used CFD algorithms; Reynolds Averaged Navier-Stokes (RANS) by exploring data driven turbulence models.

With advances in machine learning techniques the field of data driven turbulence modelling is gaining popularity. It often involves using more accurate forms of CFD such as large eddy simulation (LES) and direct numerical simulation (DNS). LES and DNS are significantly more accurate than RANS simulations but computationally very expensive. A RANS simulation takes a couple of hours, an LES simulation can take in the order of days and a DNS simulation can take months, therefore LES and DNS are less suitable for everyday fluid simulations. RANS currently is and is expected to remain the industry standard, so improving RANS is an active area of research.

The most widely used turbulence models in RANS are linear eddy viscosity models, relying on a linear relation between the Reynolds stress anisotropy and the mean rate of strain. Simple experiments show that these two quantities are simply not related. Therefore most state of the art data driven turbulence research is focused on improving the Reynolds stress anisotropy.

The most promising method of data driven turbulence modelling is considered to be SpaRTA framework by Schmeltzer [1]. This method introduces two turbulence model corrections, one to the Reynolds stress anisotropy and a second correction to account for the change in production of turbulent kinetic energy by the changed Reynolds stress. The magnitude of these corrections is determined by comparing RANS to LES and DNS. The resulting RANS simulations give an 80% reduction in flow field errors made by standard $k - \omega$ SST turbulence model.

However, the method of symbolic regression applied in SpaRTA is quite restrictive and lacks flexibility. The research presented in this report aims to improve on the SpaRTA research by applying a new symbolic regression method named deep symbolic regression (DSR), by Petersen [2]. DSR uses a recurrent neural network to find expressions and has shown very good results on several symbolic regression benchmarks.

The research question investigated in this report is:

> *Can the use of the deep symbolic regression framework in data driven turbulence modelling improve the accuracy of Reynolds Averaged Navier-Stokes simulations beyond current state-of-the-art methods?*

1.1 Report outline

In this report firstly the basics of CFD are introduced, the need for turbulence models in RANS is detailed in chapter 2. Here also the related work in the field of data driven turbulence modelling and symbolic regression is discussed. Lastly in chapter 2 the research objective is discussed in detail.

Chapter 3 covers the methodology, detailing turbulence model corrections that are applied. Also the working of the DSR framework is explained together with the modifications that were required to make DSR suitable for the turbulence modelling problem. The experimental setup and the hyperparameter search to optimise the working of the neural network is covered in chapter 4.

The application of DSR to find improved turbulence models is presented in chapter 5. The resulting models are implemented in a CFD solver and the performance is discussed and compared to the best models found with the SpaRTA framework. Lastly the conclusions and recommendations for future work are presented in chapter 6.

2

Background and Related Work

2.1 Computational fluid dynamics

When simulating fluids, RANS is the industry standard because of the relatively small computational requirements. Unfortunately RANS suffers from poor accuracy in a range of flows [3], mainly in areas with high turbulence. The difficulty for RANS to predict turbulence is embedded in the way the algorithm is derived; all effects of turbulence are determined by turbulence models. These turbulence models can provide accurate results for a range of flows but fail to capture some fundamental effects of turbulence. The explanations presented throughout this section are inspired by explanations by Pope [4], Tsinober [5] and Hulshoff [6].

2.1.1 Turbulence

Once flows reach a certain Reynolds number it transitions from laminar to turbulent. Such turbulent flows are unsteady, irregular, seemingly random and chaotic. Turbulent flows are often described in terms of the eddy cascade introduced by Richardson [7] and later mathematically refined by Kolmogorov [8]. The eddy cascade is the process in turbulent flows of whirls, also named eddies, breaking down into smaller whirls and so on. These larger and smaller scales of eddies all interact and exist over an extremely wide range. For example in an atmospheric flow the largest scales are hundreds of kilometres in size and the smallest are less than a millimetre. There are an estimated 10^{29} excited degrees of freedom and many of those interact non-linearly [5]. Generally speaking, the process of breaking down into smaller eddies transfers kinetic energy from larger scales to smaller scales, until the eddies become so small that they are dominated by the viscosity and the kinetic energy is dissipated.

Looking at Figure 1 many different scales can be observed, from large eddies almost as large as the diameter of the jet, to the smallest eddies that can be captured by the photo. For engineering applications an engineer might be interested in what the velocity of flow in this jet is. Measuring the axial velocity in a point on the centreline of the jet may result in a graph as in Figure 2. Due to the turbulence there are large fluctuations in the velocity in this point. Therefore a single instantaneous measurement does not give

Figure 1: Turbulent jet at $Re \sim 10^4$, image taken from [9].

much information due to the large variance in velocity. That is why for most engineering applications the mean of the velocity is the quantity of interest. This is exploited in the derivation of the RANS equations, solving the Navier-Stokes equations directly for the mean velocity gives a large reduction in computational requirements.

2.1.2 RANS Governing Equations

The Navier-Stokes equations for incompressible Newtonian viscous flows are as follows; conservation of mass in equation 1, conservation of momentum in equation 2.

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{1}$$

$$\rho \frac{\partial u_i}{\partial t} + \rho \frac{\partial}{\partial x_j}(u_i u_j) = -\frac{\partial p}{\partial x_i} + \frac{\partial \tau_{ij}}{\partial x_j} \tag{2}$$

Here u_i is the velocity vector, ρ is the density, p is the pressure and τ_{ij} is the viscous stress tensor. τ_{ij} is proportional to the rate of strain S_{ij} and the dynamic viscosity μ, the strain rate tensor is calculated by velocity gradients, see below:

$$\tau_{ij} = 2\mu S_{ij} \tag{3}$$

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{4}$$

To find the RANS equations it is assumed the flow can be decomposed in a mean and a fluctuating component, this is called the Reynolds decomposition. Referring back to Figure 2, the mean axial velocity is the dashed line and is just over 2.7 m/s, the fluctuating component causes the variance around this mean. The Reynolds decomposition of velocity is:

$$u = \bar{u} + u' \tag{5}$$

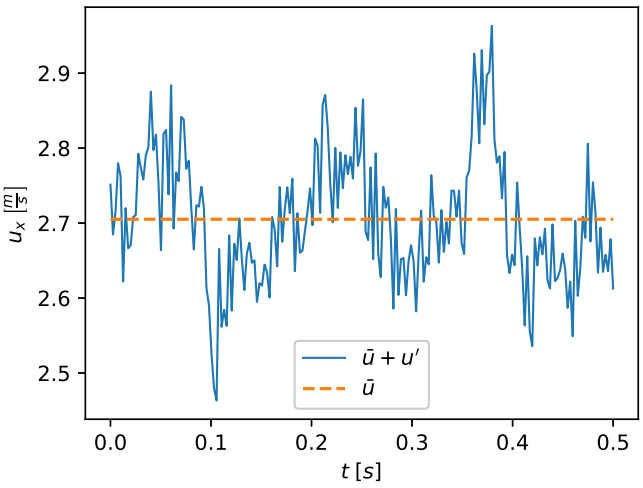


Figure 2: Example variation of axial velocity u_x in the centreline of a turbulent jet

Here $\bar{u}$ is the mean component, u' is the fluctuating component. The average of $\bar{u}$ is simply $\bar{u}$, the average of u' is assumed to be zero. The bar over any variable denotes the mean of that variable. By substituting the decomposition of u in to the Navier-Stokes equations (eqs. 1 and 2) and taking the average of the result, the following is obtained:

$$\frac{\partial \bar{u}_i}{\partial x_i} = 0 \tag{6}$$

$$\rho \frac{\partial \bar{u}_i}{\partial t} + \rho \frac{\partial}{\partial x_j}(\bar{u}_i \bar{u}_j) = -\frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\bar{\tau}_{ij} - \rho \overline{u_i' u_j'}\right) \tag{7}$$

These equations are almost the same as the initial equations for u except for the additional term $\rho \overline{u_i' u_j'}$. This term is named the Reynolds stress because it can be seen to act like an apparent stress, the value of this Reynolds stress is unknown and must be determined. This is the closure problem in RANS, with the Reynolds stress there now are five unknowns in four equations. This is why RANS needs turbulence models, to express the Reynolds stress in terms of known parameters to close the system.

2.1.3 RANS Turbulence Modelling

There is an extremely wide range of available turbulence models. However, the most widely used are one or two equation linear eddy viscosity models (LEVM) and these are expected to remain the most common models in the foreseeable future due to their simplicity [10]. In this section the focus will be on two equation turbulence models since these are the most used in the data driven turbulence research.

Looking at the Reynolds stresses $\overline{u_i' u_j'}$, a distinction is made between isotropic and anisotropic parts of the Reynolds stress. The isotropic stress is $\frac{2}{3} k \delta_{ij}$, the anisotropic stress a_{ij} is then defined as:

$$a_{ij} \equiv \overline{u_i' u_j'} - \frac{2}{3} k \delta_{ij} \tag{8}$$

a_{ij} is also commonly expressed as normalised anisotropy tensor b_{ij}:

$$b_{ij} \equiv \frac{a_{ij}}{2k} = \frac{\overline{u_i' u_j'}}{k} - \frac{1}{3}\delta_{ij} \tag{9}$$

Linear eddy viscosity models assume there is a linear relationship between the Reynolds stress anisotropy b_{ij} and the mean rate of strain tensor $\bar{S}_{ij}$, scaled by the eddy viscosity ν_t. This relation is also named the Boussinesq approximation or the turbulent viscosity hypothesis:

$$b_{ij} = -\frac{\nu_t}{k} \bar{S}_{ij} \tag{10}$$

With b_{ij} the Reynolds stresses can be expressed, however there are still two unknown quantities. The turbulent kinetic energy k and the eddy viscosity ν_t need to be determined to close the system of equations, this is where the aforementioned two equation turbulence model comes in. In these models two transport equations are solved for two turbulence quantities. Most two equation turbulence models have one equation for k and a second equation for an additional variable, the eddy viscosity is determined using k and this second variable. When k and ν_t, can be determined the system of RANS equations is closed and can be solved. The choice of the second variable used next to k differs per model, some of the most popular models are detailed below.

$k - \epsilon$ MODEL

The $k-\epsilon$ model consists of two transport equations, one for the turbulent kinetic energy k and another for the rate of dissipation ϵ, see equations 11 and 12. Note there are many subtle variations of the $k - \epsilon$ model, the one presented here follows Launder and Spalding [11].

$$\frac{\partial k}{\partial t} + \bar{u}_i \frac{\partial k}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_i}\right] + \mathcal{P} - \epsilon \tag{11}$$

$$\frac{\partial \epsilon}{\partial t} + \bar{u}_i \frac{\partial \epsilon}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_t}{\sigma_\epsilon}\right)\frac{\partial \epsilon}{\partial x_i}\right] + C_{\epsilon 1}\frac{\epsilon}{k}\mathcal{P} - C_{\epsilon 2}\frac{\epsilon^2}{k} \tag{12}$$

The production term $\mathcal{P}$ is given by:

$$\mathcal{P} = -2k b_{ij} \bar{S}_{ij} = 2\nu_t \bar{S}_{ij} \bar{S}_{ij} \tag{13}$$

The eddy viscosity ν_t is then calculated as follows:

$$\nu_t = \frac{C_\mu k^2}{\epsilon} \tag{14}$$

Lastly, all the model constants in the equations above are:

$$C_\mu = 0.09, \quad C_{\epsilon 1} = 1.44, \quad C_{\epsilon 2} = 1.92, \quad \sigma_k = 1.0, \quad \sigma_\epsilon = 1.3 \tag{15}$$

The $k-\epsilon$ model is used widely because it is relatively easy to use and computationally inexpensive. The accuracy is acceptable in simple flows where the streamline curvature and pressure gradients are small. However, for more complex flows the model can be quite inaccurate. In boundary layers with strong pressure gradients the performance is poor due to overprediction of turbulent kinetic energy, resulting in a too large wall shear stress and thus incorrect body forces [12].

$k-\omega$ MODEL

Another popular turbulence model is the $k-\omega$ model. This model outperforms $k-\epsilon$ in the viscous near wall region and is better in modelling effects of streamwise pressure gradients. The downsides are that the $k-\omega$ model is inaccurate for non-turbulent boundaries and the outer parts of boundary layers, also the model is very sensitive to the boundary conditions specified for ω [13]. The k and ω equations as introduced by Wilcox [14] are:

$$\frac{\partial k}{\partial t} + \bar{u}_i \frac{\partial k}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\left(\nu + \sigma^* \nu_t\right)\frac{\partial k}{\partial x_i}\right] + \mathcal{P} - \beta^* \omega k \tag{16}$$

$$\frac{\partial \omega}{\partial t} + \bar{u}_i \frac{\partial \omega}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\left(\nu + \sigma \nu_t\right)\frac{\partial \omega}{\partial x_i}\right] + \frac{\gamma\omega}{k}\mathcal{P} - \beta\omega^2 \tag{17}$$

The production term is the same as in the $k-\epsilon$ model, see equation 13. The turbulent viscosity is given by:

$$\nu_t = \gamma^* \frac{k}{\omega} \tag{18}$$

The model constants in the $k-\omega$ model are:

$$\beta = \frac{3}{40}, \quad \beta^* = \frac{9}{100}, \quad \gamma = \frac{5}{9}, \quad \gamma^* = 1, \quad \sigma = \frac{1}{2}, \quad \sigma^* = \frac{1}{2} \tag{19}$$

$k-\omega$ SST MODEL

Menter introduced the $k-\omega$ shear-stress transport (SST) model, aiming to combine the favourable properties of both the $k-\epsilon$ and the $k-\omega$ models. As discussed before, the $k-\omega$ model is superior in the modelling of the viscous near wall region. However, the solutions are inaccurate for the outer part of the boundary layer and are very sensitive to the boundary conditions specified for ω. The $k-\epsilon$ model is not sensitive to such boundary conditions. Combining these models is not as complicated as it seems, start by expressing the ϵ equation (eq. 12) of the $k-\epsilon$ model in terms of ω using $\epsilon = \omega k$:

$$\frac{\partial \omega}{\partial t} + \bar{u}_i \frac{\partial \omega}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_t}{\sigma_\epsilon}\right)\frac{\partial \omega}{\partial x_i}\right] + C_{\epsilon 1}\frac{\omega}{k}\mathcal{P} - C_{\epsilon 2}\omega^2 + 2\frac{C_\mu}{\sigma_\epsilon}\frac{1}{\omega}\frac{\partial \omega}{\partial x_i}\frac{\partial k}{\partial x_i} \tag{20}$$

Comparing this result with the ω equation of the $k-\omega$ model (eq. 17), shows the ϵ equation expressed in ω is identical to the ω equation, except it contains an additional term. This additional term, the last term on the right in equation 20 captures the

difference between both models. To make the comparison between models the value of the constants is ignored for now. In the equations of the $k-\omega$ SST model this last term is multiplied by a blending function F_1:

$$\frac{\partial k}{\partial t} + \bar{u}_i \frac{\partial k}{\partial x_i} = \frac{\partial}{\partial x_i}\left[(\nu + \sigma_k \nu_t)\frac{\partial k}{\partial x_i}\right] + \mathcal{P} - \beta^* \omega k \tag{21}$$

$$\frac{\partial \omega}{\partial t} + \bar{u}_i \frac{\partial \omega}{\partial x_i} = \frac{\partial}{\partial x_i}\left[(\nu + \sigma_\omega \nu_t)\frac{\partial \omega}{\partial x_i}\right] + \gamma \frac{\omega}{k}\mathcal{P} - \beta\omega^2 + 2(1-F_1)\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial \omega}{\partial x_i}\frac{\partial k}{\partial x_i} \tag{22}$$

The blending function F_1 allows to switch between the $k-\omega$ or $k-\epsilon$ models. F_1 equals one in the viscous sublayer and logarithmic region of the boundary layer and gradually becomes zero in the rest of the domain. That means that in the inner part of the boundary layer the last term in eq. 22 is not included and essentially the $k-\omega$ model is used. When F_1 is zero in the rest of the domain the $k-\epsilon$ model is used. This blending function F_1 is dependent on the distance to the closest wall.

Menter noticed that using the model above still resulted in overpredictions of wall shear stress due to too large values of ν_t in the inner boundary layer. That is addressed with another blending function F_2 that limits the the eddy viscosity ν_t. The eddy viscosity is given by:

$$\nu_t = \frac{a_1 k}{\max(a_1\omega, SF_2)} \tag{23}$$

Where $a_1 = 0.31$ and $S = \sqrt{2\bar{S}_{ij}\bar{S}_{ij}}$. To achieve full similarity to both models the employed model coefficients are also affected by the blending function. Let ϕ_1 be a constant used in the $k-\omega$ model and ϕ_2 the corresponding constant from the transformed $k-\epsilon$ model. Then the final constant ϕ used in the model is calculated as follows:

$$\phi = F_1\phi_1 + (1-F_1)\phi_2 \tag{24}$$

The constants and blending functions are presented in appendix C.

2.1.4 LEVM SHORTCOMINGS

The turbulence models introduced above are well suited for most engineering approaches and are among the most widely used models due to their simplicity. However, all of these turbulence models are linear eddy viscosity models which rely on the Boussinesq approximation (equation 10). This approximation is fundamentally incorrect which can be shown with a simple experiment [4].

Consider the experimental setup in Figure 3. At first the air flows through the straight section, where a grid introduces (nearly) isotropic turbulence. In this section there is no mean straining, $\bar{S}_{ij} = 0$. Then the flow enters the contraction, which is designed to produce a uniform axial strain rate. After the contraction there is again a straight section where there is no mean straining. The turbulent viscosity hypothesis predicts

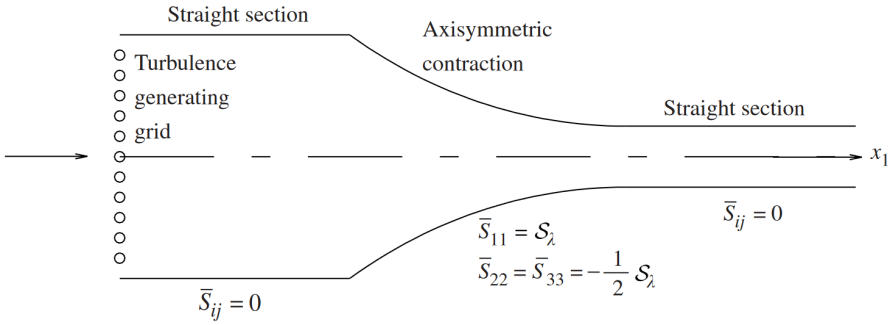


Figure 3: Schematic drawing of system used to study the effect of axisymmetric mean straining of turbulence, image taken from [4].

that the Reynolds stress anisotropy there is zero. However, measurements by Warhaft show that the anisotropy generated in the contraction decays slowly and is non-zero in the second straight section [15]. This shows the Boussinesq approximation is incorrect.

There is also a range of non-linear eddy viscosity turbulence models. Most of such models assume the Reynolds stress anisotropy is a function of both the mean rate of strain $\bar{S}_{ij}$ and the mean rate of rotation $\bar{R}_{ij}$. These models are often very complex and difficult to use while not resulting in significant improvements, their simple counterparts remain the industry standard.

2.1.5 LES and DNS

In the averaging of the Navier-Stokes equations the turbulent fluctuations are removed from the equations, all effects of turbulence on the flow is accounted for using turbulence models. However, turbulence can be modelled directly, albeit at high computational cost.

In direct numerical simulation (DNS) the Navier-Stokes equations are solved for all the scales of turbulent motion. In such simulations no simplifications are made to the Navier-Stokes equations, the result is the most accurate description of flows possible. However, it comes with extremely large computational requirements. Mainly because the smallest turbulent structures must be able to be represented on the mesh. The resulting mesh is so fine that each iteration requires a very large amount of computations. Because the size of the smallest turbulent structures rapidly decreases with increasing Reynolds number this type of simulation is only possible for low Reynolds numbers, even with today's advances in high performance computing.

A more usable type of simulations is large-eddy simulations (LES). In LES the largest scales of turbulent motion are solved directly as in DNS. However, the smallest scales of turbulence are not computed and their effect is accounted for using simple models. By avoiding the computations of the smallest turbulent scales, the computational requirements are far less than for DNS.

The largest scales of turbulence are affected by the geometry under consideration and also have the strongest effect on the rest of the flow. Thus including these in the computations results in a large increase in accuracy. The smallest scales of turbulence are dominated by viscous dissipation of energy, this effect is mainly local and does not strongly affect the rest of the flow. By accounting for this dissipation of energy with a simple model, there is no longer the need for an extremely fine mesh as required by DNS. LES is an elegant solution allowing for accurate analysis of a large range of Reynolds numbers. However, the computational cost is still very large when compared to RANS so RANS remains the standard for analysing flows. Therefore, attempts to increase the accuracy of the RANS is an active area of research.

2.2 Deep learning

The use of machine and deep learning for engineering problems is gaining popularity due to the success and versatility of applications. Also in data driven turbulence modelling research such techniques are becoming more popular. In this section the foundations are given for some of the machine learning techniques that are used and discussed in this report.

2.2.1 Supervised Machine Learning

The most common type of machine learning is called supervised machine learning. In supervised machine learning data is used to find a model or function f relating the input X to output y, that is:

$$f(X) = y \tag{25}$$

This model f could be many things, ranging from a simple polynomial to a complex neural network. Lets consider a simple regression example where input X is used to make predictions of outcome y. To find model f with supervised machine learning a training data set is required. In the training data set for each input data point x_m it is known what the corresponding output y_m is.

$$X = \{x_1, x_2, ..., x_n\} \tag{26}$$

$$y = \{y_1, y_2, ..., y_n\} \tag{27}$$

A plot of this example training set can be seen in Figure 4 where each dot represents an input-output data pair x_m, y_m from the training set.

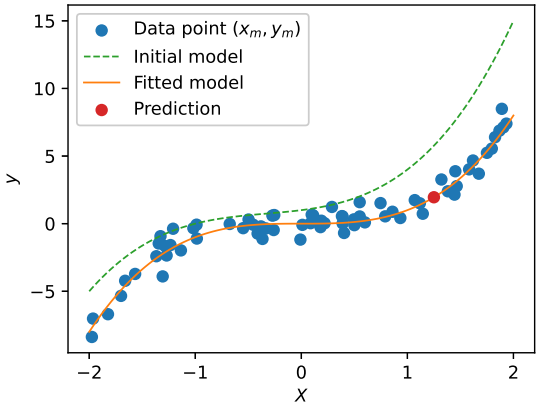


Figure 4: Example data set to illustrate machine learning basics.

The predictions, or output of the model f are denoted $\hat{y}_m$; $f(x_m) = \hat{y}_m$. To evaluate the performance of the model a cost function λ is used, one of the most commonly used is the root mean square (RMS). This cost function compares the predictions $\hat{y}_m$ made by the model f against the known true values of y_m in the training set. The root mean square in this case would be:

$$\lambda_{rms} = \sqrt{\frac{1}{n}\sum_{m=1}^{n}(y_m - \hat{y}_m)^2} \tag{28}$$

The output of the model can be tweaked using the model weights, for the example in Figure 4 this model could be a third order polynomial as in equation 29. The model weights in this case are a, b, c and d. If each weight is initialised to one ($a = b = c = d = 1$) the resulting model is the dashed line in Figure 4, which is clearly not a great representation of the data set. During supervised training of a machine learning model an algorithm minimises the cost function, by changing the model parameters until no further reduction of the cost function is possible.

$$\hat{y}_m = ax_m^3 + bx_m^2 + cx_m + d \tag{29}$$

In the example the minimum of the cost function is found for $a = 1$ and $b = c = d = 0$. The fitted model with these model parameters is also plotted in Figure 4. Now the model can be used to make predictions of y with previously unseen data X.

This example is very simple and fitting a polynomial is a well known method from the field of statistics. However, the method of training a model and using it to make predictions also works with very different models on highly dimensional data, being able to learn incredibly complex representations.

2.2.2 Neural Networks

One of these models that is able to learn complex and non-linear representations is a neural network (NN). There are many types of neural networks, the simplest form is a feedforward neural network (FNN), also named a multilayer perceptron (MLP). Similarly as the example in the previous section, the goal of an FNN is to approximate some function f which maps input data X to output y, $f(X) = y$. In the polynomial example in equation 29, the third order polynomial form of the model was chosen beforehand. For problems in higher dimensions this becomes difficult very quickly, the advantage of a neural network is that no assumptions need to be made regarding the form of function f beforehand.

A NN is constructed of multiple layers, see Figure 5. An input layer where the input data X is presented, one or multiple hidden layers and finally an output layer where the desired output y is returned. The amount of hidden layers is also named the depth of the network, hence the name deep learning. Each of these layers contain a set of nodes, all these nodes are connected to all nodes in the preceding and subsequent layers. All these connections are weighted with weights w. The output of each node $\hat{y}$ is calculated by the weighted sum of the values in all nodes of the previous layer. To extend this system of linear combinations to be able to represent non-linear functions, the weighted sum in each node is transformed by a non-linear activation function g [16]. The most popular activation function is the rectified linear unit (ReLU), but also sigmoid or hyperbolic tangent functions are often used [17]. These neural networks are trained by adjusting all the connection weights w until the desired output is produced, deep learning systems can contain hundreds of millions of such weights [18].

Say an FNN is to be trained to tell whether an image contains a cat or a dog. That is a classification problem with two classes, it is either a cat, or a dog. First a large set of images is required that are known to contain a cat or a dog, usually this labelling requires manual work. In this case a logical output would be two values, one expressing the probability that the image contains a cat, the other that the image contains a dog. The input could be the red, green and blue values for each pixel in the image, so for a 400 by 600 pixel image that is already 720,000 inputs. During the training there is a cost function expressing the errors in the classification, this cost function is minimised by adjusting the network weights w. When a suitable minimum is found, this trained network can now be presented with a new image and if properly trained the network is able tell whether it contains a cat or dog.

The optimisation of the weights w to minimise the cost function is no trivial task, especially if there are millions of weights that affect the solution. That is why most algorithms use stochastic gradient descent (SGD). In SGD the network is presented with only a subset of the training data each iteration, this subset is inputted and passed through the network during what is called the forward pass. At the end of the forward pass the output is compared to the desired output and the cost is determined. Then

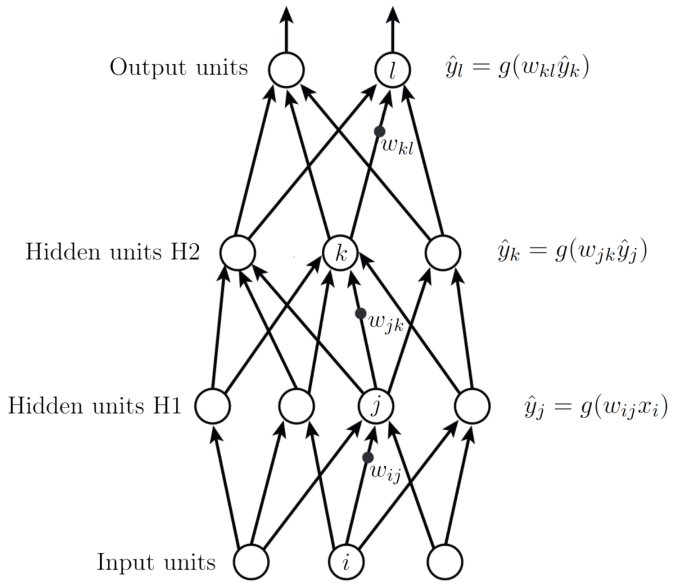


Figure 5: A schematic drawing of an FNN with two hidden layers. Note the Einstein summation notation, the output of node j is the weighted sum of all input nodes x_n, tranformed by activation function g. Image adapted from [18].

the gradient of each individual weight with respect to the output is determined during the backward pass using algorithmic differentiation in reverse mode, also named back-propagation [19]. Knowing what weights have the strongest effect on reducing the cost function significantly speeds up the minimisation of the cost function. The method is called stochastic because each training step only a subset of the training data is used. By using a subset rather than the full set to determine the average cost and gradients the result is a noisy estimate of the gradient [18].

A large drawback is the generalisation of such networks. When trained to see the difference between a cat and dog such networks can perform very well, but present this trained network with a picture of a fox and it will just give probabilities expressing whether it is a cat or dog. So although the classification performance of such networks can be impressive, it rarely outperforms a humans ability to interpret the world. However, in many engineering problems the target is not classification but regression, here a neural network can easily process large amounts of data and discover patterns that a person would never find. In fact it was proven by Hornik that an FNN can approximate any continuous function given the activation function is bounded, continuous and non-constant [20].

Next to the FNN described above there are many different architectures of neural networks that are well suited for certain types of problems. The most notable are convolutional neural networks (CNN), these are the norm when it comes to image processing. This because CNNs efficiently work with multiple arrays as input, such as red green

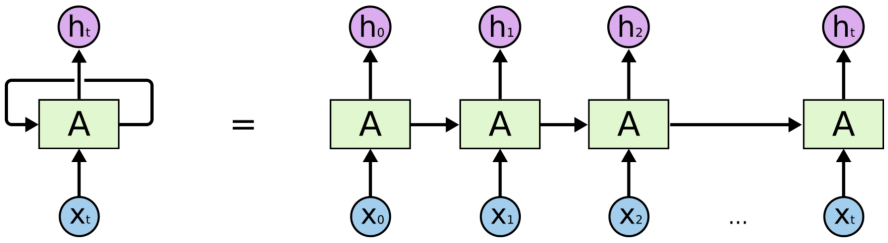


Figure 6: A schematic drawing of a single unit RNN, and the unfolded interpretation of the same single unit network, image taken from [21].

and blue color pixel values of images [18]. Another popular type are recurrent neural networks (RNN). These RNNs are well suited to work with time series or other types of sequential data, for example an expression.

2.2.3 Long short-term memory networks

The DSR framework uses a long short-term memory (LSTM) neural network to generate symbolic expressions. An LSTM network is a popular type of RNN and is introduced below.

RNNs work with sequential data, for example a data set with t time-steps of inputs $x_1, \ldots, x_t$. These inputs are presented one by one to the RNN. The idea of an RNN is that the output of the previous time-step is also used by the network, together with the input of the current time-step. That way the network can learn structures in the sequential data. One way to visualise this is to unfold the RNN, see Figure 6 where on the left a single unit RNN is shown with the recurrent connection and on the right the unfolded visualisation.

The unfolded RNN resembles a FNN with a single hidden layer, but it has crucial differences. An FNN will have unique weights for each connection while the single RNN unit shares the same weights for each time step. The output of the RNN is a function of all previous inputs and outputs, where each output is produced using the same update as during the previous time step due to the sharing of weigths [16].

An RNN is essentially a very deep neural network that maintains a state vector for each time step. This state vector allows to remember the history of inputs presented to the network previously [18]. However, training such a network with backpropagation of gradients proves difficult due to vanishing or exploding gradients. Each time step the gradient either decreases or increases, resulting in incredibly small or large gradients for certain weights, making gradient based training less effective [22].

This problem is addressed by LSTM cells as introduced by Hochreiter and Schmidhuber [23]. Next to the cell output at each time-step h_t, an LSTM cell also maintains the cell state C_t. Instead of directly computing a new cell state C_t each time step, the cell state is maintained through each time step and the LSTM cell computes a correction $\tilde{C}_t$. This allows the network to maintain the cell state and thus input history over a large amount of time steps. Also the gradients of calculating $\tilde{C}_t$ are more convenient and will not explode nor vanish [24].

An LSTM unit contains input, output and forget gates, it maintains memory state C_t and outputs hidden state h_t. See Figure 7 for a schematic drawing. There are many subtle variants of LSTM cells, the one detailed here is as presented by Gers [25], which is the version implemented in TensorFlow, a popular deep learning module for Python. The governing equations are as below.

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{b}) \tag{30}$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_{ii}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i) \tag{31}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_{if}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) \tag{32}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{io}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o) \tag{33}$$

$$\mathbf{C}_t = \mathbf{f}_t \times \mathbf{C}_{t-1} + \mathbf{i}_t \times \tilde{\mathbf{C}}_t \tag{34}$$

$$\mathbf{h}_t = \mathbf{o}_t \times \tanh(\mathbf{C}_t) \tag{35}$$

A time step update of an LSTM cell is as follows. Firstly the input vector $\mathbf{x}_t$ is used together with the hidden state vector of the previous time step to determine the candidate state vector $\tilde{\mathbf{C}}_t$ (eq. 30). $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ are also used to find activation vectors for the input, output and forget gates, $\mathbf{i}_t$, $\mathbf{o}_t$ and $\mathbf{f}_t$ respectively. The sigmoid activation function σ is used to scale these activation vectors between 0 and 1. Now the new memory state vector $\mathbf{C}_t$ can be determined with equation 34. The forget gate activation vector $\mathbf{f}_t$ determines what part of the memory state vector $\mathbf{C}_{t-1}$ of the previous time step is maintained. Similarly the input gate activation vector $\mathbf{i}_t$ determines what part of the candidate state $\tilde{\mathbf{C}}_t$ is stored in the final memory state $\mathbf{C}_t$. Lastly the output $\mathbf{h}_t$ of the cells is calculated as the hyperbolic tangent of the new memory state $\mathbf{C}_t$. The output activation vector $\mathbf{o}_t$ is used to determine what part of $\mathbf{C}_t$ is actually outputted.

The weight matrices $\mathbf{W}$ and bias vectors $\mathbf{b}$ contain the trainable weights, these are adjusted so the network produces the desired output. Depending on the task, the output $\mathbf{h}_t$ can be used directly or is first transformed to calculate the cost function. Finally the cost function is minimised by adjusting the trainable weights in $\mathbf{W}$ and $\mathbf{b}$. Note that the hidden state h_t is visible to other cells in the network, the memory state C_t is only retained inside each individual cell [26].

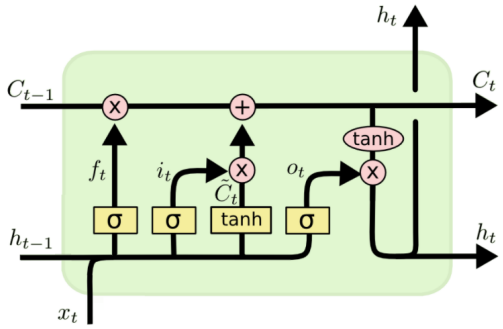


Figure 7: A schematic drawing of a single LSTM unit, image adapted from [21].

2.3 Data Driven Turbulence Modelling

Having introduced the basics of turbulence modelling and machine learning now the relevant research in data driven turbulence modelling is discussed. There is a wide variety of approaches where one of the least drastic is the calibration of model parameters using data. Other researchers completely replace the linear eddy viscosity hypothesis and models an new anisotroppy tensor. Lastly there are methods that leave the Boussinesq approximation in place and only model corrections to the anisotropy tensor from data.

2.3.1 Model Calibration

Edeling et al. [27] applied Bayesian calibration to the coefficients in 5 turbulence models using experimental data of 14 flow cases. The resulting coefficients showed large variance for the different conditions. This shows there is likely not a single set of constants applicable to each condition. Instead they used a sensor to identify the most relevant flow cases in the calibration set, to estimate the model constants with higher weighting for calibration scenarios with similar conditions.

Another method using Bayesian calibration of model coefficients is by Ray et al. [28]. Here the model constants C_μ, $C_{\epsilon 1}$ and $C_{\epsilon 2}$ of the $k-\epsilon$ turbulence model are calibrated using experimental data over a square cylinder. With this experimental data the joint probability density function of these coefficients is estimated, which improves the accuracy in turbulent flow simulations significantly. Also they developed an analytical model to estimate the model constants [29]. The results are good but very specifically tuned to jet in crossflow problems, so this method is unlikely to yield a universal turbulence model.

Furthermore Fabritius used a genetic algorithm to optimise the model constants of the $k-\epsilon$ and $k-\omega$ SST turbulence models [30]. The genetic algorithm was used to minimise the difference between experimental and simulation data. Again drawing the same conclusion that large gains can be made by tuning the constants for a specific flow case, but constants tuned on a certain case translate badly to different flows.

The limited success of these methods in finding a universal set of constants is likely attributed to the assumptions made in the formulation of the used turbulence models. Most importantly that is the linear eddy viscosity hypothesis, this approximation is simply incorrect in a large range of flows. To fundamentally improve turbulence models the focus of recent research has been on departing from this approximation.

2.3.2 Anisotropy Tensor Modelling

Recent works by Ling et al. have focused on modelling the Reynolds stress anisotropy tensor, thus completely removing the flawed Boussinesq approximation. For the modelling they use the Random Forests (RF) algorithm in a supervised learning setting [31]. RF are an ensemble of binary decision trees that categorise input points, after categorising the input points relatively simple functions are used on these sub problems to get a corresponding output. The RF was used to model the anisotropy tensor as a function of local mean velocity, turbulent kinetic energy and more. In total a set of 49 rotational and translation invariant variables was used. The frame invariance is important to ensure the characteristics of turbulence are not affected by flow direction or mesh orientation. The machine learning model was trained using two DNS solutions of a flow in a duct and flow around a wall mounted cube. The resulting model was tested on a jet in crossflow and significantly improved predictions. However, the models were not incorporated in the RANS solver itself but applied after the simulation.

The same research group later used a neural network to calculate the anisotropy tensor, using both a standard FNN and a custom architecture named tensor basis neural network (TBNN) [32]. When deriving nonlinear eddy viscosity models it is often assumed the anisotropy is a function of the local mean rate of strain and rate of rotation. To ensure Galilean invariance a series of 10 base tensors is used as introduced by Pope [33], these base tensors $T^{(m)}$ are functions of the mean rate of strain $\bar{S}_{ij}$ and mean rate of rotation $\bar{R}_{ij}$. With the Cayley-Hamilton theorem Pope showed the most general expression for the anisotropy tensor is:

$$b_{ij} = \sum_{m=1}^{10} g^{(m)} T^{(m)} \tag{36}$$

So the anisotropy can be expressed as the sum of the base tensors $T^{(m)}$. Each base tensor scaled by function $g^{(m)}$, which is a scalar function of the five invariants θ_n. These invariants θ_n are also functions of $\bar{S}_{ij}$ and $\bar{R}_{ij}$, the equations for the base tensors and invariants are given in appendix D. Using this general formulation Galilean invariance of the b_{ij} model is guaranteed.

The TBNN by Ling et al. uses a FNN to predict the functions $g^{(m)}$ which are then combined with their respective base tensors $T^{(m)}$ to find the anisotropy tensor. The found values were implemented in a RANS turbulence model and the resulting simulations showed more accurate velocity predictions. The downside is that directly implementing

a neural network into a turbulence model is very difficult due to the black box nature and non smooth derivatives. Thus the solution was to first run a RANS simulation, use that to predict the Reynolds stress anisotropy and then use those values in a second RANS simulation to update the velocities.

Another method by Kaandorp and Dwight used a tensor basis random forest (TBRF) to predict the anisotropy tensor [34]. This method was inspired by the TBNN and is fairly similar but uses a random forest rather than a neural network to find $g^{(m)}$. The predicted anisotropy tensor was then propagated through the solver in a custom $k-\omega$ model. This tensor required Gaussian smoothing to provide adequate derivatives for the solver. Also several techniques were required to stabilise the solver with the new anisotropy tensor. The advantage of TBRF over the TBNN is that it is less complicated to train a random forest than it is to train a neural network. The resulting flow fields showed a close match of the RANS simulations with DNS or experimental data. Again the downside is that the model for the anisotropy tensor cannot be implemented in the turbulence model directly. b_{ij} needs to be predicted externally and can then be used to improve the flowfield in a second RANS simulation.

2.3.3 Anisotropy Tensor Correction Modelling

More recent works have focussed on using data to model corrections to the Boussinesq approximation instead of completely replacing it. Raissi [35] and Brunton [36] have explored finding analytical models purely from data. Although successful to find expressions for simple systems, finding accurate models for complex systems purely from data proved difficult, if not impossible. Raissi concluded that neither fully relying on existing equations, nor fully relying on machine learning are ideal, a combination using both is likely to be most successful.

That is a great argument to leave the Boussinesq approximation in place and learn corrections to the anisotropy tensor from data. After all the simple LEVMs produce quite good results in many flows. The corrective models for anisotropy b_{ij} with the correction b_{ij}^{Δ} next to the linear eddy viscosity hypothesis look like:

$$b_{ij} = -\frac{\nu_t}{k}\bar{S}_{ij} + b_{ij}^{\Delta} \tag{37}$$

Weatheritt and Sandberg applied gene expression programming (GEP) to find an explicit algebraic Reynolds stress model (EARSM) to model b_{ij}^{Δ} [37]. The benefit of having an EARSM is that the equation can be implemented in the turbulence model and the corrected anisotropy tensor is used every iteration. Another advantage of having a symbolic expression is that to some extent the expression can be interpreted to understand the relations, in contrast to a black-box NN or RF. The stability of the solver with the resulting EARSMs is also taken into consideration, the resulting models for b_{ij} are implemented in a $k-\omega$ SST turbulence model to test convergence [38].

Weatheritt and Sandberg introduced frozen-RANS to find the required correction to the anisotropy tensor. That because research by Thompson et al. showed that using the exact anisotropy tensor from DNS in a RANS simulation gave very poor results [39]. In frozen-RANS, k, u and b_{ij} from LES or DNS are injected into the ω-equation which is then solved to find new values for ω and the eddy viscosity ν_t. Using this ν_t in equation 37 the required correction b_{ij}^{Δ} is found which is then modelled with GEP. The results are promising, showing more accurate velocity profiles and convergence from standard boundary conditions for found EARSMs.

In GEP expressions are encoded as strings of tokens representing an expression tree. For an explanation of expression trees see section 3.2.1, the details of GEP are presented in appendix E. Downsides of the method above are that complicated restrictions needed to be put in place to ensure the GEP method produced valid expressions for tensors. Also GEP is known to scale poorly to larger problems due to the rapidly expanding search space of genes [40, 41]. Also GEP often experiences uncontrollable growth of expression trees without notable improvement op expression fitness [42].

Another method by Schmeltzer et al. uses sparse symbolic regression to find EARSMs for b_{ij}^{Δ} in $k-\omega$ SST models using LES and DNS data, the method is named SpaRTA [1]. Schmeltzer introduced k-corrective frozen-RANS, an extension of the frozen-RANS approach. This because when adding a correction term to the anisotropy tensor, the production of turbulent kinetic energy $\mathcal{P}$ in the k and ω equations is changed, see equations 13, 21 and 22. Thus a second correction term is introduced in the form of a scalar source in the turbulence model transport equations. Both correction terms are found by iteratively solving the k and ω equations with DNS or LES data injected for k, u and b_{ij}.

Implementing these corrections in the $k-\omega$ SST model resulted in significant improvements. Interestingly, in terms of improving the accuracy it was enough to only use the scalar correction to the transport equations, without also using the model for b_{ij}^{Δ}. A shortcoming of this method is the lack of flexibility in finding expressions. A set library containing products of the invariants θ_n is used to find functions $g^{(m)}$. In SpaRTA the functions $g^{(m)}$ can only be linear combinations of the values in this library.

2.4 Symbolic Regression

Symbolic regression is used in the two data-driven turbulence modelling approaches that directly implement the corrections in a custom turbulence model. However, both approaches to symbolic regression have their shortcomings. GEP as used by Weatheritt and Sandberg is known to translate poorly to large data sets. The method of Schmeltzer et al. lacks flexibility in the possible range of expressions. In this section recent advances in the field of symbolic regression are discussed.

GEP has been a popular method for symbolic regression since it's introduction by Ferreira [43] and many methods are proposed to improve the issues in GEP [40]. To increase the flexibility in expression trees the use of automatically defined functions (ADF) is often used [40]. An ADF is essentially a small sub-tree that is used as a terminal token in the chromosomes. Although the use of ADFs is successful in improving this flexibility, the search space grows drastically making it inefficient and less appropriate for large scale problems [40].

Furthermore GEP often needs to evaluate a large number of generations before a solution is reached, this is thought to be because GEP does not learn from the samples that are evaluated [44]. The progression of generations through evolution is more based on luck rather than that the algorithm learns the structure of the problem at hand. To improve the search efficiency Zhong et al. proposed self learning GEP (SL-GEP). This introduced new crossover and mutation genetic operators together with an extension of the ADF representation called complex ADF (C-ADF) [45]. SL-GEP showed improved accuracy and efficiency on several benchmark symbolic regression problems. However, the resulting programs were often overly complex.

A reason why GEP might be less applicable to the turbulence modelling problem is that the GEP evolutionary mechanisms are unsuitable for evolving constants [40]. Considering the most popular turbulence models have many constants, it is expected that finding adequate constants is of high importance for this research. In GEP the most used method to find constants is ephemeral random constant (ERC) [40]. In ERC a placeholder variable for a constant is added to the function set where later this placeholder is replaced by a value from a pre-defined set of random constants. The limitation is that during the search no new constants are added to the set. New constants can only be found when a function operates on two constants. A more successful method is to optimise the constants in each tree to maximise the fitness, as introduced by Kommenda et al. [46]. This method is likely the most appropriate to find fitting constants, albeit computationally intensive.

In contrast to GEP, (deep) neural networks are well known to work well with very large data sets [47]. Udrescu and Tegmark introduced AI-Feynman, a method using neural networks for symbolic regressions, outperforming many state of the art symbolic regression algorithms [48]. Neural networks and several other methods are used to simplify the problem into smaller sub problems. Expressions are fitted to these smaller sub problems and later this is combined to the final expression. The method relies heavily on finding expressions using brute force, i.e. trying all possible combinations of variables to find a fitting expression to a sub problems. Although it is shown that this works on a predefined set of expressions with very little noise, the algorithm is expected to quickly break down in the real world. In the turbulence modelling problem it is unknown whether an exact underlying expression even exists, thus relying on finding expressions with brute force is unlikely to be successful.

A novel method by Petersen named deep symbolic regression (DSR), leverages the representational capacity of neural networks and uses it to generate symbolic expressions [2]. Similar to previously discussed GEP methods, DSR represents models as expression trees. However, instead of progressing the fitness of these trees through evolutionary operators, an LSTM network is trained to generate these expressions. The LSTM is trained with a gradient descent approach based on reinforcement learning. Allowing the LSTM to learn from previously evaluated expressions results in a more efficient search through possible solutions to complex problems. Another advantage over GEP is that constraints to expression trees can be easily be implemented and that neural networks are well known to work easily with very large data-sets.

2.5 Research Objective

The largest source of error in the most common RANS turbulence models is the linear eddy viscosity hypothesis, which assumes a linear relation between the Reynolds stress anisotropy tensor and the mean rate of strain. Simple experiments show that this approximation is simply not true. Thus large parts of the data driven turbulence modelling research has focused on removing or correcting this approximation.

Completely replacing the linear eddy viscosity approximation is a difficult task using data alone, also this might not be the most effective approach since the approximation gives good results in a range of flows. Modelling a correction to the anisotropy tensor seems more sensible and gives good results. The most promising method is SpaRTA by Schmeltzer [1], who used sparse symbolic regression to find explicit algebraic equations for two correction terms to the $k-\omega$ SST model. One term correcting the anisotropy tensor and another correcting the transport equations of the turbulence model to account for the modified anisotropy. These correction terms were found by comparing RANS simulations to more accurate direct numerical simulations and large eddy simulations

A downside of the approach by Schmeltzer is that the sparse symbolic regression used to find expressions models lacks flexibility. GEP is a more widely used method for symbolic regression. However, research in the past decades has focused on improving GEP but it is still known to work poorly on complex problems and large data sets. Neural networks are known to work extremely well with large data sets and complex problems. The deep symbolic regression (DSR) framework introduced by Petersen uses a long short-term memory (LSTM) network to generate expressions, using the great representational capacity of neural networks to more efficiently scan the search space [2].

To advance the data driven turbulence modelling field the research proposed in this document will combine the methods of Schmeltzer and Petersen. Using the same turbulence modelling approach as Schmeltzer, but improving the symbolic regression by using DSR introduced by Petersen. The research question is:

> *Can the use of the deep symbolic regression framework in data driven turbulence modelling improve the accuracy of Reynolds Averaged Navier-Stokes simulations beyond current state-of-the-art methods?*

This research question contains two main topics of research, namely the deep symbolic regression framework and data driven turbulence modelling. The main focus will be on the performance of deep symbolic regression, where the results are tested in a data driven turbulence problem. To evaluate the performance of DSR the resulting models are compared to the models found in the SpaRTA research. Several sub-questions arise that will help answer the research question:

- Can DSR produce more complex tensor and scalar models than SpaRTA?
- Does a more complex model fit the data set better than a simple model?
- Does a more complex model result in mean velocity fields or RANS simulations closer to LES or DNS compared to more simple models?
- Do the DSR models generalise better to different flows compared to SpaRTA?

3

Methodology

3.1 Turbulence model corrections

The turbulence model under consideration is the $k-\omega$ SST model, as introduced in section 2.1.3. Two correction terms will be applied as introduced in SpaRTA by Schmeltzer [1]. The first correction b_{ij}^{Δ} is to the anisotropy tensor to augment the linear eddy viscosity hypothesis:

$$b_{ij} = -\frac{\nu_t}{k}\bar{S}_{ij} + b_{ij}^{\Delta} \tag{38}$$

Changing the anisotropy tensor alters the production of turbulent kinetic energy, thus a second correction $\mathcal{P}_k^{\Delta}$ is introduced to account for this change. The k and ω equations of this augmented $k-\omega$ SST model are:

$$\frac{\partial k}{\partial t} + \bar{u}_i\frac{\partial k}{\partial x_i} = \frac{\partial}{\partial x_i}\left[(\nu + \sigma_k\nu_t)\frac{\partial k}{\partial x_i}\right] + \mathcal{P} + \mathcal{P}_k^{\Delta} - \beta^*\omega k \tag{39}$$

$$\frac{\partial \omega}{\partial t} + \bar{u}_i\frac{\partial \omega}{\partial x_i} = \frac{\partial}{\partial x_i}\left[(\nu + \sigma_\omega\nu_t)\frac{\partial \omega}{\partial x_i}\right] + \gamma\frac{\omega}{k}(\mathcal{P} + \mathcal{P}_k^{\Delta}) - \beta\omega^2 + 2(1-F_1)\frac{\sigma_{\omega 2}}{\omega}\frac{\partial \omega}{\partial x_i}\frac{\partial k}{\partial x_i} \tag{40}$$

The magnitude of b_{ij}^{Δ} and $\mathcal{P}_k^{\Delta}$ are determined with k-corrective frozen RANS: b_{ij}, k and u from LES or DNS are used in equations 39 and 40 to find values of ω and $\mathcal{P}_k^{\Delta}$. Using the value of ν_t resulting from this frozen approach and the LES or DNS data the required correction b_{ij}^{Δ} is determined.

3.1.1 Variables used to model corrections

The corrections found with k-corrective frozen RANS will be modelled using DSR. The resulting models are used in the augmented $k-\omega$ SST model in equations 39 and 40 to improve flow field predictions over the standard model. The equations for b_{ij}^{Δ} and $\mathcal{P}_k^{\Delta}$ are evaluated in each cell of the mesh, so the variables used to model these equations must be available in each cell.

For b_{ij}^{Δ} the used variables are the Pope base tensor series and invariants as introduced in section 2.3.2. Using this base tensor series ensures the resulting models are frame

invariant. Because in this work only two dimensional flows are under consideration, only the first four base tensors and the first two invariants are used.

For $\mathcal{P}_k^\Delta$ also the turbulent kinetic energy and the mean rate of strain are added to the variable set. This is because $\mathcal{P}_k^\Delta$ is modelled as a source term in the ω equation, so the same variables should be available. To transform the base tensors to the required scalar correction, the tensors are pre-multiplied with the mean rate of strain. Adding more variables would be possible, but to make a fair comparison against SpaRTA the sets below are used.

$$X_{b_{ij}^\Delta} = \{T^{(1)}, T^{(2)}, T^{(3)}, T^{(4)}, \theta_1, \theta_2\} \tag{41}$$

$$X_{\mathcal{P}_k^\Delta} = \{T^{(1)}\bar{S}_{ij}, T^{(2)}\bar{S}_{ij}, T^{(3)}\bar{S}_{ij}, T^{(4)}\bar{S}_{ij}, \theta_1, \theta_2, k, \bar{S}_{ij}\} \tag{42}$$

The basis tensors and invariants are calculated as follows. To make the tensors dimensionless they are all divided by ω.

$$T^{(1)} = \bar{S}_{ij} \qquad T^{(2)} = \bar{S}_{ij}\bar{R}_{ij} - \bar{R}_{ij}\bar{S}_{ij}$$

$$T^{(3)} = \bar{S}_{ij}^2 - \tfrac{1}{3}I \cdot \text{trace}(\bar{S}_{ij}^2) \qquad T^{(4)} = \bar{R}_{ij}^2 - \tfrac{1}{3}I \cdot \text{trace}(\bar{R}_{ij}^2)$$

$$\theta_1 = \text{trace}(\bar{S}_{ij}^2) \qquad \theta_2 = \text{trace}(\bar{R}_{ij}^2)$$

3.2 Deep Symbolic Regression

DSR was introduced by Petersen [2] but needs significant modifications to be applied to the turbulence modelling problem. Some of these modifications are to speed up the code to be able to work with large turbulence modelling data sets. Other modifications are there to enforce standard form of expressions.

In short, DSR uses an LSTM neural network to sample expressions. The LSTM outputs a probability distribution containing probabilities to select a token from the library. These distributions are used to sample a sequence of tokens that will form an expression tree.

Some expressions will contain one or multiple constant tokens. When the sampled expression is complete these constants are optimised to maximise the reward of the expression. Each training iteration a batch of 1000 expressions is sampled and evaluated. The best 50 expressions of each batch are used to train the LSTM using the risk-seeking policy gradient objective.

All these concepts are explained in detail thoughout this section. The pseudo code of the DSR algorithm is given below. This pseudo code is cited from the original DSR paper [2], but some modifications to the code were made for this research.

Algorithm 1: Deep symbolic regression with risk seeking policy gradient

input: learning rate α, entropy weight λ_h, risk factor q, batch size N, iteration limit l_i
ouput: best fitting expression τ^*

1. Initialise LSTM with parameters $\mathbf{W}$, defining distribution over expressions $p(\cdot|\mathbf{W})$
2. **repeat**
3. $\quad \mathcal{T} \leftarrow \{\tau^{(i)} \sim p(\cdot|\mathbf{W})\}_{i=1}^{N}$ — Sample N expressions
8. $\quad \mathcal{T} \leftarrow \{\text{OptimiseConstants}(\tau^{(i)}, r, l_i)\}_{i=1}^{N}$ — Optimise constants with iteration limit
5. $\quad \mathcal{R} \leftarrow \{r(\tau^{(i)}\}_{i=1}^{N}$ — Compute rewards
6. $\quad r_q \leftarrow (1-q)$-quantile of $\mathcal{R}$ — Compute reward threshold
7. $\quad \mathcal{T} \leftarrow \{\tau^{(i)} : r(\tau^{(i)}) \geq r_q\}$ — Select subset of expressions above threshold
8. $\quad \mathcal{T} \leftarrow \{\text{OptimiseConstants}(\tau^{(i)}, r, -)\}$ — Optimise subset constants without limit
9. $\quad \mathcal{R} \leftarrow \{r(\tau^{(i)}) : r(\tau^{(i)}) \geq r_q\}$ — Update subset rewards
10. $\quad \hat{g}_1 \leftarrow \text{Mean}((\mathcal{R} - b)\nabla_W \log p(\mathcal{T}|\mathbf{W}))$ — Compute risk-seeking policy gradient
11. $\quad \hat{g}_2 \leftarrow \text{Mean}(\lambda_H \nabla_W H(\mathcal{T}|\mathbf{W}))$ — Compute entropy gradient
12. $\quad \mathbf{W} \leftarrow \mathbf{W} + \alpha(\hat{g}_1 + \hat{g}_2)$ — Apply gradients
13. $\quad$ **if** $\max \mathcal{R} > r(\tau^*)$ **then** $\tau^* \leftarrow \tau^{\arg\max \mathcal{R}}$ — Update best expression
14. **return** τ^*

3.2.1 Representing an expression

In DSR expressions are represented as expression trees (ET), these trees are constructed out of tokens. The tokens represent mathematical operations or variables and are selected from a set library $\mathcal{L}$. An example library could be $\mathcal{L} = \{\times, \div, +, -, \exp, \log, x, y, \text{const}\}$. Here the const token represents a to be determined constant. These constants are later optimised to best fit the problem at hand, see section 3.2.4 for details on the constant optimisation. Using the library $\mathcal{L}$ an example ET can be constructed, see Figure 8.

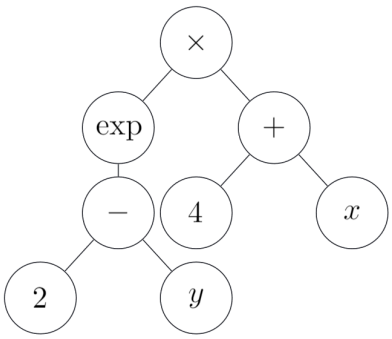


Figure 8: Example expression tree representing $\exp(2-y) \times (4+x)$, image adapted from [37].

To build the expression from the example ET in Figure 8, start at the bottom and work to the top node. The two bottom nodes are 2 and y, these simply represent the value 2 and variable y. Their parent is the $-$ token, this token represents the minus operation

and subtracts the left child from the right child. So the value at the $-$ node is $2-y$. The parent of this token is exp, so the value there is the exponent of its child value, which is $\exp(2-y)$. Finally at the top node the values of the left and right child branches are multiplied by the $\times$ token, the full expression is $\exp(2-y) \times (4+x)$.

The shape of such trees is flexible and depends on the chosen tokens because different tokens have a different amount of children. For example, the multiplication operation has two children, the two values being multiplied. The exponent token only has one child which is the value it operates on. Variables or constants are terminal tokens with no children. The number of children each token has is also called the arity.

In the turbulence modelling problem it is expected that trigonometric functions are not required to find a suitable model because these operations are simply never used in common turbulence models. The used operations and variables in the library of DSR are: $\mathcal{L} = \{\times, \div, +, -, \exp, \log, \text{const}, X\}$. Here X is the input data set for either $\mathcal{P}_k^\Delta$ or b_{ij}^Δ containing the variables as detailed in section 3.1.1.

3.2.2 Sampling an expression

In DSR, expressions are sampled one token at a time until a string of fixed length is sampled. Then, using the arity of each token and depth first traversal, an expression tree is constructed. Depth first traversal means visiting each node depth first, from left to right. Consider the following string of 10 sampled tokens. Note the `e` token represents the exponent function:

```
0123456789
*e-2y+4x-x
```
(43)

Using depth first traversal the sample above constructs the example expression tree in Figure 8. Note that the last two tokens in the sample are not used in the expression tree, because there are no operators that have free terminals. Using a sample of fixed length can result in not all tokens being required. Similarly it is also possible that more tokens are required to complete the tree when there are still open terminals. If that is the case the open terminals are filled with constants.

The goal of DSR is to find the best fitting expressions f relating the input data X to the output y, $f(X) = y$. The idea is that the LSTM network learns the structure of good fitting expressions and samples accordingly. Each time step the LSTM with weights $\mathbf{W}$ outputs a discrete probability distribution ψ. ψ contains the probability of each token in the library $\mathcal{L}$ to be selected. The next token is drawn from this probability distribution ψ. To help the LSTM understand the structure of the previously drawn tokens, the network is presented with observations at each step during sampling. These observations represent what the parent and sibling tokens are of the token that will be sampled next.

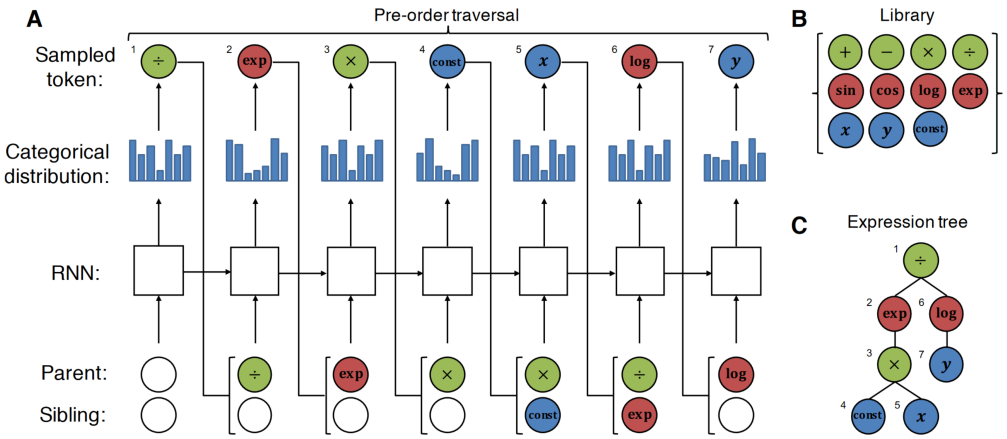


Figure 9: Visualisation of sampling procedure in DSR. The used library of tokens is shown in B. A shows 7 sampling time steps and the corresponding observations. C depicts the ET for the sampled tokens in A. Image adapted from [2].

For example, consider the token in position 4 in the example traversal in eq. 43. The sample drawn is a `y`, but before this token is chosen, the observations are passed as input to the LSTM. In this case, the observations are the parent `"-"` and the sibling `"2"`. It could be that the LSTM has learned that the combination $2 - y$ often produces well fitting expressions. Then the LSTM will assign a high probability of selecting `y`, given the observations `(-, 2)`. If at the time of sampling the parent or sibling of the token to be sampled is unknown, the observation is empty. The sampling process is visualised in Figure 9.

In other words, the probability vector ψ that is outputted each sample step defines the probabilities of selecting each token τ_i from the library $\mathcal{L}$, given the previously selected tokens $\tau_{1:(i-1)}$. See below:

$$p\left(\tau_i|\tau_{1:(i-1)};\mathbf{W}\right) = \psi^{(i)}_{\mathcal{L}(\tau_i)} \tag{44}$$

The likelihood of the entire sampled expression is simply the product of the probabilities of selecting each token in the traversal:

$$p\left(\tau|\mathbf{W}\right) = \prod_{i=1}^{|\tau|} p\left(\tau_i|\tau_{1:(i-1)};\mathbf{W}\right) = \prod_{i=1}^{|\tau|} \psi^{(i)}_{\mathcal{L}(\tau_i)} \tag{45}$$

To avoid unnecessary calculations, some constraints are introduced. Firstly the child of an operator may not be the inverse of the parent, so $\exp(\log(x))$ is not allowed as that is the same as x. Secondly, it is prevented that all children of an operator are constants, since the result would be another constant. Lastly, the expressions are limited to a specified length. These constraints are checked each sample step. If a certain token is not allowed to be selected, the probability of that token being selected is adjusted to zero.

3.2.3 Evaluating an expression

The goal is to find expressions that best fit the data set. Once samples are taken, the reward functions of these candidate expressions are evaluated. A candidate expression f gives a predicted outcome $\hat{y}$ calculated from the inputs, $f(X) = \hat{y}$. These values are compared to the known target values y by the normalised root mean squared error (NRMSE). NRMSE is similar to the root mean squared error introduced in section 2.2.1 but normalised by the standard deviation σ_y of the target set y:

$$r_{nrmse} = \frac{1}{\sigma_y}\sqrt{\frac{1}{n}\sum_{m=1}^{n}(y_m - \hat{y}_m)^2} \tag{46}$$

The NRMSE is passed though a squashing function to give a bounded reward function, which is named the inverse normalised root mean squared error (INRMSE). This is the reward function used to evaluate the fitness of expressions, here τ is the traversal:

$$r(\tau) = \frac{1}{1 + r_{nrmse}} \tag{47}$$

In case of a perfect match between y and $\hat{y}$; $r = 1$. When the match between y and $\hat{y}$ is infinitely bad; $r = 0$. That the reward function is bounded is important for the gradient based optimisation. A standard mean squared error metric can assume extremely large values when the fit is bad, such large values then dominate the gradient which has a negative effect on training [2]. Therefore, the squashing function above is used.

3.2.4 Optimising expression constants

Section 3.2.1 introduces the library $\mathcal{L}$ containing tokens which build an expression. One of those tokens is the const token, which is a to be determined constant. When this token is selected during sampling, it is simply a placeholder. After completion of the sample all constants in the expression are optimised to maximise the reward function r, thus finding the best fit by tweaking the constants. The use of many constants is expected to be crucial to the turbulence modelling problem, because the most popular turbulence models all contain multiple constants.

This constant optimisation is a computationally expensive task, so two measures are taken to reduce the time required to complete a run. First, automatic differentiation in reverse mode is implemented to efficiently calculate the gradients. Second, the number of optimisation iterations is dynamically limited. This means that all expressions initially have an iteration limit, but for the best expressions, this limit is removed to explore the potential fit of these expressions.

Algorithmic Differentiation
For non-linear optimisation, quasi-Newton methods are often used due to their fast convergence. Such methods only require the gradients of the varying parameters with

respect of the objective function. By keeping track of the changes in gradients a model of the objective function is constructed which provides superlinear convergence [49]. The most popular quasi-Newton method is BFGS named after it's inventors Broyden, Fletcher, Goldfarb and Shanno. The BFGS method is used for the constant optimisation in expressions.

Without providing the gradient to the optimisation algorithm, the algorithm will approximate the gradients locally by perturbing each individual constant and evaluating the effect on the reward function. In an expression with five constants this means the expression must be evaluated five times before all gradients are known and the constants can be updated. An improvement can be made by providing the gradients of each constant to the algorithm so that it does not need to approximate the gradients locally.

However, then the gradients must be evaluated differently. This done by implementing algorithmic differentiation (AD) in reverse mode, also named adjoint mode or backpropagation. This method is highly efficient to determine the derivatives of a large number of variables with respect to a single dependent variable. In fact it is possible to determine the derivatives of millions of variables with respect to r with one to four times the floating point operations required to determine r itself [19].

AD in reverse mode works by breaking down expression into elementary mathematical operations. For these elementary operations the derivatives are straightforward and known. By applying the chain rule backward, the derivatives of a complex expression are determined. Consider an example with the mean squared error function:

$$r_{mse} = \frac{1}{n} \sum_{m=1}^{n} (y_m - \hat{y}_m)^2 \tag{48}$$

The corresponding expression tree can be seen in Figure 10. The breakdown in elementary operations can be seen in equations 49 to 56, note that y and $\hat{y}$ are vectors containing n values.

$$v_1 = y \tag{49}$$
$$v_2 = \hat{y} \tag{50}$$
$$v_3 = v_1 - v_2 \tag{51}$$
$$v_4 = v_3^2 \tag{52}$$
$$v_5 = \text{sum}(v_4) \tag{53}$$
$$v_6 = n \tag{54}$$
$$v_7 = \frac{v_5}{v_6} \tag{55}$$
$$r_{mse} = v_7 \tag{56}$$

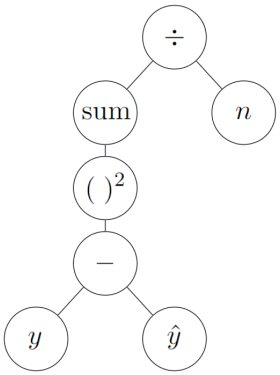


Figure 10: Expression tree of MSE function.

In the forward pass, v_1 to v_7 are evaluated, with v_7 the final target r_{mse} is known. During the backward pass the derivative of each node with respect to r_{mse} is determined, working back from v_7 to v_1. Start with the derivative of r_{mse} with respect to v_7:

$$\frac{\partial r_{mse}}{\partial v_7} = \frac{\partial r_{mse}}{\partial r_{mse}} = 1 \tag{57}$$

v_7 is a function of v_5 and v_6, that means that when the derivative with respect to v_7 is known, the derivatives of v_5 and v_6 can be determined. See the steps taken to find $\partial r_{mse}/\partial v_5$ below:

$$\frac{\partial r_{mse}}{\partial v_5} = \frac{\partial r_{mse}}{\partial v_5}\frac{\partial v_7}{\partial v_7} = \frac{\partial r_{mse}}{\partial v_7}\frac{\partial v_7}{\partial v_5} = \frac{\partial r_{mse}}{\partial v_7}\frac{1}{v_6} \tag{58}$$

By rewriting the equation above the right-hand side now contains known variables, so $\partial r_{mse}/\partial v_5$ can be determined. The same is applied to each node to find the derivatives of r_{mse} with respect to each node, see below:

$$\frac{\partial r_{mse}}{\partial v_6} = \frac{\partial r_{mse}}{\partial v_7}\frac{\partial v_7}{\partial v_6} = \frac{\partial r_{mse}}{\partial v_7} \cdot -\frac{v_7}{v_6} \tag{59}$$
$$\frac{\partial r_{mse}}{\partial v_4} = \frac{\partial r_{mse}}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \frac{\partial r_{mse}}{\partial v_5} \cdot 1 \tag{60}$$
$$\frac{\partial r_{mse}}{\partial v_3} = \frac{\partial r_{mse}}{\partial v_4}\frac{\partial v_4}{\partial v_3} = \frac{\partial r_{mse}}{\partial v_4} \cdot 2v_3 \tag{61}$$
$$\frac{\partial r_{mse}}{\partial v_1} = \frac{\partial r_{mse}}{\partial v_3}\frac{\partial v_3}{\partial v_1} = \frac{\partial r_{mse}}{\partial v_3} \cdot 1 \tag{62}$$
$$\frac{\partial r_{mse}}{\partial v_2} = \frac{\partial r_{mse}}{\partial v_3}\frac{\partial v_3}{\partial v_2} = \frac{\partial r_{mse}}{\partial v_3} \cdot -1 \tag{63}$$

In the equations above, the right-hand side contains known values, so working back like this allows to determine all required derivatives. In this case $\partial r_{mse}/\partial v_2$ is the derivative of r_{mse} with respect to $\hat{y}$. Normally, the node $\hat{y}$ is where the expression tree of a candidate function is inserted. The same method is applied to find the derivatives with respect to all constants in the $\hat{y}$ expression tree. Note that to find the derivatives the values of nodes v_1 to v_7 are required, thus the intermediate values of these nodes should be saved during the forward pass to be used in the backward pass.

This method always has the same form for the elementary operations in ET nodes. Consider the minus operation v_3 and equations 51, 62 and 63. The minus node subtracts the right child from the left child, the derivative of the right child is the same as the derivative of the parent node, but for the left child it is always minus the derivative of the parent node. These rules are specified for all mathematical operators in the library used by DSR, see appendix B for a table containing all these rules as implemented.

To achieve an additional speedup, the backward pass can be stopped once the required derivatives are known. Consider the example and the ET in Figure 10. If only the derivative $\partial r_{mse}/\partial n$ is required, evaluating equation 59 gives the required answer. By keeping track of which nodes are parents of constants during the forward pass, many unnecessary calculations can be avoided during the backward pass by only updating derivatives that are required.

The use of algorithmic differentiation in reverse mode with the DSR code results in a 50% reduction of computational time required in a benchmark problem. This benchmark problem is detailed in section 4.1.1.

Dynamic iteration limit
An additional speedup is achieved by constraining the number of iterations for each constant optimisation routine. However, to explore the full potential of the best expressions it is interesting to know what constants and corresponding reward would result from unconstrained optimisation.

Each training step in DSR includes sampling a batch of 1000 expressions for which the constants are optimised. Initially the maximum number of iterations is limited for all expressions in the batch. After this first optimisation the best expressions are selected and the iteration limit is lifted to explore the maximum potential of these expressions. Using this iteration limit resulted in a five fold reduction of computational time with only a small loss of accuracy. See section 4.1.2 for details of choosing this limit.

3.2.5 Training the LSTM using policy gradients

After sampling and evaluating a batch of expressions, those expressions are used to train the LSTM. The difficulty is that there is no direct relation between the rewards r of each expression and the output of the LSTM ψ. However, there is a direct relation between the probability distributions ψ and the expected value of the rewards. Therefore risk seeking policy gradient is used to train the LSTM with gradient based training, which is a modification of the standard policy gradient objective.

Standard policy gradient
The objective is to maximise the expectation of the rewards r:

$$J_{\text{std}}(\mathbf{W}) \doteq \mathbb{E}_{\tau\sim p(\tau|\mathbf{W})}[r(\tau)] \tag{64}$$

To find the gradient of the objective J_{std} with respect to the LSTM weigths $\mathbf{W}$, the REINFORCE policy gradient is used [50]:

$$\nabla_{\mathbf{W}} J_{\text{std}}(\mathbf{W}) = \nabla_{\mathbf{W}} \mathbb{E}_{\tau\sim p(\tau|\mathbf{W})}[r(\tau)] \tag{65}$$

$$= \mathbb{E}_{\tau\sim p(\tau|\mathbf{W})}[r(\tau)\nabla_{\mathbf{W}} \log p(\tau|\mathbf{W})] \tag{66}$$

Using a batch of N expressions $\tau^{(i)}$, a Monte Carlo estimate of the gradients $\nabla_{\mathbf{W}}$ is:

$$\approx \frac{1}{N}\sum_{i=1}^{N} r(\tau^{(i)})\nabla_{\mathbf{W}} \log p(\tau^{(i)}|\mathbf{W}) \tag{67}$$

This estimate has high variance, a baseline function b is subtracted to reduce variance:

$$\approx \frac{1}{N}\sum_{i=1}^{N} \left[r(\tau^{(i)}) - b\right]\nabla_{\mathbf{W}} \log p(\tau^{(i)}|\mathbf{W}) \tag{68}$$

Risk-seeking policy gradient.
The standard policy gradient objective J_{std} is effective when the desired objective is to maximise the average performance of a policy. In this case a policy is an expression. However, in this application of symbolic regression the final performance is measured by the single best fitting expression, the average performance of all expressions is less important. Inspired by risk-averse learning that is robust to catastrophic outcomes ([51]), Petersen introduces the risk-seeking policy gradient. This objective aims to improve the performance of the highest raking expressions at the expense of average performance.

In practice this means that only the best performing quantile of expressions in a batch are used to train the LSTM, i.e. all expression with a reward greater than r_q. The risk seeking objective J_{risk} is defined as:

$$J_{\text{risk}}(\mathbf{W}; q) \doteq \mathbb{E}_{\tau\sim p(\tau|\mathbf{W})}[r(\tau)\,|r(\tau) \geq r_q(\mathbf{W})] \tag{69}$$

Here r_q is the $(1-q)$-quantile of the rewards. r_q is estimated for each batch and is defined as:

$$r_q(\mathbf{W}) \doteq \inf\{r(\tau) : \text{CDF}(r(\tau)\,|\mathbf{W}) \leq 1-q\} \tag{70}$$

Similar to the standard policy gradient in equations 65 to 68, the risk seeking policy gradient can be estimated by:

$$\nabla_{\mathbf{W}} J_{\text{risk}}\left(\mathbf{W}; q\right) \approx \frac{1}{qN} \sum_{i=1}^{N} \Big[r(\tau^{(i)}) - b \Big] \cdot \mathbf{1}_{r(\tau^{(i)}) \geq r_q(\mathbf{W})} \nabla_{\mathbf{W}} \log p(\tau^{(i)}|\mathbf{W}) \tag{71}$$

Here $\mathbf{1}_x$ returns 1 if condition x is true and 0 otherwise, this makes the selection between expressions above or below r_q. The baseline b used in DSR is simply a constant and equals 0.5.

3.2.6 Entropy objective

To promote selecting diverse expressions an entropy objective is added to the risk seeking policy gradient. This essentially introduces a regularisation parameter to the objective function to prevent overfitting. For a discrete probability distribution like the LSTM output ψ, the entropy H_ψ is defined as:

$$H_\psi(\psi) = -\sum_{j=1}^{n_{\mathcal{L}}} p\left(\tau_j\right) \log p\left(\tau_j\right) \tag{72}$$

Here $p(\tau_j)$ denotes the probability of selecting the j^{th} token from library $\mathcal{L}$, these probabilities for all tokens are contained in ψ. $n_{\mathcal{L}}$ is the number of tokens contained in library $\mathcal{L}$. The equation above gives the entropy of a single probability distribution $\psi^{(i)}$ used to sample a single token τ_i. To find the entropy of a full expression the following equation is used:

$$H(\tau_1 \cap \tau_2) = H(\psi^{(1)}) + H(\psi^{(2)}|\tau_1) \tag{73}$$

This equation, derived by Shannon [52], shows that the entropy of joint event $\tau_1 \cap \tau_2$ equals the entropy of event τ_1 plus the conditional entropy of event τ_2 given τ_1. This is exactly how the expressions are sampled, the probability distributions used to sample the next token are conditioned on the previously selected tokens. The selection of τ_1 using $\psi^{(1)}$ changes the LSTM output $\psi^{(2)}$ due to the LSTMs ability to maintain a memory state and the observations presented each time step. So naturally, all probability distributions $\psi^{(i)}$ are conditioned on the previously selected tokens. The expression used to find the entropy of a sampled expression then becomes:

$$H_\tau(\tau) = \sum_{i=1}^{n_i} H_\psi(\psi^{(i)}) \tag{74}$$

The equation above gives the entropy of a single sampled expression. n_i denotes the number of tokens in the complete traversal τ. However, the target to maximise is the entropy of the LSTM itself. This is also named the entropy of the source and can be calculated with the following expression [52]:

$$H_{\text{LSTM}} = \sum_{k=1}^{n_\tau} p(\tau^{(k)}) H_{\tau^{(k)}}(\tau^{(k)}) \tag{75}$$

So, the entropy of the source is the entropy of each possible expression weighted by the probability of that expression occurring. This is summed over all possible traversals n_τ given a certain traversal length and size of library $\mathcal{L}$. This is a too large number to evaluate, for example using traversal length 30 and a library of 14 tokens gives roughly $2.4 \cdot 10^{34}$ possible traversals. Therefore this value will be approximated for each sampled batch of N expressions using the following estimator:

$$H_{\text{LSTM}} \approx \frac{1}{N} \sum_{k=1}^{N} H_{\tau^{(k)}}(\tau^{(k)}) \tag{76}$$

With this estimator the weighting by the probability of an expression $\tau^{(k)}$ occurring is incorporated because higher probability states will occur more often in the sampled batch. That this expression is an estimator for equation 75 is verified numerically.

When this entropy is high there is high randomness, when the entropy is zero there is no randomness. To promote sampling diverse expressions the goal is to promote high entropy, thus H_{LSTM} is added to the J_{risk} objective to be maximised together. However, maximum entropy is not desired, since then the sampled expressions are completely random. Therefore, the magnitude of the entropy objective is scaled with parameter λ_H to adjust the relative importance compared to the risk seeking policy gradient objective. The final objective function is:

$$J_{\text{target}} = J_{\text{risk}} + \lambda_H H_{\text{LSTM}} J_{\text{risk}-1} \tag{77}$$

Here $J_{\text{risk}-1}$ is the magnitude of the risk seeking policy gradient objective of the previous iteration. This is used to ensure the entropy objective is always in the same order of magnitude as the risk seeking policy gradient objective. It was encountered that during training the maximisation of J_{risk} resulted in large increases, while the magnitude of the entropy objective did not see such an increase. When J_{risk} was sufficiently large, the small values of $\lambda_H H_{\text{LSTM}}$ did not have the desired effect of regularisation. Therefore the scaling with $J_{\text{risk}-1}$ is introduced so that the entropy objective is always a certain fraction λ_H of the risk seeking policy objective.

3.3 LSTM architecture

The network architecture used by DSR consists of one or multiple layers, where each layer contains a number of LSTM units. The number of units and layers that work best on the turbulence modelling problem is determined in the hyperparameter search as detailed in section 4.1.3. The output of the LSTM are the probability vectors ψ, the shape of which is the same regardless of the number of LSTM units in the final layer. Therefore as final layer a dense layer is used, which is a fully connected neural network layer that simply transforms the outputs of all LSTM cells to the required output shape.

3.4 Tensor basis DSR

To make predictions of the corrections to the anisotropy tensor, the standard form introduced in equation 36 is used to ensure Galilean invariance. Similar to the SpaRTA framework, only the first four base tensors are used to model b_{ij}^{Δ}, the resulting standard form is:

$$b_{ij}^{\Delta} = g^{(1)}T^{(1)} + g^{(2)}T^{(2)} + g^{(3)}T^{(3)} + g^{(4)}T^{(4)} \tag{78}$$

To ensure this form, tensor basis DSR (TBDSR) is introduced, inspired by the TBNN [32] and TBRF [34]. TBDSR uses a single LSTM network to simultaneously sample four expressions for $g^{(1)}$ to $g^{(4)}$. These scalar functions are combined with the correct base tensor to create a single expression for b_{ij}^{Δ}. This expression is then evaluated and used to train the LSTM as described above in this chapter. Each sample step the LSTM is presented with parent and sibling observations of all four $g^{(m)}$ functions to ensure the structure of the full equation can be interpreted by the LSTM.

Furthermore, in TBDSR batching is applied to reduce computational time. Each training set contains 15,000 to 20,000 cells in the mesh. When fitting expressions to all six unique components of the symmetric tensors, the resulting data set contains more than 90,000 points. To improve the computational time of TBDSR each training iteration a batch of 15,000 of these points is used. The next iteration this batch is rotated and 15,000 different points are used.

4

Experimental setup

4.1 Hyperparameter search

Finding what neural network architecture works best on a given problem is not an exact science. During the hyperparameter search the variables and parameters that affect the LSTM training are varied, to explore what combination of parameters give the best result. In this case the goal is to find the combination of parameters that produces an expression that gives the highest reward r on the data set.

In the original DSR research a hyperparameter search was performed, the goal was to fully recover certain benchmark expressions [2]. In the turbulence modelling problem it is not expected that a perfect match is possible, thus the goal of finding the highest possible reward is slightly different than the goal of fully recovering an expression. Therefore, a new hyperparameter search is performed where the values found by Petersen are used as a starting point. Also because the network architectures for symbolic regression of $\mathcal{P}_k^\Delta$ and b_{ij}^Δ are different, separate hyperparameter searches are performed for both corrections.

To get a good impression of performance for a single combination of parameters, several LSTMs should be trained. This because the training is dependent on random processes like sampling of tokens and random initialisation of neural network weights. Due to this randomness the final maximum reward of two LSTMs, trained with the same settings can differ significantly. For each combination of parameters 100 LSTMs are trained with different seeds for the random initialisation. Because training such a large amount of LSTMs is a computationally expensive task, the hyperparameter search is not performed on the large turbulence data set but on a small benchmark problem.

The settings that affect the training phase can be divided into two categories; parameters that affect the optimisation of constants in each expression and parameters that affect the training of the LSTM. Which are discussed in sections 4.1.2 and 4.1.3 respectively.

4.1.1 Benchmark equations

To reduce the computational requirements the hyperparameter search is not performed on the large turbulence data set, but on significantly smaller fabricated data sets. To make the benchmark problem as similar as possible the same number of input variables are created with similar magnitudes as the the inputs from the turbulence data set. The used input variables in X are randomly sampled from a uniform distribution as below. The values in the brackets denote the lower limit, the upper limit and the number of sampled points:

$\mathcal{P}_k^\Delta$	
$x_1 =$	$U(0, 7, 30)$
$x_2 =$	$U(0, 0.1, 30)$
$x_3 =$	$U(0, 1, 30)$
$x_4 =$	$U(-1, 0, 30)$
$x_5 =$	$U(-0.0002, 0.0003, 30)$
$x_6 =$	$U(0, 2, 30)$
$x_7 =$	$U(-1, 1, 30)$

b_{ij}^Δ	
$x_1 =$	$U(0, 2, 50)$
$x_2 =$	$U(0, 0.1, 50)$
$x_3 =$	$U(0, 1, 50)$
$x_4 =$	$U(-1, 0, 50)$
$x_5 =$	$U(-0.0002, 0.0003, 50)$
$x_6 =$	$U(2, 5, 50)$
$x_7 =$	$U(-1, 1, 50)$
$x_8 =$	$U(0, 2, 50)$
$x_9 =$	$U(0, 0.1, 50)$
$x_{10} =$	$U(-1, 0, 50)$
$x_{11} =$	$U(1, 7, 50)$

These input variables are used to calculate a corresponding target variable y. To achieve as much similarity as possible to the turbulence modelling problem these expressions are chosen to be rather complex, containing many constants and a wide range of operations present in the token library $\mathcal{L}$. Furthermore, it is expected that the best model does not use all inputs as introduced in section 3.1.1, so some available inputs are not used in the benchmark equations. The used benchmark expression for $\mathcal{P}_k^\Delta$ is:

$$y_{bm,\mathcal{P}_k^\Delta} = 4.5x_2 \cdot \exp(x_3) + x_4^2 + 0.5 \cdot \log\left(1.7x_2 + \frac{1}{x_1}\right) \tag{79}$$

For the b_{ij}^Δ benchmark a similar form will be enforced as in the TBDSR introduced in section 3.4. Here the input variables x_1, x_2, x_3, x_4 are used as dummy base tensors and the scaling functions $g^{(m)}$ are modelled using the remaining variables in the set. The benchmark expression for b_{ij}^Δ is:

$$y_{bm,b_{ij}^\Delta} = g_{bm}^{(1)} \cdot x_1 + g_{bm}^{(2)} \cdot x_2 + g_{bm}^{(3)} \cdot x_3 + g_{bm}^{(4)} \cdot x_4 \tag{80}$$

Where:

$$g_{bm}^{(1)} = 0.1 \cdot (x_6 + x_7) + 1.7 \cdot \log(x_8) + 0.23 \tag{81}$$

$$g_{bm}^{(2)} = 3 \cdot x_8 \cdot x_9 + 0.7 \cdot \exp(x_9) + 15 \tag{82}$$

$$g_{bm}^{(3)} = \exp(x_7) + \frac{3}{\log(x_6)} \tag{83}$$

$$g_{bm}^{(3)} = 10 \cdot x_5 + x_8 \tag{84}$$

The different terms and constants in the benchmark equations above are chosen to ensure all terms are roughly in the same order of magnitude. Lastly, some noise is added. This ensures the equations cannot be exactly recovered using the input variables present in the input set. This noise is sampled from a normal distribution with mean zero and a standard deviation of 0.15 for both benchmark expressions.

4.1.2 Constant optimisation control

The parameters affecting the constant optimisation make a trade-off between duration and accuracy. The aim is to set these parameters to achieve high accuracy while limiting the computational requirements. This is controlled by the dynamic iteration limit as introduced in section 3.2.4. In practice this means there is an iteration limit when optimising the constants, but to explore the maximum possible reward this iteration limit is removed for a selected subset of expressions.

In section 3.2.5 the risk seeking policy gradient is introduced, where the LSTM is trained on only the best performing expressions of a sampled batch. A risk factor q of 0.05 is used, which means that the top 5% of each sampled batch is used for training the LSTM. This presents a natural opportunity to remove the iteration limit for that subset, allowing the LSTM to train on fully optimised expressions.

Limiting the iterations before selecting this sub batch introduces two risks. Firstly, the top 5% expressions could be different than they would have been without an iteration limit. That also means the LSTM is trained with different expressions than would be used without an iteration limit. Secondly, it is possible that the best expression of the batch is not included in the sub batch. This could mean not finding the all time best expression even though it was present in the sampled batch.

A DSR runs with unconstrained optimisation is analysed to determine the iteration limit, see Figure 11. The match percentage expresses the similarity of the selected sub batch with and without iteration limit. Figure 11 also shows the probability density of how many iterations are required to ensure the best expression of the whole batch is selected for unconstrained optimisation.

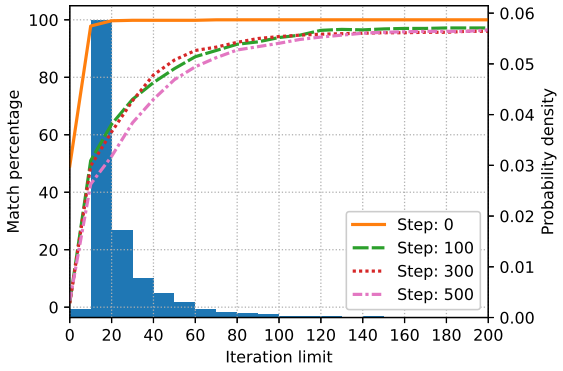


Figure 11: Batch match percentage and probability density function versus iteration limit.

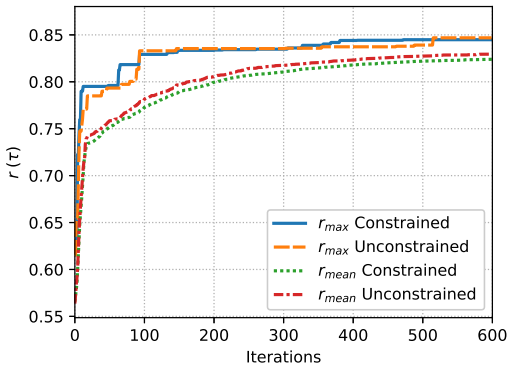


Figure 12: Comparison of rewards for constrained and unconstrained optimisation.

Figure 11 shows four curves of average match percentages taken at different training steps. This because during training the neural network learns to use more constants which affects the constant optimisation. For the first full batch, only 20 constant optimisation iterations are required to get a 100% match of the unconstrained sub batch. When training progresses, more constants are present in the expressions and the match percentages drop. At training step 500 the selected sub batch with a iteration limit of 200 is roughly 95% similar as the sub batch resulting from unconstrained optimisation.

The probability density shows that in most cases the best expression of a full batch would be selected for the sub batch with a relatively low iteration limit. However, there are also cases where the best expression would only be included in the sub batch after a constant optimisation limit of more than a 1000 iterations.

A reasonable choice seems to be an iteration limit of 100 iterations for the full batch. Then, in 97.4% of the cases the best expression of the full batch will be included in the sub batch. The used batch for training is roughly 90% similar as it would have been with unconstrained optimisation. To see what effect this has on the LSTM training this iteration limit of 100 is implemented and compared to a run where the constant optimisation is not constrained, see Figure 12.

It can be seen that constraining the constant optimisation iterations results in a lower mean reward. This is to be expected because some of the expressions are simply not fully optimised, thus the rewards are lower. However, in symbolic regression the target is the single best fitting expression with the maximum reward, not the average performance of a batch. It can be seen that the maximum rewards of constrained and unconstrained optimisation are very similar, showing that constraining the number of iterations has little effect on finding the best expression. Setting the limit at 100 iterations resulted in a 80% reduction of computational time required to evaluate 600,000 expressions.

4.1.3 LSTM control

When tweaking the hyperparameters that affect the training of a neural network there are three important factors [16]. Firstly, the capacity of the model to represent the problem at hand, in this case that mainly has to do with the number of layers and LSTM units. For example a NN with a single layer containing one LSTM unit is unlikely to be able to learn the structure of an expression. Secondly, it is important that the model can effectively maximise the objective function. Lastly it is important to what extent the cost function effectively regularises the model, in this case that would be the weight associated with the entropy objective.

Common methods to explore the best combination of parameters are grid search and random search. In grid search each parameter under consideration is discretised and all possible combinations of parameters are evaluated. For increasing number of parameters to be varied this quickly results in very high computational cost. A more efficient method is random search, where random combinations of hyperparameters are evaluated. For problems with a large amount of hyperparameters it is shown that random search converges to good values for each parameter much quicker than grid search [53].

However, random search is still computationally expensive because a significant amount of redundant parameter combinations are evaluated. In this research a manual search will be performed. This starts by evaluating the sensitivity of relevant parameters, after which combinations of well performing settings are studied.

The parameters that are varied in the hyperparameter search can be seen in table 1. This table also shows the values for each parameter as resulted from the original DSR article, these values are used as a starting point. Furthermore, due to limited computational resources some parameters that showed little sensitivity in the original DSR paper are not included in this hyperparameter search. Lastly, the number of training iterations is lowered to 1,000 to reduce computational time. These parameters not included in the search are listed in table 2.

Table 1: LSTM training control parameters and initial values.

Name	Symbol	Value
Entropy weight	λ_H	0.005
Learning rate	α	0.0005
Number of units	n_{units}	32
Number of layers	n_{layers}	1
Initialiser	-	Zeros

Table 2: LSTM training control parameters that are adopted from the original DSR research.

Name	Symbol	Value
Batch size	N	1,000
Number of training iterations	-	1,000
Risk factor	q	0.05
Optimiser	-	Adam

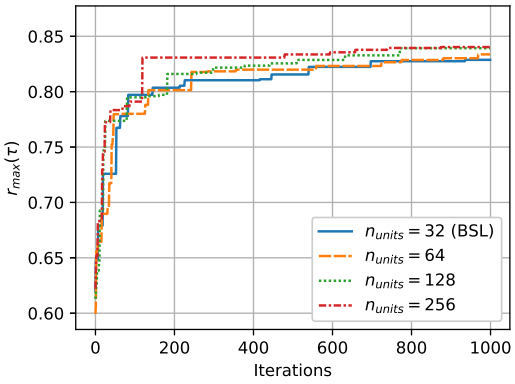
The optimiser Adam is an extension to stochastic gradient descent discussed previously and is widely used in deep learning [16]. Adam applies adaptive learning rates for each weight in the neural network, which makes the training more efficient than standard SGD.

In the symbolic regression problem at hand the goal is to find the best fitting expression. For each combination of parameters 100 LSTMs are trained. The evaluation of each setting will be done based on the best expression found in any those 100 runs. The results of the sensitivity analysis for the $\mathcal{P}_k^{\Delta}$ benchmark are presented in Figure 13.

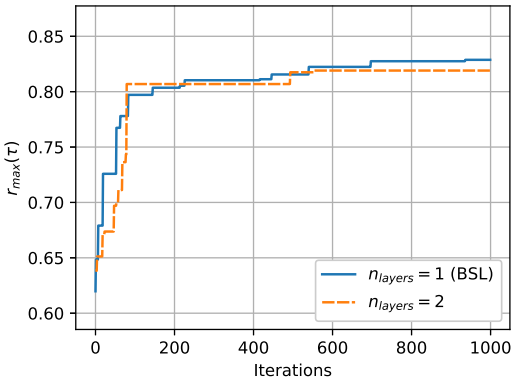
Looking at the results it is clear that the biggest sensitivity exists in the learning rate and the number of LSTM units used. Some differences are observed when varying the initialiser, entropy weight and number of layers although these seem less sensitive.

Varying the the learning rate in Figure 13c, the biggest improvement is made for increasing α, resulting in the only runs that find a reward above 0.85. The learning rate controls the magnitude of the change in network weights each training step. For a high learning rate convergence is much faster but the risk of a too large learning rate is overshooting the optimum. Because many graphs in Figure 13 are still increasing at iterations close to 1,000 it can be concluded that the neural networks are not fully trained after evaluating 1,000,000 expressions. Increasing the iteration limit would likely result in higher rewards. However, to keep the computational requirements reasonable the limit is kept at 1,000,000 and the learning rate is used to compensate for this by allowing to make bigger steps in training. The risk of missing the optimum exists when using a large learning rate. To avoid this the used optimiser Adam applies a learning rate decay, reducing magnitude of network weight updates with increasing number of iterations. From Figure 13c it is clear that the best value for the learning rate is 0.01 because this gives the highest reward.

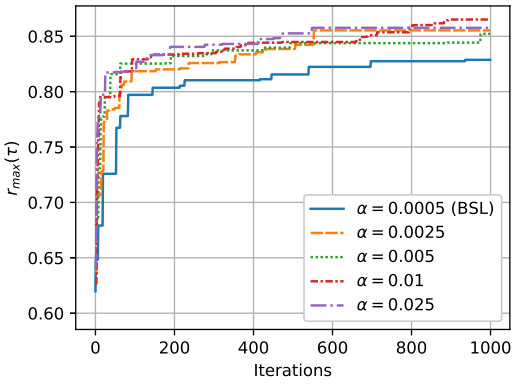
Increasing the number of LSTM units per layer also improves the maximum reward, using 64 units is still in the same order of magnitude as the baseline value of 32. But using 128 or 256 clearly improves the maximum reward, likely because a larger number of units allows the network to learn more complex relations. However, adding a second layer of LSTM units resulted in slightly worse performance compared to baseline, see Figure 13b.



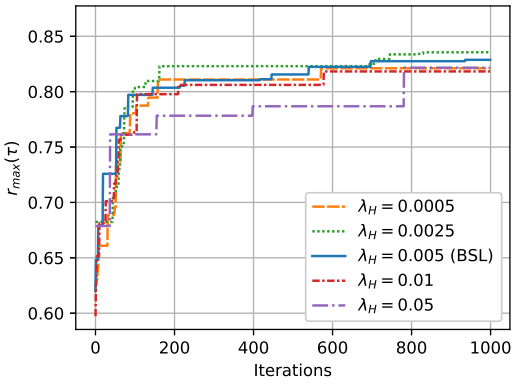
(a) Number of LSTM units per layer.



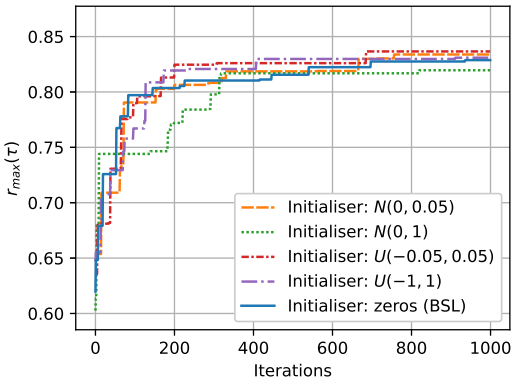
(b) Number of layers in neural network.



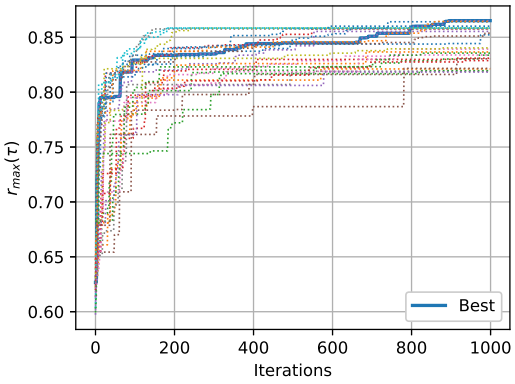
(c) Learning rate α.



(d) Entropy weight λ_H.



(e) Neural network weights initialiser.



(f) All $\mathcal{P}_k^{\Delta}$ runs with best highlighted.

Figure 13: Maximum reward vs Iteration for varying LSTM control parameters in the $\mathcal{P}_k^{\Delta}$ benchmark. BSL indicates this is the baseline value.

By initialising the weights of the LSTM units randomly the search space is likely increased because the starting point of each network is different. DSR originally used zeros initialisation where all weights are set to zero, changing this to normal or uniform distributions resulted in small improvements. Although a large spread in the weights can also be detrimental, as is observed for the initialisation of weights from a normal distribution with mean zero and a standard deviation of one in Figure 13e. Sampling from a uniform distribution with a small range of (-0.05, 0.05) is considered best.

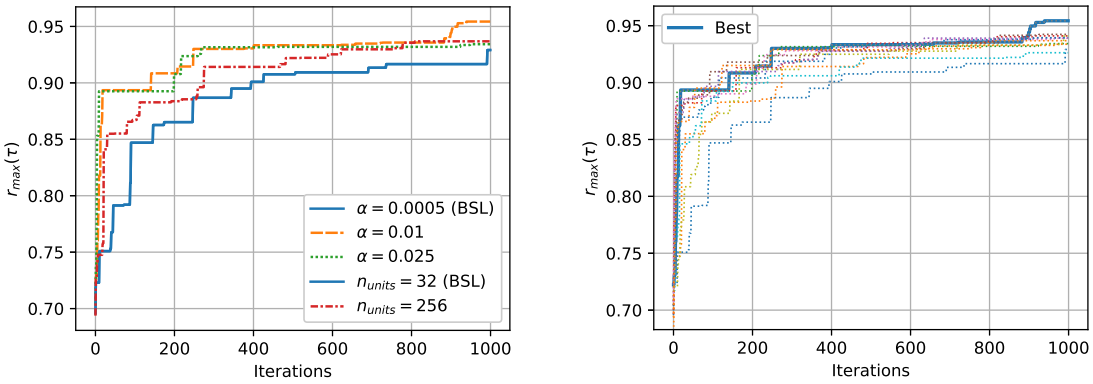
Lastly, the entropy weight gives a slight improvement by halving the value from baseline. A too large value of λ_H results in too much regularisation where the network is forced to sample practically random expressions. A too low entropy weight allows the network to quickly assign large probabilities to select certain tokens narrowing the search space. An entropy weight of 0.0025 allows to focus on maximising the policy gradient objective while preventing the search space becoming too small.

Having evaluated the sensitivity and best value of each parameter individually, the best values are combined to see if that further increases the reward function. However, not a single combination resulted in a reward higher than observed with learning rate 0.01 and the other parameters as set in table 1. Significant differences in training speed and average reward of all expressions were observed, but in the end the single highest reward is what counts. See Figure 13f, where all $\mathcal{P}_k^{\Delta}$ runs are plotted and the run with the highest final reward is highlighted. It can be seen that some combinations have a steep learning curve, finding rewards above 0.85 after 200 iterations. These runs likely found a local minimum quickly and then failed to improve.

Running the sensitivity analysis for b_{ij}^{Δ} is a significantly heavier computational task. Considering the largest sensitivity of the learning rate and the number of LSTM units the search is limited to these two parameters. Figure 14a shows the sensitivity of both the learning rate and the number of units. A similar result is observed as before where increasing the value of both improved the final maximum reward. Several combinations of parameters are explored but again the best performing run is with a learning rate of 0.01 and the other parameters as in table 1. The LSTM hyperparameters that will be used for the turbulence data set are presented in table 3.

Table 3: LSTM training control parameters and final values.

Name	Symbol	Value
Entropy weight	λ_H	0.005
Learning rate	α	0.01
Number of units	n_{units}	32
Number of layers	n_{layers}	1
Initialiser	-	Zeros



(a) Learning rate and number of units

(b) All b_{ij}^{Δ} runs with best highlighted.

Figure 14: Maximum reward vs Iteration for varying LSTM control parameters in the b_{ij}^{Δ} benchmark. BSL indicates this is the baseline value.

4.2 Turbulence modelling

Three high fidelity LES and DNS cases are used to find models for $\mathcal{P}_k^{\Delta}$ and b_{ij}^{Δ}. The cases are simple two dimensional geometries exhibiting flow separation over a curved surface. This is one of the flow conditions where traditional RANS turbulence models make the largest errors. The models that DSR finds for these CFD cases are implemented in a custom $k-\omega$ SST turbulence model in OpenFoam. This will verify convergence of the solver and allows test whether the resulting RANS flow field is improved. The SpaRTA research uses the same CFD cases as training data which allows a fair comparison of the models found by DSR and SpaRTA.

4.2.1 High Fidelity CFD cases

Periodic hills (PH)

The first case is periodic hills, flow through a channel with evenly spaced hills on the bottom surface, see Figure 15. The high fidelity data is an LES simulation performed by Breuer [54] at a Reynolds number of 10,595. The RANS simulations that are compared with this case are performed on a fine mesh of 120×130 cells. Cyclic boundary conditions are used on the inlet and outlet to simulate an infinitely long channel. Also there is experimental data available at $Re = 37,000$ for the same geometry from a study by Rapp [55]. This experimental data is used to test the generalisation of models found with DSR by using them at a significantly higher Reynolds number than used in the training cases.

Converging-diverging channel (CD)

The second case is a DNS simulation over an asymmetric bump at a Reynolds number of 12,600 performed by Laval [56]. The corresponding RANS case uses a channel flow

at the same Reynolds number as inlet boundary condition and is solved on a mesh of 140×100 cells. The full domain ranges from $0 < x/H < 12.5$, Figure 16 shows only part of the domain.

Curved backward-facing step (CBFS)

The third case is a LES simulation over a curved backward-facing step at a Reynolds number of 13,700. The RANS simulations are performed on a mesh of 140×150 where a fully developed boundary layer is applied to the inlet. See the domain in Figure 17, this figure is zoomed to the step and does not show the full domain. The full domain ranges from $-7.3 < x/H < 15.3$ and $0 < y/H < 9.5$.

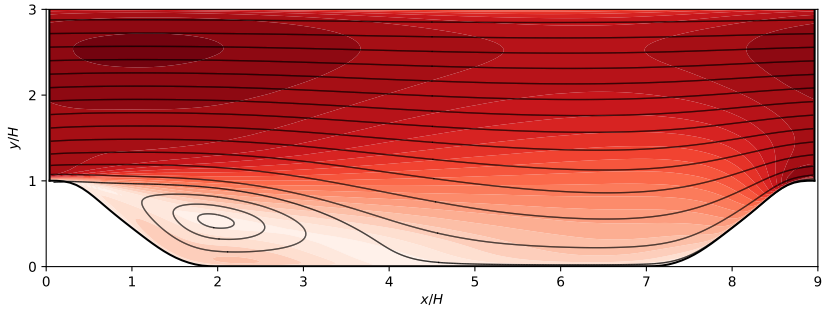


Figure 15: Flow domain of the PH_{10595} case, showing streamlines and coloured by magnitude of velocity, LES data.

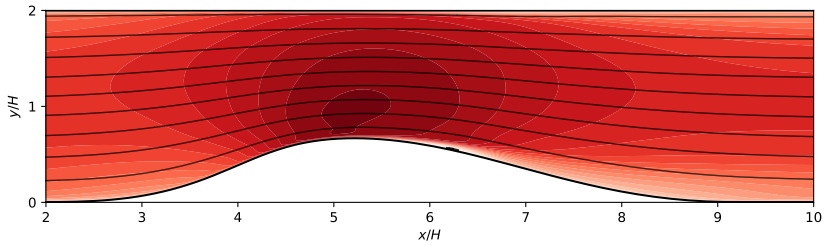


Figure 16: Flow domain of the CD_{12600} case, showing streamlines and coloured by magnitude of velocity, DNS data.

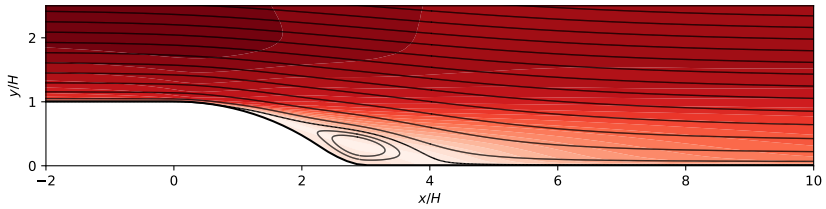


Figure 17: Flow domain of the $CBFS_{13700}$ case, showing streamlines and coloured by magnitude of velocity, LES data.

4.2.2 Model selection

DSR is used to fit expressions to the training cases. A single DSR run evaluates 1,000,000 (not all unique) models in batches of 1,000, the best fitting model of each batch is saved. A large number of candidate models are found after completing 100 parallel DSR runs. From these the best models are selected based on their fit to the data set.

The models that achieve the best combined fit on all data sets, as well as models that perform good on the training data set are implemented in a turbulence model and tested in a RANS simulation. The models for $\mathcal{P}_k^\Delta$ and b_{ij}^Δ are first tested individually and the fit on each of the three CFD cases is evaluated. The resulting flow fields are compared to the LES or DNS cases on the same domain. The metric to judge CFD performance is the mean squared error of the velocity field with respect to the high fidelity data, normalised by the mean squared error of the standard $k-\omega$ SST model.

For each training data set the five best models for $\mathcal{P}_k^\Delta$ and b_{ij}^Δ are selected. The 25 possible combinations of these models are tested in CFD and evaluated. Finally, the best model for each domain is selected, these will be compared to the models that were found in the SpaRTA research. See the three best SpaRTA models below:

$$M_{sparta}^{(1)} \begin{cases} b_{ij}^\Delta &= 0 \\ \mathcal{P}_k^\Delta &= 2k(0.39T^{(1)})\bar{S}_{ij} \end{cases} \tag{85}$$

$$M_{sparta}^{(2)} \begin{cases} b_{ij}^\Delta &= 0.1T^{(1)} + 4.09T^{(2)} \\ \mathcal{P}_k^\Delta &= 2k(1.39T^{(1)})\bar{S}_{ij} \end{cases} \tag{86}$$

$$M_{sparta}^{(3)} \begin{cases} b_{ij}^\Delta &= 0 \\ \mathcal{P}_k^\Delta &= 2k(0.93T^{(1)})\bar{S}_{ij} \end{cases} \tag{87}$$

Highly complex models were found with the SpaRTA symbolic regression. However, using no models for b_{ij}^Δ gave the best results on two of the three cases. To find $\mathcal{P}_k^\Delta$ models SpaRTA uses the same base tensor architecture which is used to find b_{ij}^Δ models. The resulting tensor models are then multiplied by $2k\bar{S}_{ij}$ to find a scalar term similar to the standard turbulent kinetic energy production term. The models SpaRTA found for $\mathcal{P}_k^\Delta$ are again very simple models consisting of just the first base tensor scaled by a constant.

To ensure a fair comparison to SpaRTA the DSR models should be limited to similar complexity as the SpaRTA models. This is achieved by limiting the length of expressions in DSR. For $\mathcal{P}_k^\Delta$ that means seven tokens will be used to find models, as that is how many tokens are required to build the SpaRTA models above.

The complexity for b_{ij}^Δ is more difficult, as models cannot be shorter than 0. Five tokens are used to create the b_{ij}^Δ correction of $M_{sparta}^{(2)}$. However, TBDSR directly samples the functions g that scale the base tensors and sums the four tensors. So with a token limit

of 5, the final expression can have a length over 20 tokens. Thus to achieve similar complexity as $M^{(2)}_{sparta}$ only one token would be allowed, which is not an interesting test for DSR. The possibility is explored that models found with DSR can outperform using no model for b^{Δ}_{ij}, a maximum model length of 3 tokens for each function g is used. The experiments where DSR models for $\mathcal{P}^{\Delta}_{k}$ are tested individually will serve as a comparison against using no model.

The effect of more complex models is studied by also running DSR with more tokens. For $\mathcal{P}^{\Delta}_{k}$ DSR is ran with the number of tokens limited to 7, 10, 12 and 20. For b^{Δ}_{ij} the token limits are 3, 5, and 10.

5

Results and Discussion

In this chapter the results of applying DSR to the turbulence modelling problem are presented. First the selection based on reward achieved on the data set is analysed. These models are then implemented in a turbulence model and tested in CFD. Using these CFD results the best three models are chosen and the flow field is analysed in detail. Lastly the best models are tested for a true prediction in a case with a Reynolds number roughly triple the magnitude of the cases used in training.

5.1 DSR training

During the hyperparameter search the maximum number of tokens for each expression was 30. Running DSR with significantly lower token limits showed that the optimal hyperparameters for shorter expressions are slightly different. For example, all parallel instances of $\mathcal{P}_k^{\Delta}$ models of length 10 got stuck in a local minimum. When halving the learning rate, DSR managed to find what is expected to be the global optimum. See Figure 18 for distributions of rewards of unique expressions found with varying and learning rate and entropy weight. BSL indicates the baseline hyperparameters as determined in section 4.1.3.

Furthermore, by reducing the learning rate DSR found slightly more unique expressions, see Figure 18. Also, with entropy weight $\lambda_H = 0.005$ only few unique expressions are found that are close to the maximum reward. Therefore, the entropy weight is increased to force the algorithm to sample more diverse expressions. With an entropy weight of 0.05 DSR found the suspected global optimum and also many unique expressions with high reward. The DSR runs with the turbulence data are performed with these updated settings of $\alpha = 0.005$ and $\lambda_H = 0.05$.

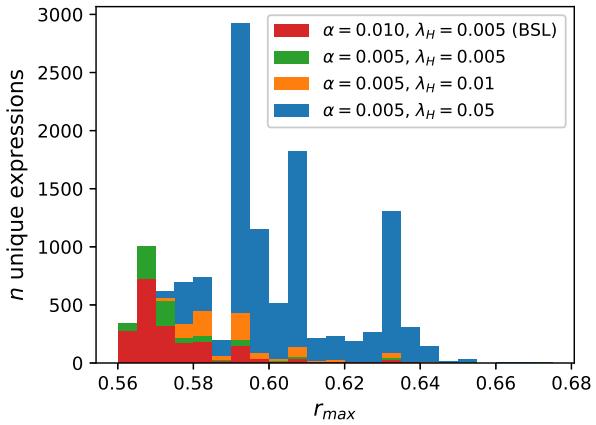


Figure 18: Distribution of rewards for different values of learning rate α and entropy weight λ_H for $\mathcal{P}_k^\Delta$ models of 10 tokens.

Running DSR is a computationally expensive task, especially considering that many parallel runs need to be performed with random initialisation to cover a sufficiently large search space. Increasing the number of tokens in an expressions results in significantly longer time required to complete a DSR run. See Figure 19, here the average processor time a single parallel DSR run takes is shown against the number of tokens.

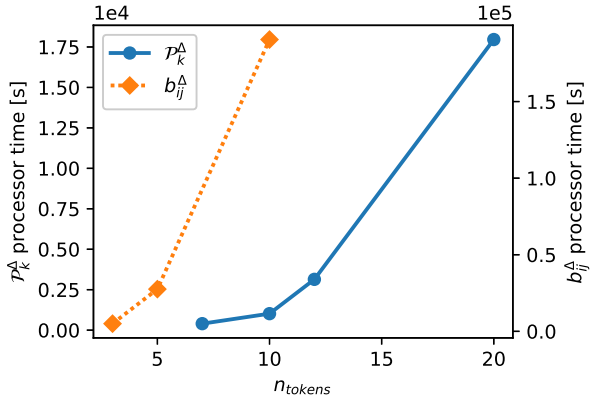


Figure 19: Processor time versus number of tokens for $\mathcal{P}_k^\Delta$ on left axis and b_{ij}^Δ on right axis.

The required processor time for both models scales roughly with $O(n_{tokens}^{3.5})$. The duration increases quickly because a longer expression needs more floating point operations to be evaluated. Furthermore, when an expression is longer, more constants can be selected. With increased number of constants to be optimised, more iterations are required to find the optimal constants. So, a single function evaluation is more expensive, but also more function evaluations are required when increasing the number of tokens.

5.2 Model discovery

Running DSR on each data set described in section 4.2.1 results in a large amount of models for $\mathcal{P}_k^\Delta$ and b_{ij}^Δ. The first selection is made to include the models that achieve the highest reward across the three test cases, as well as expressions that achieve the highest reward on the training case.

5.2.1 $\mathcal{P}_k^\Delta$ models

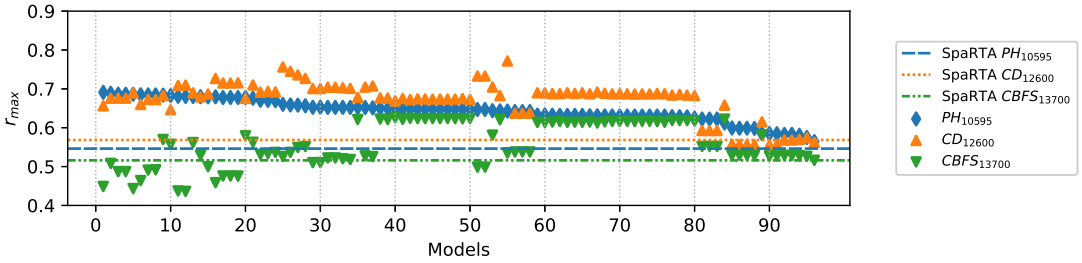
For $\mathcal{P}_k^\Delta$, 96, 83, and 95 models are selected that are trained on PH_{10595}, CD_{12600} and $CBFS_{13700}$ respectively. These models and their fit to each of the data sets are presented in Figure 20. Here all models are plotted on the x-axis and are sorted by the best fit on the training case. The reward function is bounded from zero to one, a reward of one is a perfect fit. If a model has only one point for one data set, that means the reward function on the other data sets is below 0.4.

A trend can be seen that the models which achieve the highest reward on the training set achieve a poor reward on the other two data sets. This shows that the best fitting expressions are overfitted to the training set and do not generalise well to the other cases.

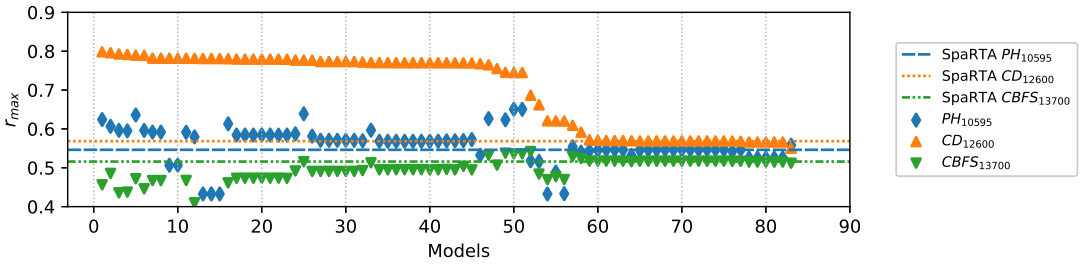
Furthermore it can be seen that in terms of achieved fit on the training set all models greatly outperform the SpaRTA models. However, the models presented in the SpaRTA article are not selected based on fit to the data but on the performance in a CFD solver. So this comparison may not be the most representative of the possible performance of SpaRTA to find expressions.

5.2.2 b_{ij}^Δ models

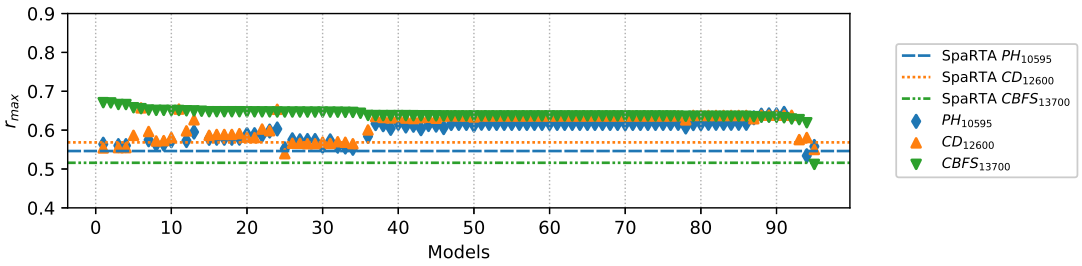
For b_{ij}^Δ 92, 94 and 93 models are selected for PH_{10595}, CD_{12600} and $CBFS_{13700}$ respectively. The same analysis is made for the maximum reward in Figure 21, here the results are similar as for $\mathcal{P}_k^\Delta$. The poor generalisability is especially clear with the models trained on PH_{10595} and $CBFS_{13700}$ in Figures 20a and 20c respectively. For these cases the best 10 models on the training set achieve poor rewards on the other two cases.



(a) Trained on PH_{10595}

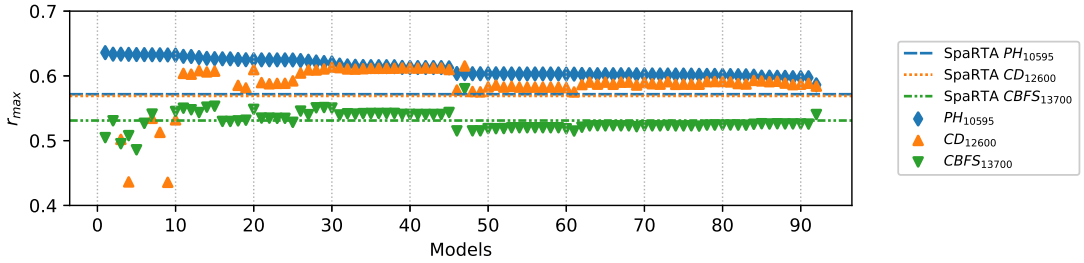


(b) Trained on CD_{12600}

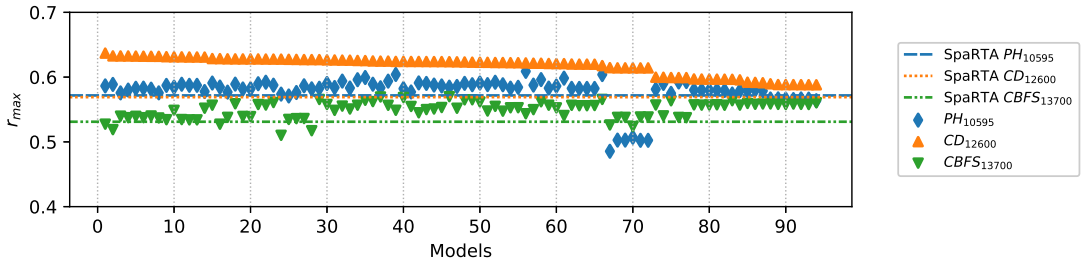


(c) Trained on $CBFS_{13700}$

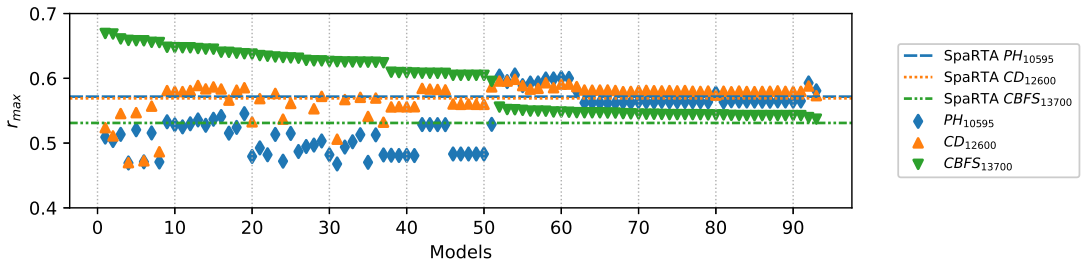
Figure 20: $\mathcal{P}_k^{\Delta}$ models with maximum reward for each training set.



(a) Trained on PH_{10595}



(b) Trained on CD_{12600}



(c) Trained on $CBFS_{13700}$

Figure 21: b_{ij}^{Δ} models with maximum reward for each training set.

5.3 CFD CROSS-VALIDATION

The models presented in the previous section are implemented in a $k-\omega$ SST turbulence model and the resulting flow field is compared to the high fidelity data. First, the models for $\mathcal{P}_k^{\Delta}$ and b_{ij}^{Δ} are tested separately to evaluate individual performance. Based on these results, the five best models are selected for both corrections and all possible combinations of these models are also tested in CFD.

Introducing the corrections into a turbulence model negatively affects the stability of the solver, especially simulations with b_{ij}^{Δ} corrections rarely converge. To improve stability of the solver the cases are initialised from partly converged standard $k-\omega$ SST simulations. This improved stability and more simulations with corrective models converged.

5.3.1 $\mathcal{P}_k^\Delta$ MODELS

The results of testing individual $\mathcal{P}_k^\Delta$ models are shown in Figure 22. This figure shows the same models presented in the previous section, this time sorted on the x-axis by the error of the resulting flow field. This error is the mean squared error (mse) of the RANS velocity field with respect to the high fidelity data, normalised by the mse of a standard $k-\omega$ SST model. So a value below one means the flow field has improved over the standard $k-\omega$ SST model.

Firstly it is clear that some models fail to converge or have a normalised error above 1, which means the flow field is worse than the standard $k-\omega$ SST turbulence model. Of all $\mathcal{P}_k^\Delta$ models tested across the three test cases 62% of these models converged and improved the flow field on all three cases.

DSR gives a large number of models that outperform the SpaRTA models on each test case. On the PH_{10595} data set DSR found three models that outperform $M_{sparta}^{(1)}$ on all three testcases. On the other two data sets DSR found many models that give large improvements over SpaRTA models on two of the three test cases. Also, DSR provides many models that have a lower combined error, which is the sum of errors across the three testcases. Of the 274 $\mathcal{P}_k^\Delta$ models tested individually, 29% achieved a combined error below the best SpaRTA model.

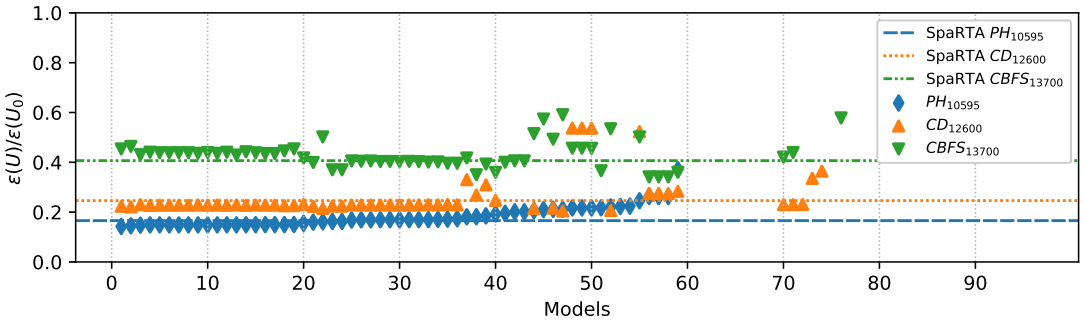
5.3.2 b_{ij}^Δ MODELS

The results of using individual b_{ij}^Δ models are quite poor, from the 279 models tested across the three training cases only 52 models converged on all three cases. Furthermore only 6 models resulted in an improved flow field across all test cases.

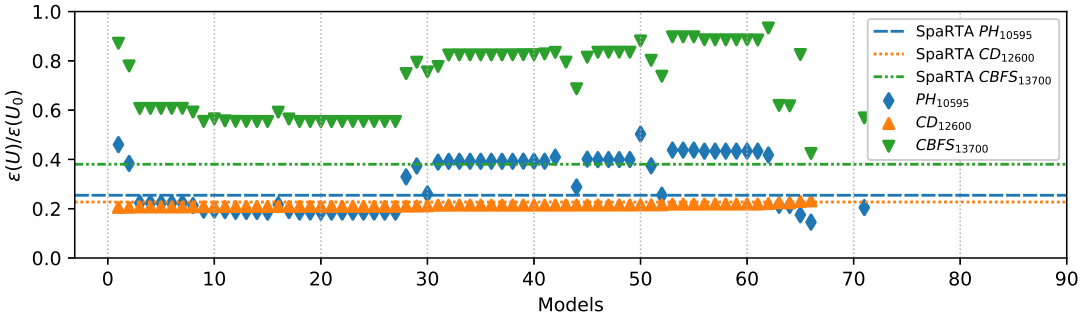
The normalised flow field errors of implementing b_{ij}^Δ models in a CFD solver are shown in Figure 23. The CFD simulations with the b_{ij}^Δ models often diverged, therefore only the best 30 for each training case are included in Figure 23. Note that the limits on the y-axis are increased to include $\varepsilon(U)/\varepsilon(U_0)$ values above 1. None of the individually tested b_{ij}^Δ models gave better results than the individually tested $\mathcal{P}_k^\Delta$ models.

$M_{sparta}^{(2)}$ is the only SpaRTA model with a b_{ij}^Δ correction. This correction is also tested separately and only converged for the CD_{12600} test case but gave a normalised error greater than 4. On the other two cases the model did not converge.

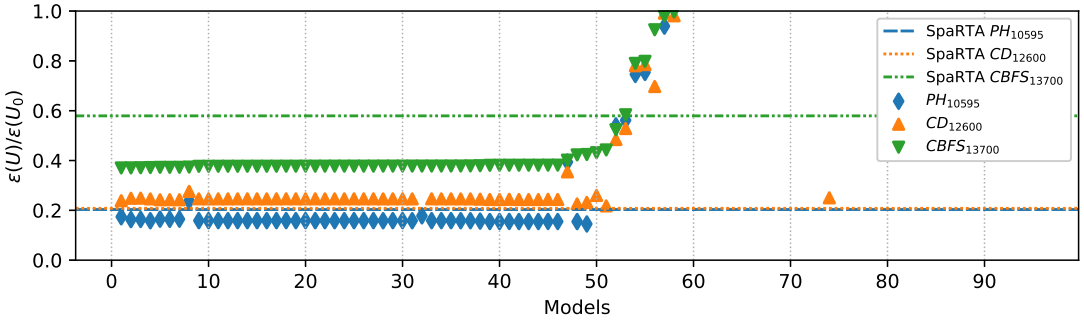
To improve convergence, the same b_{ij}^Δ models are scaled to only include 20% of the correction. This increased the number of converging models, but also reduced the improvements seen in the flow field. Decreasing the magnitude of the correction moved all errors closer to one. This is to be expected, because decreasing the size of the correction brings the simulation closer to a standard $k-\omega$ SST model.



(a) Trained on PH_{10595}



(b) Trained on CD_{12600}



(c) Trained on $CBFS_{13700}$

Figure 22: Normalised error of flow field obtained by implementing $\mathcal{P}_k^{\Delta}$ models in a $k-\omega$ SST turbulence models.

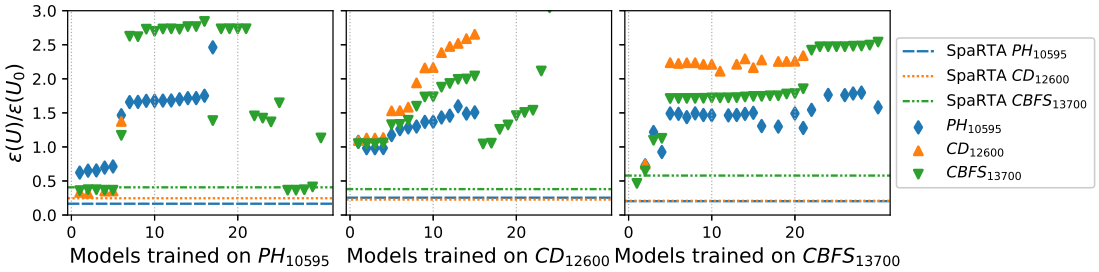


Figure 23: Normalised flow field errors for the 30 best b_{ij}^{Δ} models trained on each data set.

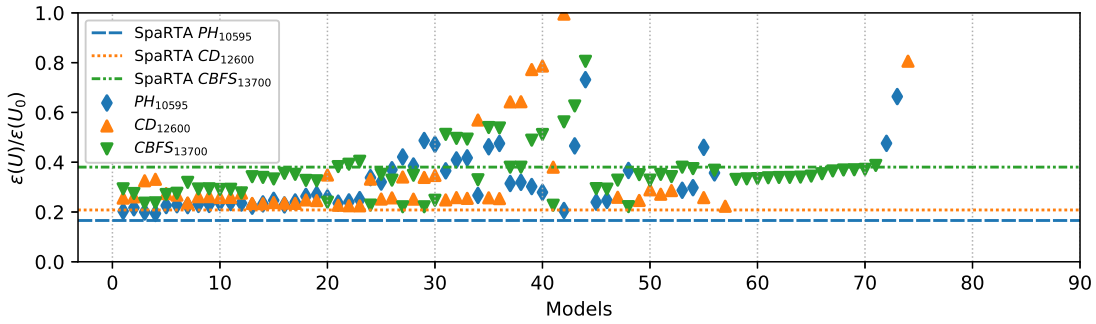


Figure 24: Normalised flow field errors for all tested combined models.

5.3.3 Combined $\mathcal{P}_k^{\Delta}$ and b_{ij}^{Δ} models

Using the results of testing both models separately, the five best $\mathcal{P}_k^{\Delta}$ and b_{ij}^{Δ} models are selected and for each test case the 25 possible combinations of these models are tested. For the CD_{12600} there were several interesting $\mathcal{P}_k^{\Delta}$ models, so seven are selected. To give an impression of the DSR models, the expressions of these selected models are shown in appendix A.

The flow field errors of the models combining both corrections are shown in Figure 24. The resulting flow field is improved for most combinations, considering that most b_{ij}^{Δ} models individually worsened the flowfield this is an interesting result. Good results are achieved on the $CBFS_{13700}$ case, the best error in this domain by individual models is 0.31, the best combined model achieved 0.22. On the other two cases, the results of combined models are worse than the individual $\mathcal{P}_k^{\Delta}$ models.

5.4 Best models

DSR produced numerous models that outperformed the best SpaRTA cases. For each test case the model that achieved the lowest error on is selected. Furthermore, models that diverge on one of the test cases are excluded. The best model for each case is not necessarily found by training on that same case. However, the reward optimised during training is a different goal than the CFD flow field error found after implementation. Therefore the models with the lowest flow field error are selected for each case regardless of which case is used during training.

The best models for CFD cases PH_{10595}, CD_{12600} and $CBFS_{13700}$ can be seen in equations 88, 89 and 90 respectively. The constants are rounded for readability, the expressions with full constants are presented in appendix A. The normalised errors of these models and the SpaRTA models are presented in table 4.

$$M_{dsr}^{(1)} \begin{cases} b_{ij}^{\Delta} & = 0 \\ \mathcal{P}_k^{\Delta} & = \frac{k}{\theta_1 + 0.76} T^{(1)} \bar{S}_{ij} \end{cases} \tag{88}$$

$$M_{dsr}^{(2)} \begin{cases} b_{ij}^{\Delta} & = 0 \\ \mathcal{P}_k^{\Delta} & = 18.05(14.73\theta_1 - 0.56\log(\theta_1) - 2.36) k T^{(1)} \bar{S}_{ij} \end{cases} \tag{89}$$

$$M_{dsr}^{(3)} \begin{cases} b_{ij}^{\Delta} & = 1.06\exp(109.96\theta_2) T^{(1)} + 22.15\exp(-41.23\theta_1) T^{(2)} \\ & \quad + \frac{\theta_1 + 0.13}{\theta_2} T^{(4)} \\ \mathcal{P}_k^{\Delta} & = 1.73k \left(T^{(1)} + T^{(3)} \right) \bar{S}_{ij} \end{cases} \tag{90}$$

Table 4: Normalised errors $\varepsilon(U)/\varepsilon(U_0)$ of the flow field of the best models found with DSR and SpaRTA.

Model	PH_{10595}	CD_{12600}	$CBFS_{13700}$	ε_{sum}
$M_{dsr}^{(1)}$	0.1453	0.2211	0.4637	0.8300
$M_{dsr}^{(2)}$	0.2115	0.2050	0.5911	1.0075
$M_{dsr}^{(3)}$	0.4212	0.3407	0.2210	0.9829
$M_{sparta}^{(1)}$	0.1659	0.2463	0.4066	0.8189
$M_{sparta}^{(2)}$	0.2546	0.2271	0.3804	0.8621
$M_{sparta}^{(3)}$	0.2033	0.2082	0.5793	0.9908

For PH_{10595} as training case the best model is the first model from Figure 22a, this model gives the largest flow field improvement. In terms of reward achieved on the training set this model was ranked 39^{th} out of 96 tested models, showing that a high reward in training does not necessarily translate to achieving good results in CFD.

The best model on the CD_{12600} domain is the model with index 47 in Figure 22a, this model is found by training on the PH_{10595} data set. This model achieves a slightly lower error than the best models trained on the CD_{12600} data. Also, the errors on the other two cases are better than the best models trained on CD_{12600} in Figure 22b.

Lastly, the best model for $CBFS_{13700}$ is the model with index 27 in Figure 24, using $\mathcal{P}_k^\Delta$ and b_{ij}^Δ models together. Interestingly, individually neither model stands out, but combined the performance is very good. Separately, the model for $\mathcal{P}_k^\Delta$ achieves a flow field error of 0.554 on the $CBFS_{13700}$ case, 113 of 274 tested $\mathcal{P}_k^\Delta$ models achieved a better error than that. Similarly, the b_{ij}^Δ model tested individually gave normalised errors above one for all test cases, i.e. the flow field is worse than standard $k-\omega$ SST. However, when used together these models give the lowest error on the $CBFS_{13700}$ case.

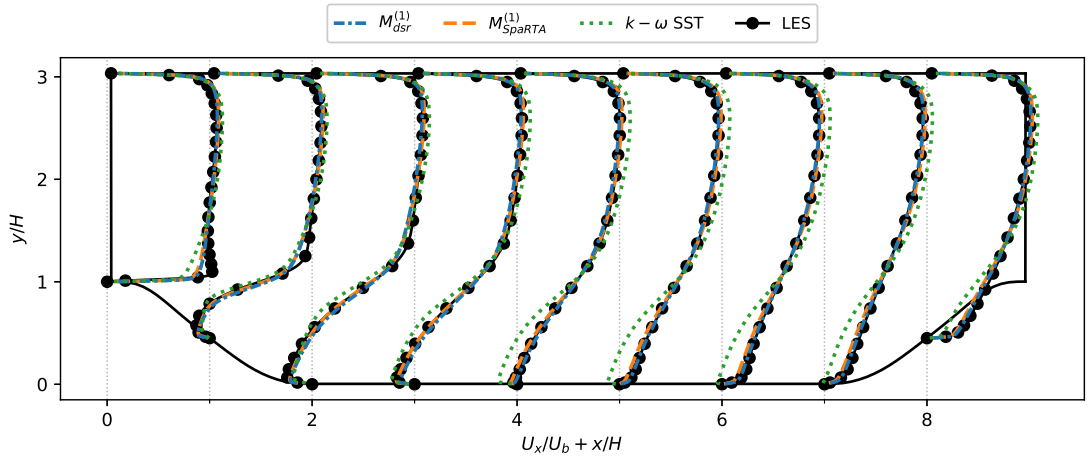
These models are more complex than the best models found in SpaRTA. This shows that the ability of DSR to create more complex expressions is beneficial for the turbulence modelling problem. In the SpaRTA research also complex models consisting of many terms are considered, but it was found that the simple models resulted in more accurate CFD simulations. This result does not mean that simple models are better, but that the regression technique in SpaRTA might be less suited to produce good models. DSR has shown that increased complexity in the models translates to improved performance of a turbulence model. However, high complexity is not the key to improving turbulence models, as the most complex models found with DSR are not the best performing models.

Looking at the flow field errors in table 4 it is clear that DSR did not provide a single model that achieved the best results of SpaRTA on all cases. However, significant improvements are found in the individual cases. DSR also provided 94 models with a lower combined error ε_{sum} than $M_{sparta}^{(1)}$, showing that the models found by DSR can generalise better across test cases than the best SpaRTA models.

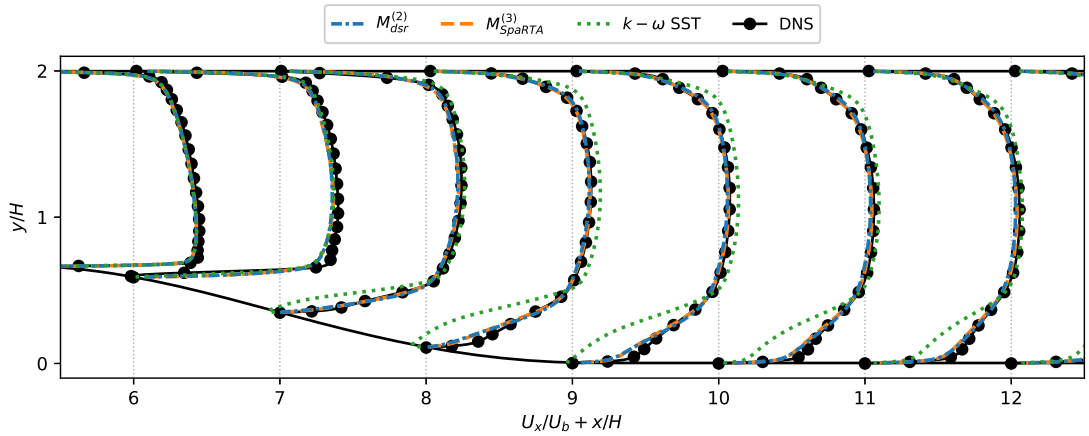
5.4.1 Flow field analysis

For each test case, the flow field of the best DSR model is compared to the flow fields of the best SpaRTA model, the high-fidelity data and a standard $k-\omega$ SST model. Figure 25 shows profiles of normalised x velocity, Figure 26 shows the turbulent kinetic energy, Figure 27 shows the xy component of Reynolds stress τ_{xy} and finally Figure 28 shows the skin friction coefficient on the lower wall.

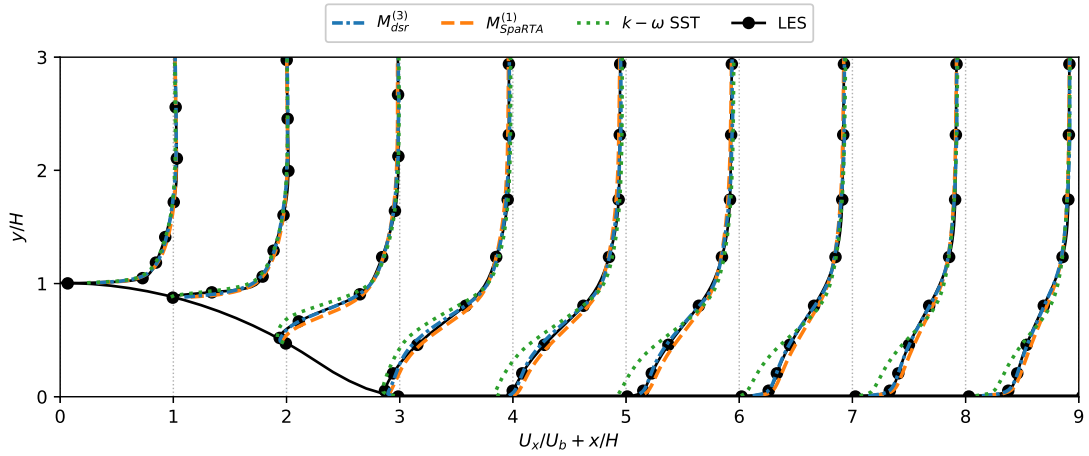
The standard $k-\omega$ SST model strongly overpredicts the size of the separated region in all cases. This is attributed to the model underestimating the turbulent kinetic energy in separated shear layers. Through the eddy viscosity, increased k will result in higher shear stresses which shortens the recirculation bubble [57, 58].



(a) Trained on PH_{10595}

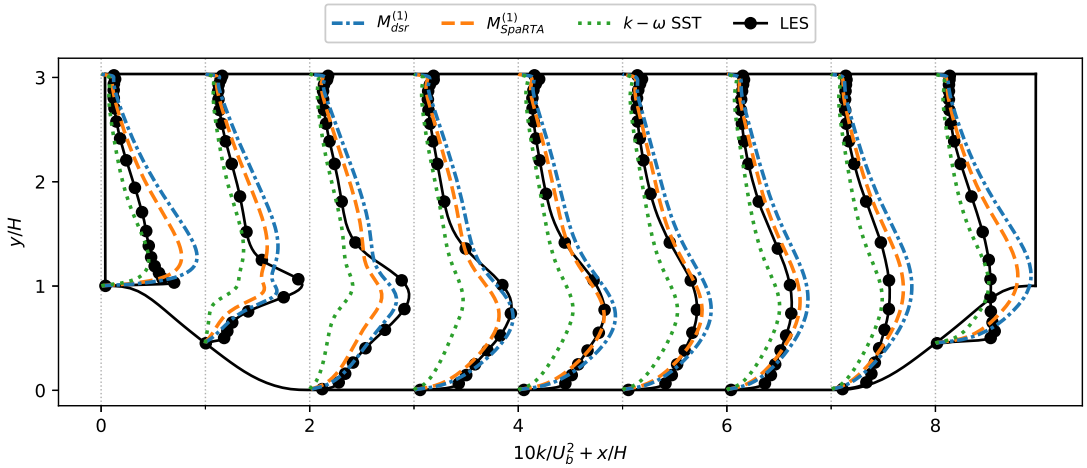


(b) Trained on CD_{12600}

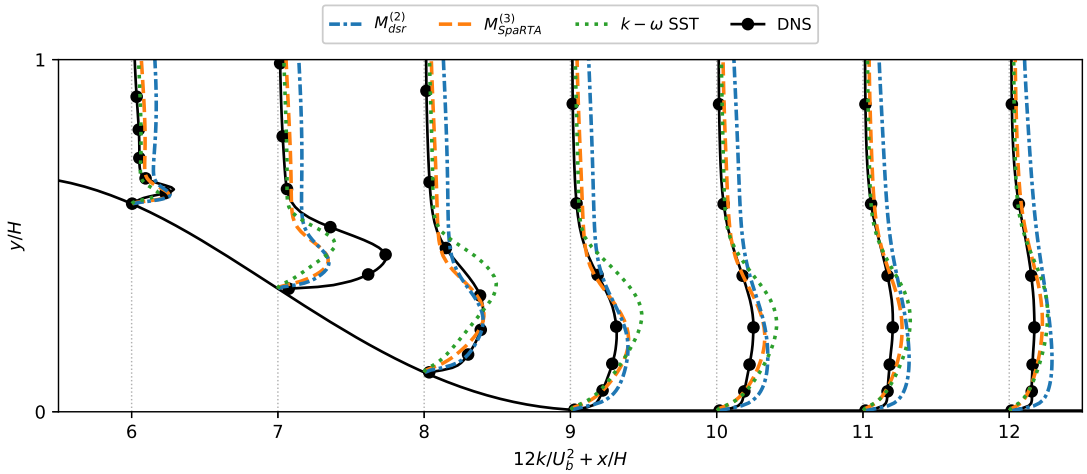


(c) Trained on $CBFS_{13700}$

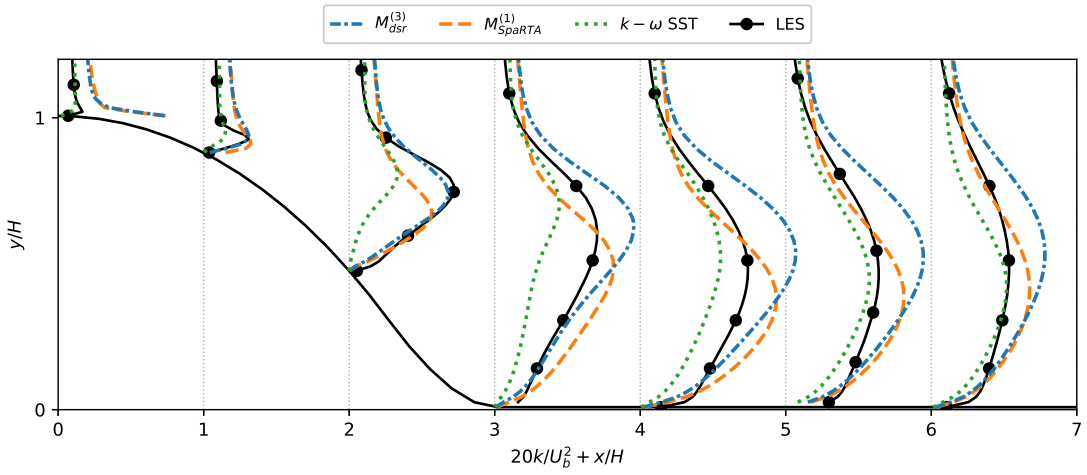
Figure 25: Comparison of x velocity profiles of the best DSR model, the best SpaRTA model, standard $k-\omega$ SST and high fidelity CFD data.



(a) Trained on PH_{10595}

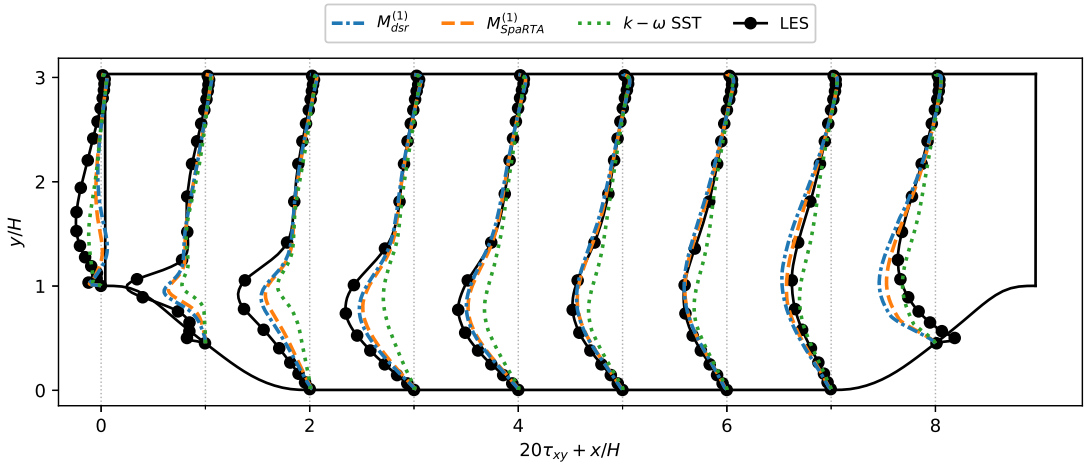


(b) Trained on CD_{12600}

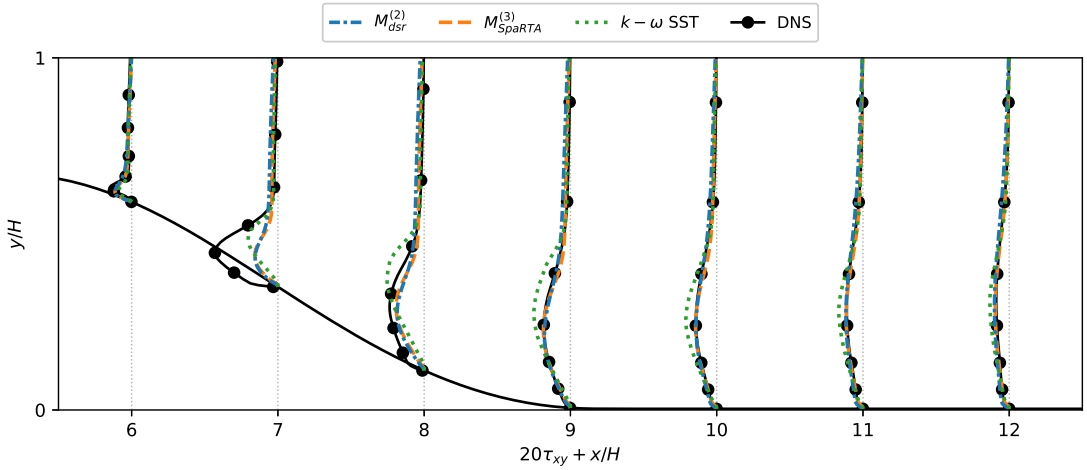


(c) Trained on $CBFS_{13700}$

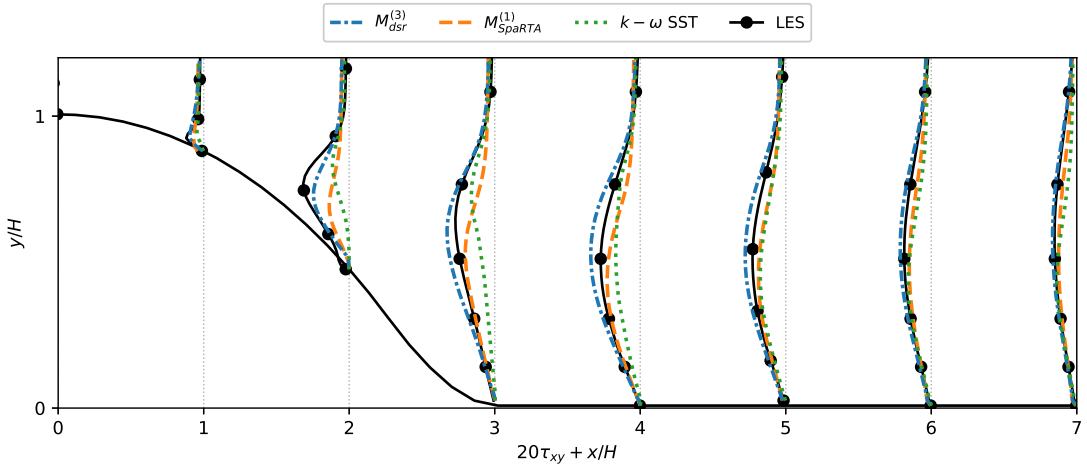
Figure 26: Comparison of k profiles of the best DSR model, the best SpaRTA model, standard $k-\omega$ SST and high fidelity CFD data.



(a) Trained on PH_{10595}

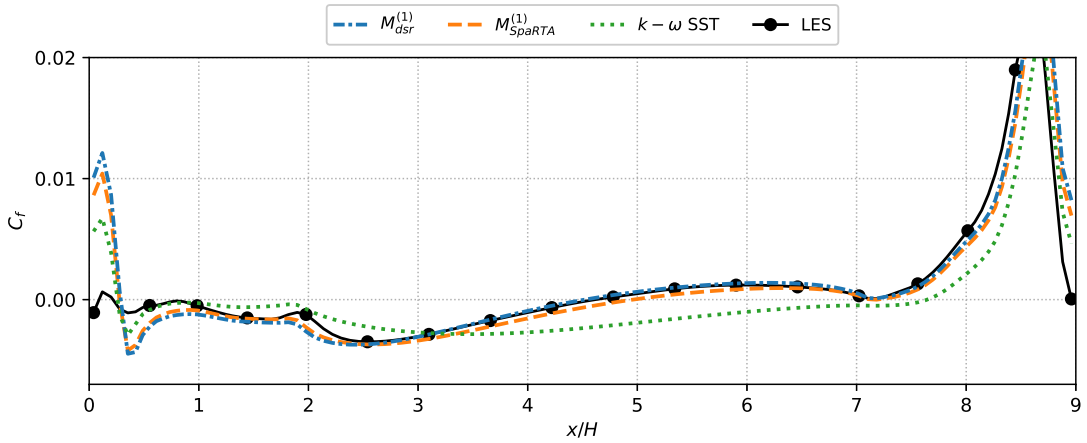


(b) Trained on CD_{12600}

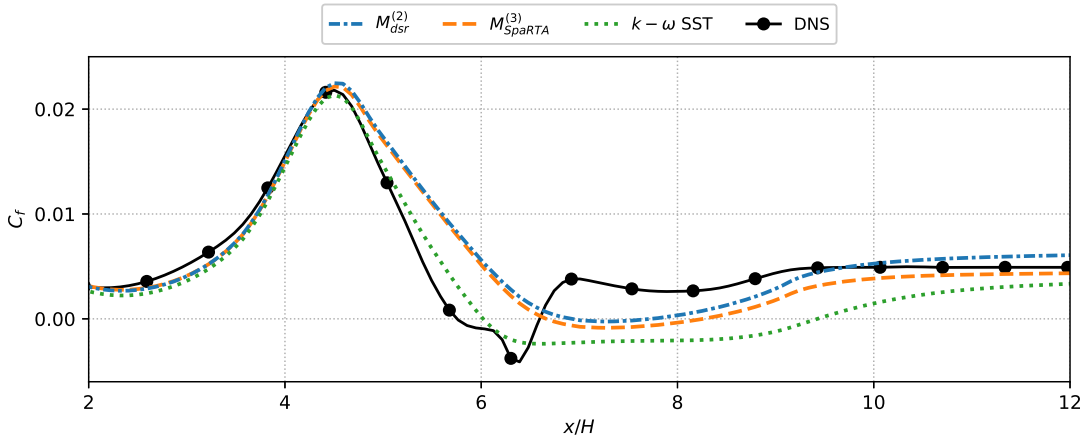


(c) Trained on $CBFS_{13700}$

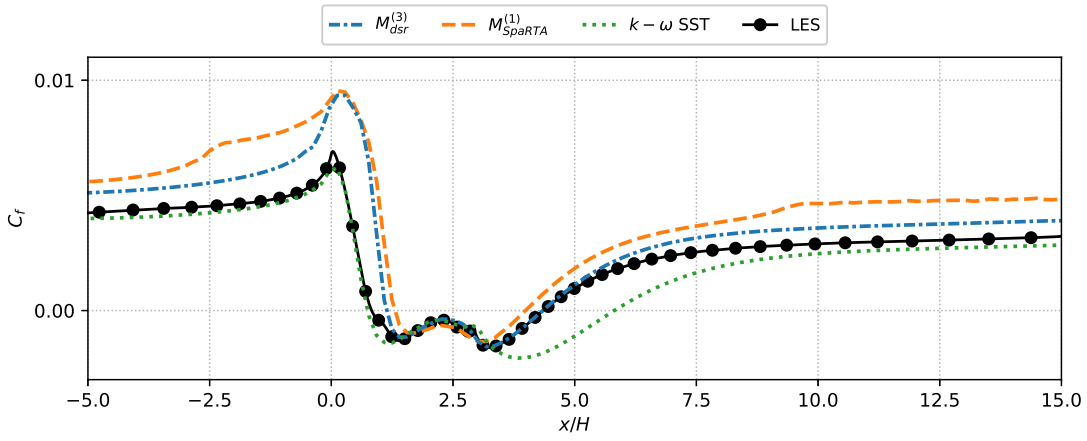
Figure 27: Comparison of τ_{xy} profiles of the best DSR model, the best SpaRTA model, standard $k-\omega$ SST and high fidelity CFD data.



(a) Trained on PH_{10595}



(b) Trained on CD_{12600}



(c) Trained on $CBFS_{13700}$

Figure 28: Skin friction on the lower wall of the best DSR model, the best SpaRTA model, standard $k-\omega$ SST and high fidelity CFD data.

On the PH_{10595} case, the DSR model offers significant improvements over the best SpaRTA model. Looking at the turbulent kinetic energy of $M^{(1)}_{dsr}$ in 26a the model fails to accurately capture the spatial structure of k in the free shear layer. However, it gives significant increases and manages to match the lower part of the LES profile for $x/H \leq 4$. Further downstream the turbulent kinetic energy is overestimated. The increase in k translates to increased shear stress in Figure 27a, but it is still quite far off the LES values of τ_{xy}. The increased shear stress shortens the recirculation zone. Looking at C_f in Figure 28a the reattachment point predicted by the DSR model is practically the same as in LES. Also the velocity profiles are very good, see Figure 25a. Both DSR and SpaRTA overestimate the velocity for $x/H \leq 2$, where the DSR model has a slightly larger overshoot than the SpaRTA model. However, for $x/H \geq 4$, the DSR model matches the LES data almost exactly, while $M^{(1)}_{sparta}$ underestimates the velocity.

On the CD_{12600} domain, little improvement of DSR over SpaRTA is observed, which is to be expected considering the normalised error of $M^{(2)}_{dsr}$ is only marginally smaller than the error of $M^{(3)}_{sparta}$. The flow contains a small recirculation bubble on the leeward side of the hill, which is not accurately captured by any of the RANS models, see Figure 28b. The standard $k-\omega$ SST model more closely captures the separation point, but greatly overpredicts the size of the separated region. The augmented turbulence models predict smaller recirculation bubbles, but too far downstream. In the other variables the biggest difference between DSR and SpaRTA can be seen in Figure 26b where DSR gives higher values of k throughout the domain. In terms of velocity profiles in Figure 25b no difference can be seen between SpaRTA and DSR, which is to be expected since the normalised errors are of the same order of magnitude.

Lastly, on the $CBFS_{13700}$ there are significant improvements of the DSR model over SpaRTA. $M^{(3)}_{dsr}$ manages to almost exactly match the turbulent kinetic energy profile at $x/H = 2$ in Figure 26c. The result is that also the shear stress τ_{xy} is closely matched on the leeward side of the step and thus also the reattachment point exactly corresponds to the LES data, see Figures 27c and 28c. Further downstream, $M^{(3)}_{dsr}$ overestimates k, τ_{xy} and C_f, but the mean velocity profiles are very close to the LES data, see Figure 25c.

It should be noted that both $M^{(3)}_{dsr}$ and $M^{(1)}_{sparta}$ significantly mispredict the skin friction on $CBFS_{13700}$ before and after the separated zone. The too high skin friction downstream of $x/H = 5$ of both models can be attributed by the significant overprediction of k resulting in too large shear stress and thus skin friction. Looking at Figure 26c, especially the SpaRTA model overpredicts the turbulent kinetic energy close to the wall which could be the cause of the large C_f.

However, the large skin friction upstream of the recirculation zone, $x/H < 2$ in Figure 28c, is less understood. Similar errors can be seen on the PH_{10595} domain in Figure 28a for $x/H < 0.5$. A highly detailed study of the PH domain by Jang [58] attributes this poor representation of the shear stresses due to the rapid acceleration on the PH domain

at $x/H < 8$. However, a similar acceleration happens on the CD_{12600} domain on the windward side of the hill and there the skin friction and shear stresses are accurately captured, see Figure 28b. The cause of the discrepancy in C_f on the $CBFS_{13700}$ domain upstream of $x/H = 2$ should be investigated in future research.

Generally, the DSR models improve the standard $k - \omega$ SST models by increasing the turbulent kinetic energy where standard $k - \omega$ SST underestimates. This results in accurate representation of the recirculation bubbles on the PH_{10595} and $CBFS_{13700}$ domain. However, the increases in k in the free shear layer dissipate slowly and k is overpredicted in the regions behind the recirculation zone. This results in the skin friction being overpredicted in parts of the domain. The overprediction of k has little effect on the velocity profiles and the velocity fields closely match the high fidelity data.

5.5 True prediction at high Reynolds number

To test the best model, it is compared to experimental results on the periodic hills domain at a Reynolds number of 37,000, see Figure 29. This Reynolds number is almost triple the values of the training cases, so it is a good opportunity to test the generalisability of the models beyond conditions encountered in training.

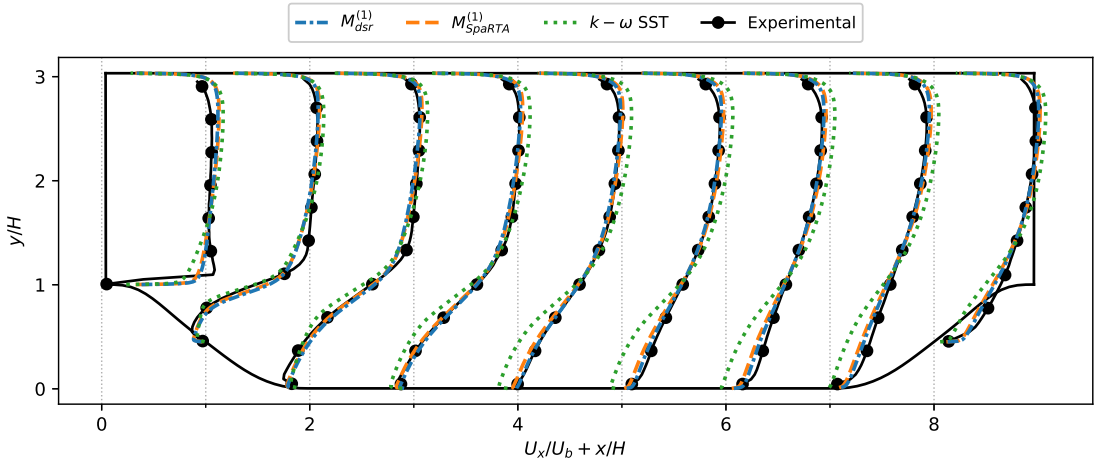


Figure 29: Profiles of x-velocity of the best models from DSR and SpaRTA compared to a standard $k - \omega$ SST model and experimental data. At $Re = 37,000$

The results are good, a significant improvement over the standard $k - \omega$ SST model is observed at this Reynolds number. DSR captures the velocity profiles more accurately compared to SpaRTA for $x/H \geq 3$. However, for $x/H \geq 6$ bigger differences are observed between the high fidelity data and the DSR model. This is a slight hint that possibly the models become less accurate for larger Re numbers, but still a significant improvement over standard $k - \omega$ SST is achieved. Another cause for some differences could be that in this comparison experimental data is used instead of LES data.

5.6 Model complexity and generalisability

When fitting models with DSR, the length of sampled expressions in the batch quickly increases until the limit of the number of tokens is hit. Indicating that the neural network learns that longer and thus more complex expressions achieve a better fit to the training set. To investigate the effect of expression length, the maximum reward achieved by $\mathcal{P}_k^\Delta$ model for each length of expression is shown in Figures 31a, 31c, 31e.

For each expression length, the best performing model on the training set is plotted. The maximum reward on the training set shows an increasing trend for increasing number of tokens in the expressions. So more complex models are able to fit the data better. However, these complex model are possibly overfitted to the training data, so the rewards that these expressions achieve on the other two data sets are also shown.

The rewards for the two cases other than the training set show a decreasing trend with increasing n_{tokens}. This confirms that more complex models that fit the training data well perform worse on the cases not seen during training. This shows that the complex models may have limited generalisability. These complex models are finely tuned to fitting the $\mathcal{P}_k^\Delta$ of the training set, but these details can be very case specific. The simple models are fitted to the general behaviour of $\mathcal{P}_k^\Delta$ which is largely similar in the other cases.

After implementation in CFD, the same graphs are made for the flow field errors in Figures 31b, 31d, 31f. It is clear that the more complex models no longer perform better than the simple models. However, the performance is also not worse. The best flow field errors achieved by the models does not seem to be related to the complexity of the expressions.

The poor generalisability of the more complex models is also seen after implementation in CFD. Looking at the errors that the best models in one case achieve in the other two cases, there is an increasing trend with an increasing number of tokens. Especially for the models trained on the CD_{12600} and $CBFS_{13700}$ data sets in Figures 31d and 31f, the error achieved on cases other than the training case increases for increasing n_{tokens}. This shows that the good performance that complex models achieve on the training case does not translate in good performance on the other cases. So, longer expressions generalise poorly compared to simple models.

Furthermore, the length of expressions had an adverse effect on the convergence of CFD simulations. As discussed, the flow field errors achieved by these models are good, but there is a clear trend that more complex models are less stable and converge less often than the simple models. See Figure 30, for example, for $\mathcal{P}_k^\Delta$ models consisting of 7 tokens, all tested models converged, while for models of 20 tokens only 50% converged. The same trend is visible for b_{ij}^Δ models.

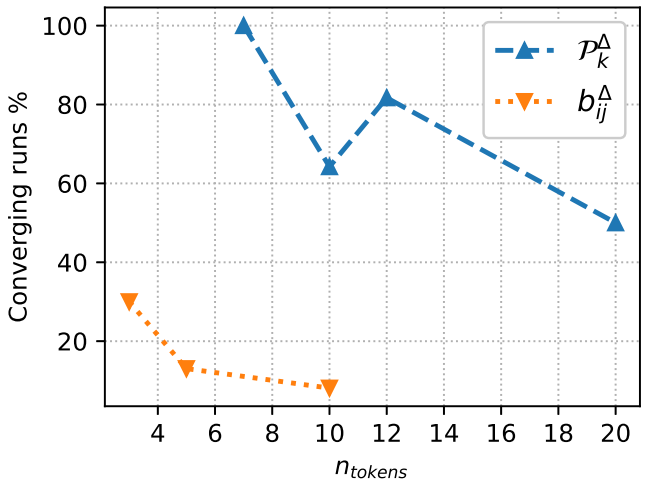
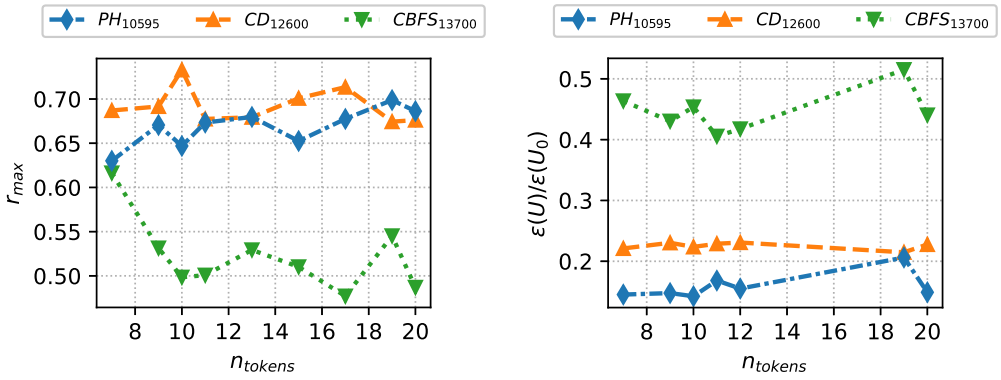
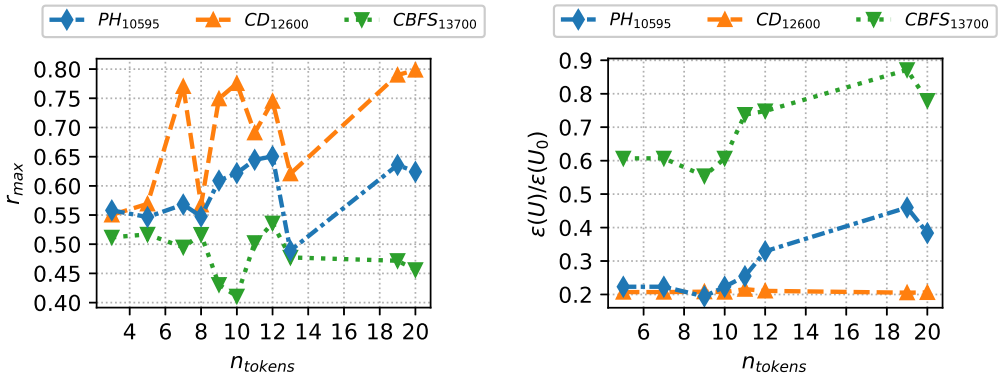


Figure 30: Percentage of converging models versus expression length, for both $\mathcal{P}_k^\Delta$ and b_{ij}^Δ models.



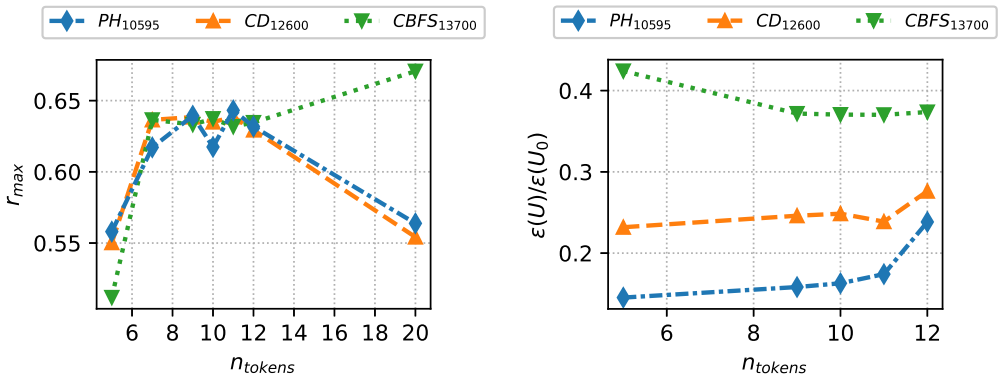
(a) Trained on PH_{10595}

(b) Trained on PH_{10595}



(c) Trained on CD_{12600}

(d) Trained on CD_{12600}



(e) Trained on $CBFS_{13700}$

(f) Trained on $CBFS_{13700}$

Figure 31: Maximum reward and minimum normalised flow field error versus number of tokens in expression, for $\mathcal{P}_k^{\Delta}$ models trained on each case.

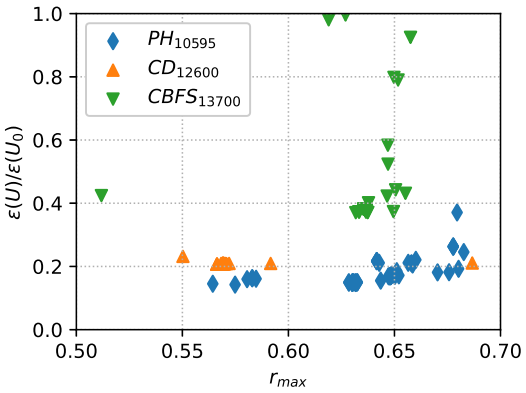


Figure 32: Plot of expression reward versus normalised flow field error.

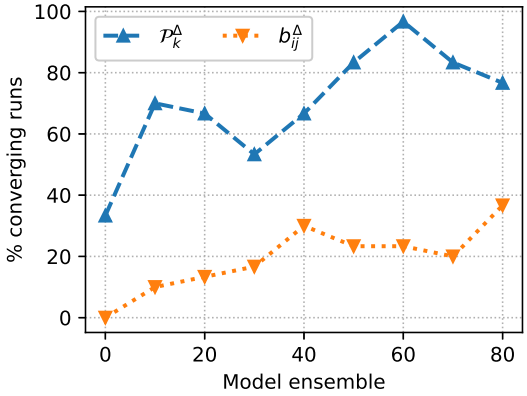


Figure 33: Average percentage converging CFD simulations of groups of 10 models, ranked by fit on the training data.

It is studied whether an expression with a high reward on the training data translates to a more accurate CFD flow field. The best DSR models achieve a significantly better fit to the training sets than the best SpaRTA models and show better performance once implemented in a CFD simulation. However, no correlation exists between the achieved rewards by DSR models and the corresponding flow field errors, see Figure 32. For the PH_{10595} there is a slight hint that models with a high reward r_{max} have worse flow field errors, for the other cases there is no clear relation between achieved reward and flow field error.

However, Figure 32 does not include runs that diverged, there is a clear trend where the best-fitting models diverge most often. Figure 33 shows the average convergence of groups of models, these groups consist of 10 models and are sorted based on training reward. So index 0 on the x-axis shows the percentage of converging runs of the 10 best fitting models. There is no clear relation between the achieved reward on the training set and the resulting flow field errors, but there is a clear trend where the best fitting models most often fail to produce a converging CFD simulation.

5.7 Model structure analysis

5.7.1 $\mathcal{P}_k^\Delta$ models

The occurrence of tokens across all models for $\mathcal{P}_k^\Delta$ is summarised in Figure 34. The most commonly used variables are $T^{(1)}\bar{S}_{ij}$, k and θ_1. It is to be expected that k and one of the base tensors are included in all models, as the product of these is the only combination that gives the correct dimensions. Furthermore, it is clear that the first invariant is used

much more often than the second. Lastly, as expected the use of constants in the models is very important, the most occurring token is the constant token. The large importance of the first base tensor $T^{(1)}$ and the first invariant θ_1 shows that the mean rate of strain is very important in the modelling of $\mathcal{P}_k^{\Delta}$. This because the $T^{(1)}$ and θ_1 are exclusively functions of $\bar{S}_{ij}$.

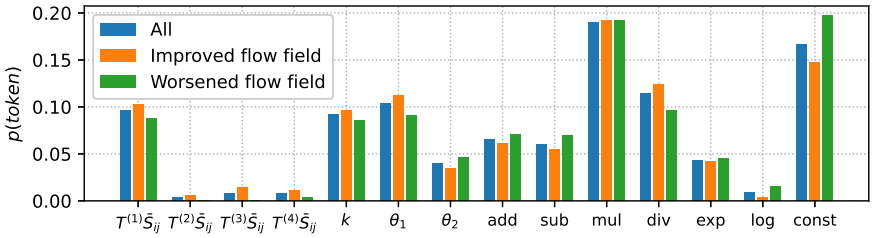


Figure 34: Figure showing the normalised occurrence of tokens in all $\mathcal{P}_k^{\Delta}$ models. Also for models that improved over standard $k-\omega$ SST and models that did not.

To study if there is a relation between certain variables and the CFD simulations the same token distribution is plotted for models that have improved the flow field across all cases and models that did not. This division plotted in Figure 34 shows there is little difference between tokens occurring in improving and non-improving models. Because there is little difference in token occurrence, it can be concluded that no single token is responsible for a models CFD flow field results.

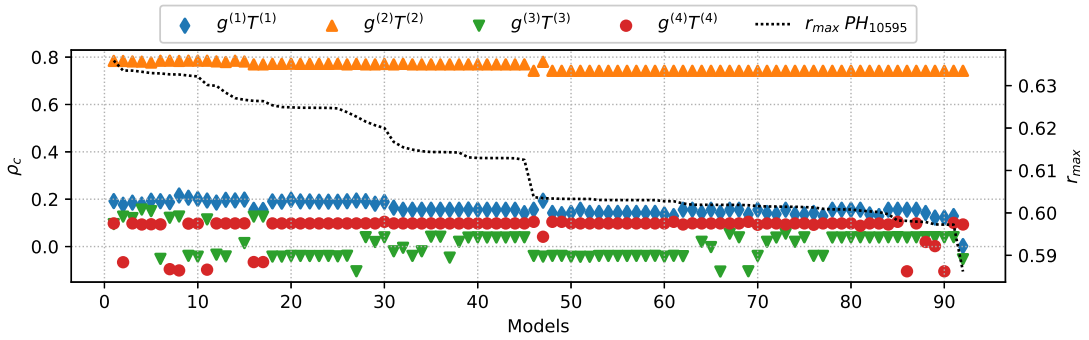
5.7.2 b_{ij}^{Δ} MODELS

Each tensor model for b_{ij}^{Δ} has the same structure of scalar functions $g^{(m)}$ that scale the base tensors $T^{(m)}$, see equation 78. To investigate the structure of the tensor models for b_{ij}^{Δ}, the correlation coefficient ρ_c between each term $g^{(m)}T^{(m)}$ and the target data for b_{ij}^{Δ} is determined, see Figure 35. The models on the x-axis are sorted based on reward achieved on the training data set, this reward r_{max} is also shown on the secondary axis.

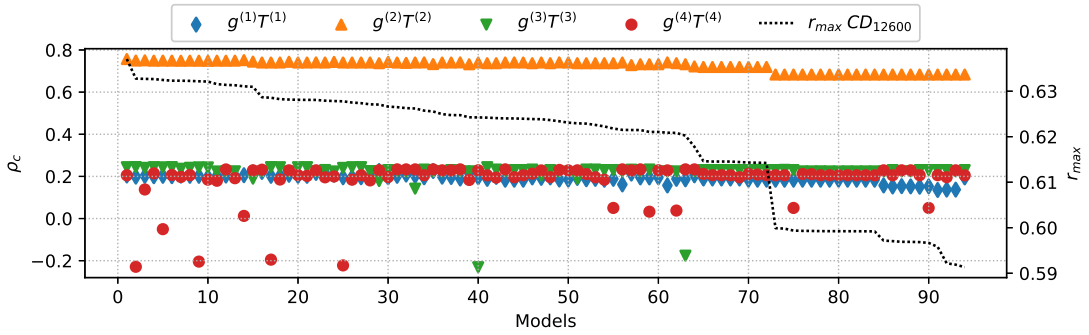
The results for PH_{10595} and CD_{12600} in Figures 35a and 35b respectively are fairly similar. In these cases, the correlation of the term with the second base tensor $T^{(2)}$ is the highest by a big margin. Which shows this term is most important in modelling the correction to the Reynolds stress anisotropy. Furthermore, the correlation coefficient of this $g^{(2)}T^{(2)}$ term shows a decreasing trend for decreasing reward r_{max}. This is an interesting result, since the initial motivation of the research was to depart from the Boussinesq approximation assuming a linear relation between the mean rate of strain and b_{ij}. The first base tensor is simply the mean rate of strain but the second base tensor is also a function of the mean rate of rotation $\bar{R}_{ij}$. That the correction is not modelled with terms that are exclusively functions of $\bar{S}_{ij}$ is an encouraging result showing that indeed variables other than $\bar{S}_{ij}$ are required.

The plot for the $CBFS_{13700}$ case looks very different, the correlation of $g^{(2)}T^{(2)}$ is significantly lower. The other terms show higher values of ρ_c than observed in the other two cases. Still, the first base tensor, which is simply the mean rate of strain, is not important for the modelling. An interesting pattern is that $T^{(3)}$ or $T^{(4)}$ shows high correlation, but never both at the same time. Using multiple base tensors to model the correction rather than one offers more flexibility in the resulting b_{ij}^{Δ} models. Possibly this is related to the fact that the models trained on $CBFS_{13700}$ generalise best in CFD. The majority of models that improved the combined error ε_{sum} across all three cases over SpaRTAs best model are all found using the $CBFS_{13700}$ training set. It is recommended to investigate this relation in future research.

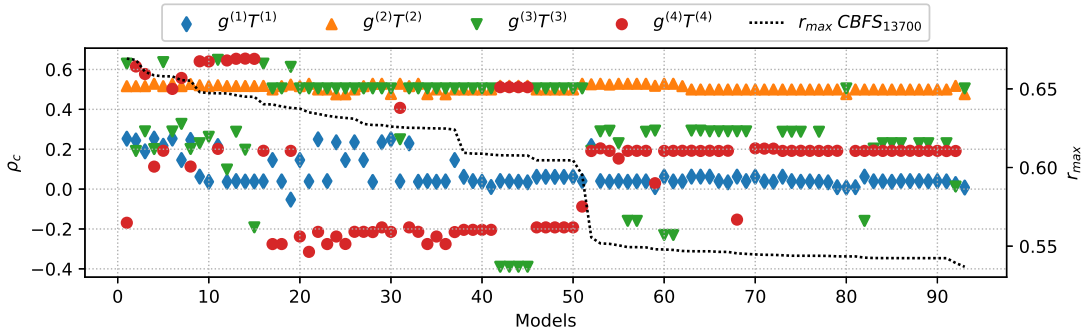
The difference between $CBFS_{13700}$ and the other cases is likely due to the fact that in the other two cases the domain is fully turbulent, while in $CBFS_{13700}$ there is a laminar region in the centre. In the laminar region, the variables used to find models are significantly different which affects training and thus the resulting models.



(a) Trained on PH_{10595}



(b) Trained on CD_{12600}



(c) Trained on $CBFS_{13700}$

Figure 35: The left Y axis shows the correlation coefficient between each $g^{(m)}T^{(m)}$ product and the training target b^{Δ}_{ij}. The models for each dataset are on the x-axis sorted by reward, the right y-axis shows the reward of these models.

6

Conclusions and Recommendations

The best models found with DSR are implemented in a $k-\omega$ SST turbulence model and tested in CFD. The resulting RANS flow field are significantly closer to the more accurate LES and DNS solutions, the best DSR model reduces the error of standard $k-\omega$ SST by 85%. Furthermore the DSR models are compared to models found with the SpaRTA framework. In SpaRTA the same high fidelity data sets are used thus it serves as a good comparison to test the performance of DSR models. DSR found models that give more accurate CFD flow fields compared to SpaRTA, also DSR provided models that generalise better across testcases than the models found in SpaRTA.

Standard $k-\omega$ SST turbulence models overpredict the size of separated flow regions, this is attributed to the underprediction of turbulent kinetic energy which through the eddy viscosity leads to underprediction of shear stress in the flow. The corrective models show significant increases of turbulent kinetic energy, matching LES or DNS k profiles in free shear layers but often overpredicting in the rest of the domain. The result is that the size of recirculation bubbles is captured very accurately by the DSR models, but the skin friction in other parts of the domain is slightly too large.

The best model is also used in a CFD simulation on the periodic hills domain at $Re = 37,000$, which is roughly triple the Reynolds number used in the training sets. The flow field is compared to experimental data and results in large improvements over the standard $k-\omega$ SST model. This shows that the model is able to generalise to flows outside the training conditions.

The models found by DSR are more complex than the models from the SpaRTA research, these complex models achieve better rewards on the data set and also result in flow fields closer to the high fidelity data. However, complexity should not be the goal when searching augmenting models, as the most complex models give a diverging CFD simulation most often. Also the best performing models in terms of CFD flow field error are not the models that achieved the best fit to the dataset, so achieving a high reward during training is not a guarantee to give a good turbulence model.

It is found that the models that perform best on a single case often achieve quite poor results on the other two test cases. This shows that selecting models based on absolute performance on a single case results in less well generalising modes. To find turbulence models that are more applicable in other domains it is recommended to investigate different metrics for model selection, for example the summed error across test cases. Also it would be interesting to include more types of flows where it is known that RANS is inaccurate, such as a square duct.

DSR is strong in fitting expressions, largely due to the great freedom the expression tree based approach offers. However, when left unconstrained DSR quickly starts sampling incredibly complex expressions, the computational cost of analysing these expressions is very high. Fortunately this complexity is not desired as it is found that the most complex expressions perform poorly in a CFD simulation. When limiting DSR to sample short expression the computational cost is in the order of hours.

Analysis of the individual terms of b_{ij}^{Δ} models, correcting the Reynolds stress anisotropy, showed that the most important variables to model the corrections are functions of the mean rate of rotation. This is an encouraging result confirming that indeed different quantities than the mean rate of strain are required to make the predictions closer to LES or DNS results.

It is expected that significant improvements can be made by including many different nondimensional variables. Currently a base tensor series is used to ensure Galilean invariance, this invariance is crucial in finding good turbulence models. However, the variables to scale these base tensors are exclusively functions of the same two quantities used to construct the base tensor series. For future research it is recommended to investigate using different quantities together with the base tensors to possibly form a better model.

Lastly, the possibility should be explored to find models for both corrections b_{ij}^{Δ} and $\mathcal{P}_k^{\Delta}$ together. In this research the symbolic regression for both corrections is performed completely separate and the resulting models are tested together afterwards. However, when using a model for b_{ij}^{Δ} that does not perfectly match the target data, the required correction $\mathcal{P}_k^{\Delta}$ becomes different. An approach could be to first complete several iterations to find expressions for b_{ij}^{Δ}. The best resulting expression can then be used to update the required correction $\mathcal{P}_k^{\Delta}$. Then several iterations to find a model for the new target $\mathcal{P}_k^{\Delta}$ are completed and the best resulting model is used to update the target for b_{ij}^{Δ}. This combined approach might offer increased flexibility and result in better models.

A

DSR models

The best models selected for each training case are presented below. The models presented in the report have the constants rounded, for some modes this affected results. Therefore here the models are presented with unrounded constants.

PH_{10595}:

$$\mathcal{P}_k^{\Delta} = -0.37684455789889304 \cdot T^{(1)}\bar{S}_{ij} \cdot k \cdot \log(\theta_1)$$
$$\mathcal{P}_k^{\Delta} = T^{(1)}\bar{S}_{ij} \cdot k/(\theta_1 + 0.7565724466440847)$$
$$\mathcal{P}_k^{\Delta} = 2.6023083595866809 \cdot T^{(1)}\bar{S}_{ij} \cdot k \cdot \exp(27.66970208043289 \cdot \theta_2)$$
$$\mathcal{P}_k^{\Delta} = 18.051454372466072 \cdot T^{(1)}\bar{S}_{ij} \cdot (k \cdot (14.732374647837743 \cdot \theta_1 - 0.56383361007307031 \cdot \log(\theta_1) - 2.359457656850785))$$
$$\mathcal{P}_k^{\Delta} = T^{(1)}\bar{S}_{ij} \cdot k \cdot (-\theta_2 + 22.597796272693083 \cdot \exp(-67.66792845742125 \cdot \theta_1 + 67.66792845742125 \cdot \theta_2) + 0.5920784333003265)$$

CD_{12600}:

$$\mathcal{P}_k^{\Delta} = T^{(1)}\bar{S}_{ij} \cdot k$$
$$\mathcal{P}_k^{\Delta} = 1.7275169022127954 \cdot k \cdot (T^{(1)}\bar{S}_{ij} + T^{(3)}\bar{S}_{ij})$$
$$\mathcal{P}_k^{\Delta} = T^{(1)}\bar{S}_{ij} \cdot k/(-\theta_1 - \theta_2 + 0.577507820713581)$$
$$\mathcal{P}_k^{\Delta} = T^{(1)}\bar{S}_{ij} \cdot k \cdot (\theta_1 + 1.6867576888570992)$$
$$\mathcal{P}_k^{\Delta} = -0.4798775496233655 \cdot T^{(1)}\bar{S}_{ij} \cdot k \cdot (2810.7934743004272 \cdot \theta_2 \cdot (\theta_1 - 0.05156590806754547) - 0.15393304737658667/\theta_1)$$
$$\mathcal{P}_k^{\Delta} = T^{(1)}\bar{S}_{ij} \cdot k \cdot (40.107398786683452 - 42.599125945389638 \cdot \exp(-\theta_1) + 0.085782434322780765/\theta_1)$$

$CBFS_{13700}$:

$$\mathcal{P}_k^{\Delta} = k \cdot (T^{(1)}\bar{S}_{ij} - T^{(4)}\bar{S}_{ij})$$
$$\mathcal{P}_k^{\Delta} = 0.09957432942278285 \cdot T^{(1)}\bar{S}_{ij} \cdot k/(\theta_1 - 2 \cdot \theta_2)$$
$$\mathcal{P}_k^{\Delta} = k \cdot (0.032903997080294206 \cdot T^{(1)}\bar{S}_{ij} + T^{(3)}\bar{S}_{ij})/\theta_1$$
$$\mathcal{P}_k^{\Delta} = 0.033132303910860905 \cdot T^{(1)}\bar{S}_{ij} \cdot k \cdot \exp(\theta_2)/\theta_1$$
$$\mathcal{P}_k^{\Delta} = T^{(1)}\bar{S}_{ij} \cdot k/(15.57028770525079 \cdot \theta_1 - 14.57028770525079 \cdot \theta_2)$$

PH_{10595}:

$$\begin{aligned}
b_{ij}^{\Delta} = {} & T^{(1)} \cdot (\exp(85.08913572952108 \cdot \theta_2) - 0.2244596333909864) \\
& + 18.80345287823771 \cdot T^{(2)} \cdot \exp(-43.30366475053411 \cdot \theta_1) + T^{(3)} \cdot \log(\theta_1) \\
& + T^{(4)} \cdot (-\theta_1 - 7.152510051853767)
\end{aligned}$$
$$\begin{aligned}
b_{ij}^{\Delta} = {} & T^{(1)} \cdot (\exp(86.59784324132431 \cdot \theta_2) - 0.223208221372159) \\
& + T^{(2)} \cdot \exp(3.083111823522146 \cdot \exp(-20.882640581074906 \cdot \theta_1)) \\
& - 0.03943153239507558 \cdot T^{(3)}/\theta_1 + T^{(4)} \cdot (-\theta_2 - 5.1701607266800576)
\end{aligned}$$
$$\begin{aligned}
b_{ij}^{\Delta} = {} & T^{(1)} \cdot (1.83406808561933 \cdot \theta_1/(\theta_1 - \theta_2) + 1.83406808561933 \cdot \exp(\theta_2) - 2.814254562181835) \\
& + T^{(2)} \cdot \exp(3.084927423815286 \cdot \exp(\theta_1 \cdot (\theta_2 - 20.936107786965343))) \\
& + T^{(3)} \cdot (\exp(\theta_2) + 1673990.554004833)/(\theta_2 - 1277440.3809665763) \\
& - 4.558459090270554 \cdot T^{(4)}
\end{aligned}$$
$$\begin{aligned}
b_{ij}^{\Delta} = {} & T^{(1)} \cdot (27.175908315376077 \cdot \theta_2 + 0.5692800486621243) \\
& + T^{(2)} \cdot (-5.743591548588832 \cdot \log(\theta_1) - 14.744990524412433) \\
& + T^{(3)} \cdot (-629.70730243722663 \cdot \theta_2 - 6.752487071734653) \\
& + T^{(4)} \cdot (-452.805863404235 \cdot \theta_2 - 6.666870421943138)
\end{aligned}$$
$$\begin{aligned}
b_{ij}^{\Delta} = {} & T^{(1)} \cdot (\exp(97.120244731119394 \cdot \theta_2) - 0.1889405179956901) \\
& + 18.814060803694183 \cdot T^{(2)} \cdot \exp(-43.346201193836826 \cdot \theta_1) \\
& - 2.5531963938366067 \cdot T^{(3)} \cdot \exp(-\theta_1) + T^{(4)} \cdot (\theta_2 - 5.818816615302255)
\end{aligned}$$

CD_{12600}:

$$
\begin{aligned}
b_{ij}^{\Delta} &= -0.08423103723284284 \cdot T^{(1)} \cdot \log(\theta_1) - 1.5132223168251326 \cdot T^{(2)} \cdot \log(\theta_1) \\
&\quad + 0.12392696207225239 \cdot T^{(3)}/\theta_1 + T^{(4)} \cdot (\theta_1 - 1.6533175958880926) \\
b_{ij}^{\Delta} &= T^{(1)} \cdot \theta_1 \cdot \cdot 2 - 1.5128915383762416 \cdot T^{(2)} \cdot \log(\theta_1) \\
&\quad + 0.1262147210520074 \cdot T^{(3)}/\theta_1 + T^{(4)} \cdot (-\theta_1 - 1.5560423180640641) \\
b_{ij}^{\Delta} &= 1.0649267617025036 \cdot T^{(1)} \cdot \exp(109.95727633921847 \cdot \theta_2) \\
&\quad + 22.152799466725642 \cdot T^{(2)} \cdot \exp(-41.23121471537061 \cdot \theta_1) \\
&\quad + 2.935179197934824e - 12 \cdot T^{(3)} \cdot \exp(\theta_1 - \theta_2) \\
&\quad + T^{(4)} \cdot (\theta_1 + 0.12978268312102945)/\theta_2 \\
b_{ij}^{\Delta} &= T^{(1)} \cdot (\theta_1 + 0.27681424678042454) \\
&\quad + 22.19400638723753 \cdot T^{(2)} \cdot \exp(-41.27572381660118 \cdot \theta_1) \\
&\quad + T^{(3)} \cdot (\theta_1 + 1.6468722206058466) + T^{(4)} \cdot (\theta_1 + 0.10944780518422013/\theta_2) \\
b_{ij}^{\Delta} &= T^{(1)} \cdot (\theta_1 + \theta_2) - 1.5132490325945913 \cdot T^{(2)} \cdot \log(\theta_1) \\
&\quad + 0.128376245817644 \cdot T^{(3)}/\theta_1 + T^{(4)} \cdot (\theta_2 - 1.4640984115740243)
\end{aligned}
$$

$CBFS_{13700}$:

$$
\begin{aligned}
b_{ij}^{\Delta} = {} & T^{(1)} \cdot (\exp(241.91340658225613 \cdot \theta_2) - 0.05786658037047583) \\
& + 0.11820804966786677 \cdot T^{(2)}/(\theta_1 + 0.0017010173575316813) \\
& + 0.45652636638908267 \cdot T^{(3)}/\theta_1 \\
& + T^{(4)} \cdot (\theta_2 + 9.113026170193217) \\
b_{ij}^{\Delta} = {} & T^{(1)} \cdot (19.91177166761797 \cdot \theta_2 + 0.644362176055794) \\
& + T^{(2)} \cdot (1.5407641804996342 + 0.06153920245018529/\theta_1) \\
& + 0.6485550629394776 \cdot T^{(3)}/\theta_1 \\
& + 0.5454224507031913 \cdot T^{(4)}/(\theta_1 + 0.0024250909467806903) \\
b_{ij}^{\Delta} = {} & T^{(1)} \cdot (20.046103872245954 \cdot \theta_2 + 0.6481824213970493) \\
& + 0.09283364876264767 \cdot T^{(2)}/\theta_1 \\
& + 0.770480309995562 \cdot T^{(3)}/\theta_1 \\
& + 0.6542936497207333 \cdot T^{(4)}/(0.001008120655609965 - \theta_2) \\
b_{ij}^{\Delta} = {} & T^{(1)} \cdot (19.85899850773482 \cdot \theta_2 + 0.6443120171410456) \\
& + T^{(2)} \cdot (\theta_1 + 3.418073904104651) \\
& + 0.7695851692041502 \cdot T^{(3)}/\theta_1 \\
& - 0.6640964740642666 \cdot T^{(4)}/\log(\theta_2 + 0.9989492319161185) \\
b_{ij}^{\Delta} = {} & T^{(1)} \cdot (19.775665890168218 \cdot \theta_2 + 0.63864133375829145) \\
& + T^{(2)} \cdot (1.5488932191198208 + 0.061271579985459956/\theta_1) \\
& + T^{(3)} \cdot (7.636012458330098 + 0.028950675541348416/\theta_1) \\
& + T^{(4)} \cdot (4.575692905324675 - 0.0008300841758839189/(\theta_2 \cdot (\theta_2 - 0.0011657333461313382)))
\end{aligned}
$$

B

Algorithmic differentiation in reverse mode operations

For readability here the adjoint variable notation is used, where r is the final value that the derivatives are determined for.:

$$\bar{v}_i = \frac{\partial r}{\partial v_i} \tag{91}$$

Furthermore mathematical operations of arity two, meaning they work on two children, have a left and right child, which are the subscripts lc and rc. For unary operators that have only one child, it is chosen to denote those having a single left child and no right child. See the tokens, operations and their derivative updates below:

Token	Operation	AD update
$v_{\text{mul}} =$	$v_{\text{lc}} \cdot v_{\text{rc}}$	$\bar{v}_{\text{lc}} = \bar{v}_{\text{mul}} \cdot v_{\text{rc}}$
		$\bar{v}_{\text{rc}} = \bar{v}_{\text{mul}} \cdot v_{\text{lc}}$
$v_{\text{div}} =$	$v_{\text{lc}}/v_{\text{rc}}$	$\bar{v}_{\text{lc}} = \bar{v}_{\text{div}}/v_{\text{rc}}$
		$\bar{v}_{\text{rc}} = \bar{v}_{\text{div}} \cdot (-v_{\text{div}}/v_{\text{rc}})$
$v_{\text{add}} =$	$v_{\text{lc}} + v_{\text{rc}}$	$\bar{v}_{\text{lc}} = \bar{v}_{\text{add}}$
		$\bar{v}_{\text{rc}} = \bar{v}_{\text{add}}$
$v_{\text{sub}} =$	$v_{\text{lc}} - v_{\text{rc}}$	$\bar{v}_{\text{lc}} = \bar{v}_{\text{sub}}$
		$\bar{v}_{\text{rc}} = -\bar{v}_{\text{sub}}$
$v_{\text{exp}} =$	$\exp(v_{\text{lc}})$	$\bar{v}_{\text{lc}} = \bar{v}_{\text{exp}} \cdot v_{\text{exp}}$
$v_{\text{log}} =$	$\log(v_{\text{lc}})$	$\bar{v}_{\text{lc}} = \bar{v}_{\text{log}} \cdot (1/v_{\text{lc}})$
$v_{\text{sum}} =$	$\sum(v_{\text{lc}})$	$\bar{v}_{\text{lc}} = \bar{v}_{\text{sum}}$
$v_{\text{n2}} =$	$(v_{\text{lc}})^2$	$\bar{v}_{\text{lc}} = \bar{v}_{\text{n2}} \cdot 2v_{\text{lc}}$
$v_{\text{sqrt}} =$	$\sqrt{v_{\text{lc}}}$	$\bar{v}_{\text{lc}} = \bar{v}_{\text{sqrt}} \cdot (1/2\sqrt{v_{\text{lc}}})$

C

$k-\omega$ SST model constants and blending functions

The constants in set 1, ϕ_1, are:

$$\begin{gathered}\sigma_{k1}=0.85, \quad \sigma_{\omega 1}=0.5, \quad \beta^*=0.09, \quad \beta_1=0.0750 \\ \kappa=0.41, \quad \gamma_1=\frac{\beta_1}{\beta^*}-\sigma_{\omega 1}\frac{\kappa^2}{\sqrt{\beta^*}}\end{gathered} \tag{92}$$

The constants in set 2, ϕ_2, are:

$$\begin{gathered}\sigma_{k2}=1, \quad \sigma_{\omega 2}=0.856, \quad \beta^*=0.09, \quad \beta_2=0.0828 \\ \kappa=0.41, \quad \gamma_2=\frac{\beta_2}{\beta^*}-\sigma_{\omega 2}\frac{\kappa^2}{\sqrt{\beta^*}}\end{gathered} \tag{93}$$

The blending functions are given by:

$$F1=\tanh\left(\arg_1^4\right) \tag{94}$$

$$F2=\tanh\left(\arg_2^2\right) \tag{95}$$

Where:

$$\arg_1=\min\left[\max\left(\frac{\sqrt{k}}{0.09\omega y},\frac{500\nu}{y^2\omega}\right),\frac{4\rho\sigma_{\omega 2}k}{CD_{k\omega}y^2}\right] \tag{96}$$

$$\arg_2=\max\left(2\frac{\sqrt{k}}{0.09\omega y},\frac{500\nu}{y^2\omega}\right) \tag{97}$$

$$CD_{k\omega}=\max\left(2\rho\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_i}\frac{\partial \omega}{\partial x_i},10^{-10}\right) \tag{98}$$

Finally, y in the equations above is the distance of the centroid of each cell to the nearest wall.

D

Base tensor series and invariants

The base tensor series as introduced by Pope [33], with 10 tensors $T^{(m)}$ and 5 invariants θ_n. Note that in this section S denotes the mean rate of strain $\bar{S}$ and R the mean rate of rotation $\bar{R}$, for readability the bar is omitted.

$$
\begin{aligned}
T^{(1)} &= S \\
T^{(2)} &= SR - RS \\
T^{(3)} &= S^2 - \tfrac{1}{3} I \cdot \mathrm{trace}(S^2) \\
T^{(4)} &= R^2 - \tfrac{1}{3} I \cdot \mathrm{trace}(R^2) \\
T^{(5)} &= RS^2 - S^2 R
\end{aligned}
\qquad
\begin{aligned}
T^{(6)} &= R^2 S + SR^2 - \tfrac{2}{3} I \cdot \mathrm{trace}(SR^2) \\
T^{(7)} &= RSR^2 - R^2 SR \\
T^{(8)} &= SRS^2 - S^2 RS \\
T^{(9)} &= R^2 S^2 + S^2 R^2 - \tfrac{2}{3} I \cdot \mathrm{trace}(S^2 R^2) \\
T^{(10)} &= RS^2 R^2 - R^2 S^2 R
\end{aligned}
$$

$$
\begin{aligned}
\theta_1 &= \mathrm{trace}(S^2) \\
\theta_2 &= \mathrm{trace}(R^2) \\
\theta_3 &= \mathrm{trace}(S^3) \\
\theta_4 &= \mathrm{trace}(R^2 S) \\
\theta_5 &= \mathrm{trace}(R^2 S^2)
\end{aligned}
$$

E

Gene Expression Programming

A popular method used in data driven turbulence modelling to produce explicit algebraic expressions is gene expression programming (GEP). GEP is a genetic algorithm that uses populations of individuals and creates new populations based on the fitness of the individuals and genetic modifications of those individuals. In that sense is it very similar to more traditional genetic programming (GP), however there is more freedom to produce expressions in GEP and the method is robust against producing mathematically invalid expressions. The method detailed below is GEP as first introduced by Ferreira [43].

In GEP, the genotype or chromosome is an expression in the population and is expressed as a string of fixed length. Using pre-order traversal an expression tree (ET) is constructed from the genotype, these ETs are called the phenotype. The expression trees are evaluated for their fitness, the best expressions in a population are selected and modified through genetic operators to create the next population.

E.1 Chromosome Representation

The chromosome is a string built up out of tokens, these tokens represent a mathematical operator or a variable, an example could be:

$$\begin{matrix} \texttt{01234567} \\ \texttt{*e-2y+4x} \end{matrix} \tag{99}$$

The top string is the index of each token, the bottom string is the chromosome. Each token, or symbol in the chromosome has a certain arity, the arity is how many child nodes a certain token has. For example the plus operator + has an arity of 2 since two values are added, an exponent function has arity 1 and a variable is a terminal token with arity 0. Using this arity and pre-order traversal a unique expression tree is constructed, pre-order traversal means visiting each node of the tree depth first from left to right. The expression tree denoted in equation 99 can be seen in figure 36, note that `e` denotes the exponent function. From the ET it is clear this chromosome corresponds to the following equation: $\exp(2-y)(4+x)$.

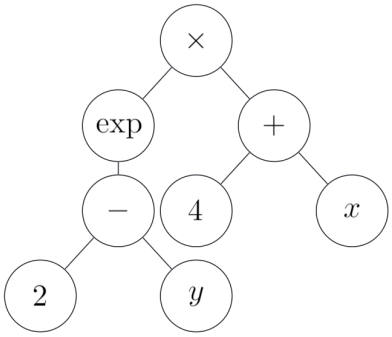


Figure 36: Example expression tree for the chromosome in equation 99, image adapted from [37]

To ensure only valid equations are produced, the gene is constructed out of a head and a tail of fixed length. The head can contain functions and variables, the tail can contain only variables. For a head of length l_h the length of the tail l_t is calculated as follows: $l_t = l_h(a_{max} - 1) + 1$. Here a_{max} is the maximum arity of functions in the function set. The tail of this length ensures there are always enough terminal tokens to complete an expression tree, however a tree can also be completed early without using the full gene. Consider the following examples with function set $\{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{x}, \texttt{y}\}$ and a head of length 3, $a_{max} = 2$, so the length of the tail is 4. `|` denotes the split between head and tail:

$$\begin{array}{l} \texttt{012 0123} \\ \texttt{+*-|}\mathbf{\texttt{xyxx}} \end{array} \tag{100}$$

Which gives the following expression tree:

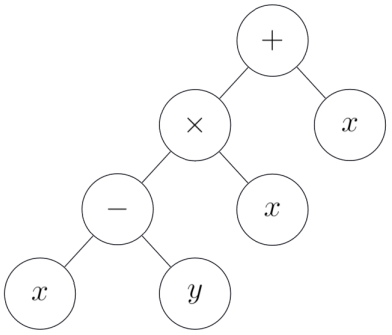


Figure 37: Expression tree for the chromosome in equation 100.

Consider the second token in the head is changed to y, the gene and expression tree then become:

$$\begin{array}{l} \texttt{012 0123} \\ \texttt{+y-|}\mathbf{\texttt{xyxx}} \end{array} \tag{101}$$

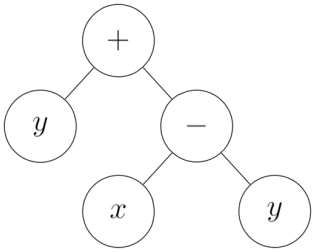


Figure 38: Expression tree for the chromosome in equation 101.

In the latter case the tree is completed without using all tokens in the gene. Also it shows that a single mutation in the head can result in a drastic change of the tree structure. A single string of head and tail is called a gene, in GEP a chromosome is often constructed of more than one genes. Using functions like addition or multiplication these sub-genes can be combined, consider the following chromosome:

$$\begin{array}{l} \texttt{012 3456 012 3456} \\ \texttt{+*-|xyxx +y-|xyxx} \end{array} \tag{102}$$

The chromosome above is a two-gene chromosome composed of the two genes in equations 100 and 101 with their ETs in figures 37 and 39 respectively. If the gene linking function is addition, the final expression tree would be:

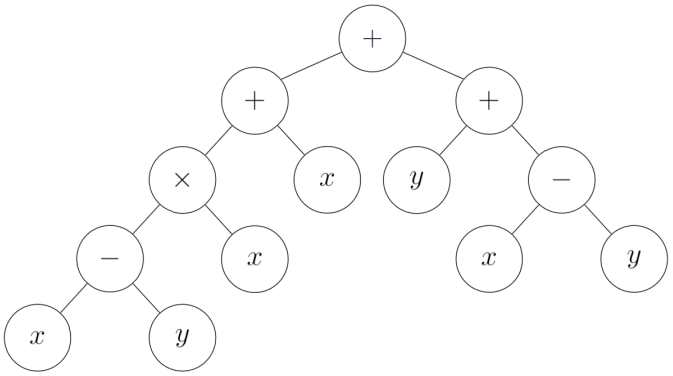


Figure 39: Expression tree for the chromosome in equation 102, constructed with two sub genes, linked by addition.

In the ET above the linking function addition could also be changed to multiplication, division or any other function with two arguments. This multi-gene representation used in GEP allows for great complexity in the resulting expressions. Also when constructing genes as above, each gene will always produce a syntactically correct expression tree, allowing great flexibility in how the genes can be modified. In traditional GP many restrictions are required in the modification process to ensure correct expressions, that problem is solved by using GEP.

E.2 Selection

For each expression the fitness is evaluated with a cost function, this cost function could be the RMS as in equation 28. The best expressions in the population are selected to reproduce with modification. There are multiple algorithms to select the best performers, Ferreira reports that for complex problems roulette wheel selection with elitism works best. Elitism means that the cloning of the best expression in the population is guaranteed. Roulette wheel selection means that expressions with high fitness have higher probability to be selected for reproduction [59], so fitter individuals will produce more offspring.

E.3 Genetic Modification

Each time an expression is selected for the next generation it is copied directly or modified through several genetic operators. These genetic operators introduce variation in the population and help evolve the expressions to maximise fitness. The genetic operators introduced by Ferreira are detailed below:

Replication
Replication means simply copying the chromosome without modification. If all expressions would be modified heavily the best performers could be lost, thus also some are just copied. This operator does not introduce genetic diversification.

Mutation
During mutation one or more tokens are randomly selected in the chromosome and then changed. It is important to ensure the head and tail structure stays intact to ensure the result is a valid expression. So when a token is replaced in the head of a gene it can become anything, if it is replaced in the tail it can only become a terminal token. See equation 100 and 101 and their ETs for an example of mutation. The `*` token on position `1` in equation 100 is replaced by a `y` token in equation 101.

Transposition
In transposition a string of multiple tokens of random length and location is selected in the chromosome and then inserted into a random location in the head of the gene. Consider the example below, a chromosome consisting of two genes with heads of length 10:

```
0123456789012345678901234567890 0123456789012345678901234567890
*-+*x-+x*yyxyyxxyxyxy e**+xyeyy*xxyyxx**xxy**yx
```
(103)

Suppose the sequence `xxy` in positions 16 through 18 in the second gene is selected to be inserted. It will be inserted in position `6` of the head of gene 1, thus between position `5` and `6`. This sequence is then copied and inserted, to ensure the expression is still valid the last tokens of the head where the sequence is inserted are deleted. In this case that

means the sequence x*y at the end of the head of gene 1 is deleted, resulting in:

```
012345678901234567890012345678901234567890
*-+*x-xxy+yxyyxxyxyxye**+xyeyy*xxyyxxxxyyx
```
(104)

Root Transposition
Similar to transposition, however the sequence is moved towards the front of the gene. Because terminal tokens at the first position are not allowed this means the sequence to be inserted must start with a function, and thus is selected from the head.

Gene Transposition
In gene transposition a whole gene in a multi-gene chromosome is randomly selected and moved to the front of the chromosome. Note that if the gene linking function is addition, the result is unchanged.

One Point Recombination
In recombination operations two genes are paired and a recombination point is chosen randomly. The tokens downstream of this recombination point are exchanged between the two chromosomes. Consider the pair of three-gene chromosomes below:

```
012345678012345678012345678
/xx-xyxxx/x*yyxxxy/e*+xxxxy
/-*/xyyxye+xeyxyxx-e/eyxxyx
```
(105)

If the recombination point is position 6 of gene 1, the resulting two chromosomes are:

```
012345678012345678012345678
/xx-xyyxye+xeyxyxx-e/eyxxyx
/-*/xyxxx/x*yyxxxy/e*+xxxxy
```
(106)

Two Point Recombination
Similar to one point recombination, but the genes are swapped twice in two randomly chosen recombination points. Consider the pair of genes in equation 105, the recombination points 6 in gene 1 and 8 in gene 2 give:

```
012345678012345678012345678
/xx-xyyxye+xeyxyxy/e*+xxxxy
/-*/xyxxx/x*yyxxxx-e/eyxxyx
```
(107)

Note that because of the fixed structure of chromosomes all recombinations are always valid expressions.

Gene Recombination
In gene recombination again two chromosomes are paired, these then swap a whole gene. Using the pair of chromosomes in equation 105, selecting gene 2 for gene recombination would result in:

$$\begin{array}{l} \texttt{012345678012345678012345678} \\ \texttt{/xx-xyxxxe+xeyxyxx/e*+xxxxy} \\ \texttt{/-*/xyyxy/x*yyxxxy-e/eyxxyx} \end{array} \tag{108}$$

With these genetic operators the new population consists of the best fitting expressions and modifications of those expressions. After forming a new population the fitness is evaluated and this is then used to form the next generation. In a process loosely resembling natural selection and evolution in nature the result after several iterations is the best fitting expression.

References

[1] M. Schmeltzer, R.P. Dwight, and P. Cinnella. "Discovery of Algebraic Reynolds-Stress Models Using Sparse Symbolic Regression". In: *Flow, Turbulence and Combustion* 104 (2020), pp. 579–603.

[2] B.K. Petersen et al. "Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients". In: *International Conference on Learning Representations* (2020).

[3] W. Rodi. "Comparison of LES and RANS calculations of the flow around bluff bodies". In: *Journal of Wind Engineering and Industrial Aerodynamics* 69-71 (1997), pp. 55–75.

[4] S.B. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

[5] A. Tsinober. *An Informal Introduction to Turbulence*. Fluid Mechanics and its Applications. Kluwer Academic Publisher, 2004.

[6] S.J. Hulshoff. *CFD II Part 2: Computation and Modelling of Turbulence*. Faculty of Aerospace Engineering, TU Delft. 2015.

[7] L.F. Richardson. *Weather Prediction by Numerical Process*. Cambridge University Press, 1922.

[8] A. Kolmogorov. "The Local Structure of Turbulence in Incompressible Viscous Fluid for Very Large Reynolds' Numbers". In: *Doklady Akademiia Nauk SSSR* 30 (1941), pp. 301–305.

[9] P.E. Dimotakis. "The Mixing Transition in Turbulent Flows". In: *Journal of Fluid Mechanics* 409 (2000), pp. 69–98.

[10] J.P. Slotnick et al. *CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences*. Technical Report NASA/CR-2014-21878. NASA, 2014.

[11] B.E. Launder and D.B. Spalding. “The Numerical Computation of Turbulent Flows”. In: *Computer Methods in Applied Mechanics and Engineering* 3 (1974), 269–289.

[12] W. Rodi and G. Scheuerer. “Scrutinizing the k-ϵ Turbulence Model Under Adverse Pressure Gradient Conditions”. In: *Journal of Fluids Engineering* 108(2) (1986), p. 174.

[13] W. Rodi and G. Scheuerer. “Influence of Freestream Values on k-ω Turbulence Model Predictions”. In: *AIAA Journal* 30(6) (1992), pp. 1657–1659.

[14] D.C. Wilcox. “Reassessment of the Scale-Determining Equation for Advanced Turbulence Models”. In: *AIAA Journal* 26(11) (1988), p. 1299.

[15] Z. Warhaft. “An experimental study of the effect of uniform strain on thermal fluctuations in grid-generated turbulence”. In: *Journal of Fluid Mechanics* 99(3) (1980), pp. 545–573.

[16] I.J. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[17] B. Ding, H. Qian, and J. Zhou. “Activation functions and their characteristics in deep neural networks”. In: *Chinese Control And Decision Conference* (2018).

[18] Y. LeCun, Y. Bengio, and G. Hinton. “Deep Learning”. In: *Nature* 521 (2015), pp. 436–444.

[19] A. Griewank and A. Walther. *Evaluating Derivatives : principles and techniques of algorithmic differentiation*. Vol. 2. Society for Industrial and Applied Mathematics, 2008.

[20] K. Hornik. “Approximation capabilities of multilayer feedforward networks”. In: *Neural Networks* 4 (1991), pp. 251–257.

[21] C. Olah. *Understanding LSTM Networks*. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Visited 14-2-2022. 2015.

[22] Y. Bengio, P. Simard, and P. Frasconi. “Learning Long-Term Dependencies with Gradient Descent is Difficult”. In: *IEEE Transactions on Neural Networks* 5(2) (1994), pp. 157–166.

[23] Y. Bengio, P. Simard, and P. Frasconi. “Long Short-Term Memory”. In: *Neural Computation* 9(8) (1997), pp. 1735–1780.

[24] R. Jozefowicz, W. Zaremba, and I. Sutskever. “An Empirical Exploration of Recurrent Network Architectures”. In: *Proceedings of the 32^{nd} International Conference on Machine Learning* 37 (2015), 2342–2350.

[25] F.A. Gers, J. Schmidhuber, and F. Cummins. “Learning to Forget: Continual Prediction with LSTM”. In: *Neural Computation* 12 (2000).

[26] G.D. Salton and J.D. Kelleher. “Persistence pays off: Paying Attention to What the LSTM Gating Mechanism Persists”. In: *Proceedings of Recent Advances in Natural Language Processing* (2018).

[27] W.N. Edeling, P. Cinnella, and R.P. Dwight. “Predictive RANS simulations via Bayesian Model-Scenario Averaging”. In: *Journal of Computational Physics* 275 (2014), pp. 65–91.

[28] J. Ray et al. “Bayesian calibration of a $k - \epsilon$ turbulence model for predictive jet-in-crossflow simulations”. In: *AIAA Fluid Dynamics Conference* 44 (2014).

[29] J. Ray et al. “Robust Bayesian Calibration of a $k - \epsilon$ Model for Compressible Jet-in-Crossflow Simulations”. In: *AIAA Journal* 56 (2018).

[30] B. Fabritius. “Application of Genetic Algorithms to Problems in Computational Fluid Dynamics”. PhD thesis. University of Exeter, 2014.

[31] J. Ling et al. “Uncertainty Analysis and Data-Driven Model advances for a Jet-in-crossflow”. In: *ASME Turbomachinery Technical Conference and Exposition* (2016).

[32] J. Ling, J. Templeton, and A. Kurzawski. “Reynolds Averaged Turbulence Modeling using Deep Neural Networks with Embedded Invariance”. In: *Journal of Fluid Mechanics* 807 (2016), pp. 155 –166.

[33] S.B. Pope. “A more general effective-viscosity hypothesis”. In: *Journal of Fluid Mechanics* 72(2) (1975), pp. 331–340.

[34] M.L.A. Kaandorp and R.P. Dwight. “Data-driven modelling of the Reynolds stress tensor using random forests with invariance”. In: *Computers and Fluids* 202 (2020).

[35] M. Raissi, G. Karniadakis, and P. Perdikaris. “Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations”. In: *arXiv preprint: arXiv:1711.10566* (2017).

[36] S.L. Brunton, J.L. Proctor, and J.N. Kutz. “Discovering governing equations from data by sparse identification of nonlinear dynamical systems”. In: *Proceedings of the National Academy of Sciences* 113(15) (2016), pp. 3932–3937.

[37] J. Weatheritt and R.D. Sandberg. “A novel evolutionary algorithm applied to algebraic modifications of the RANS stress-strain relationship”. In: *Journal of Computational Physics* 325 (2016), pp. 22–37.

[38] J. Weatheritt and R.D. Sandberg. “The development of algebraic stress models using a novel evolutionary algorithm”. In: *International Journal of Heat and Fluid Flow* 68 (2017), pp. 298–318.

[39] R.L. Thompson et al. “A methodology to evaluate statistical errors in DNS data of plane channel flows”. In: *Computers and Fluids* 130 (2016), pp. 1–7.

[40] J. Zhong, L. Feng, and J.S. Ong. “Gene Expression Programming: A Survey”. In: *IEEE Computational Intelligence Magazine* 12-3 (2017), pp. 54–72.

[41] H. Hmida et al. “Scale Genetic Programming for large Data Sets: Case of Higgs Bosons Classification”. In: *Procedia Computer Science* 126 (2018), pp. 302–311.

[42] M.A. Haeri, M.M. Ebadzadeh, and G. Folino. “Statistical Genetic Programming for Symbolic Regression”. In: *Applied Soft Computing* 60 (2017), pp. 447–469.

[43] C. Ferreira. “Gene Expression Programming: A New Adaptive Algorithm for Solving Problems”. In: *Complex Systems* 13 (2001), pp. 87–129.

[44] H. Zhang and A. Zhou. “RL-GEP: Symbolic Regression via Gene Expression Programming and Reinforcement Learning”. In: *2021 International Joint Conference on Neural Networks* (2021).

[45] J. Zhong, Y. Ong, and W. Cai. “Self-Learning Gene Expression Programming”. In: *IEEE Transactions on Evolutionary Computation* 14 (2016).

[46] M. Kommenda et al. “Effects of Constant Optimization by Nonlinear Least Squares Minimization in Symbolic Regression”. In: *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation* (2013), 1121–1128.

[47] P. Domingos. “A Few Useful Things to Know About Machine Learning”. In: *Communications of the ACM* 55(10) (2012), pp. 78–87.

[48] S.M. Udrescu and M. Tegmark. “AI Feynman: a physics-inspired method for symbolic regression.” In: *Science Advances* 6 (2020).

[49] J. Nocedal and S.J. Wright. *Numerical Optimization*. Second. Springer, 2006.

[50] R.J. Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning* 2 (1992), pp. 229–256.

[51] A. Rajeswaran et al. “EPOpt: Learning Robust Neural Network Policies Using Model Ensembles”. In: *International Conference on Learning Representations* (2016).

[52] C.E. Shannon. “A Mathematical Theory of Communication”. In: *Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[53] J. Bergstra and Y. Bengio. “Random Search for Hyper-Parameter Optimization”. In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305.

[54] M. Breuer et al. “Flow over periodic hills – Numerical and experimental study in a wide range of Reynolds numbers”. In: *Computers and Fluids* 38 (2009), pp. 433–457.

[55] C. Rapp and M. Manhart. “Flow over periodic hills: an experimental study”. In: *Experiments in Fluids* 51 (2011), 247–269.

[56] J.P. Laval and M. Marquillie. “Direct Numerical Simulations of Converging–Diverging Channel Flow”. In: *Progress in Wall Turbulence: Understanding and Modeling* (2011), pp. 203–209.

[57] S. Jakirlic. *Extended excerpt related to the test case: “Flow over a periodical arrangement of 2D hills”*. Tech. rep. TU Darmstadt, 2012.

[58] Y.J. Jang et al. “Investigation of Anisotropy-Resolving Turbulence Models by Reference to Highly-Resolved LES Data for Separated Flow”. In: *Flow Turbulence and Combustion* 69 (2002), pp. 161–203.

[59] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.