Machine Learning for RANS Turbulence Modelling of Variable Property Flows

Rafael Diez

Technische Universiteit Delft

TUDelft Delft University of Technology

Challenge the future

Machine Learning for RANS Turbulence Modelling of Variable Property Flows

by

Rafael Diez

in partial fulfillment of the requirements for the degree of

Master of Science
in Mechanical Engineering

at the Delft University of Technology,
to be defended publicly on Friday September 14, 2018 at 10:00 AM.

Supervisor:	Prof. dr. ir. R. Pecnik	
Thesis committee:	Prof. dr. ir. B. J. Boersma,	TU Delft
	Prof. dr. S. Hickel,	TU Delft
	Prof. dr. ir. M. J. B. M. Pourquie,	TU Delft

Abstract

Turbulence modelling corresponds to one of the greatest unsolved problems in physics and mathematics. This phenomenon is marked by the emergence of chaotic vortex structures in the solution of the Navier-Stokes equations, and it corresponds to the leading-order effect in the majority of the flows observed in nature. Due to the importance of turbulence modelling, researchers have designed RANS (Reynolds-Averaged Navier Stokes) turbulence models to understand their mean flow behavior. However, one important limitation present in traditional RANS turbulence models is given by their focus on isothermal incompressible fluids, which present constant molecular properties. In order to overcome these limitations, the research previously done at the Process & Energy Department of the TU Delft has established new scaling theories for variable-property flows, which bring the greatest degree of universal collapse observed to date. The application of these scaling theories to traditional RANS turbulence models has greatly improved their performance, although multiple-equation models still require further tuning.

The present thesis work thus consisted in applying the principles of Machine Learning (ML) to build improved data-driven RANS turbulence models; using the DNS database assembled by the research group. Special emphasis was placed in an emerging technique called FIML (Field Inversion Machine Learning). This technique splits the process of building ML corrections into an initial data mining stage known as Field Inversion, and a subsequent Machine Learning stage where a predictive system is trained using the corrections found.

During the present study, several advances were proposed for the previous methodology. The first contribution is given by the synthesis of a direct Hessian-free Field Inversion optimizer, which continuously seeks the best learning hyper-parameters to use while remaining unconditionally stable. As a result, the Field Inversion optimization process can be fully automatized for any RANS turbulence model. This marked a large contrast with respect to the Field Inversion optimizers employed in leading publications, where probabilistic Bayesian inversion algorithms are used. These algorithms can result unnecessarily complex, since they perform a side-study of the probabilities associated to obtaining certain parameters, whereas the FIML approach only requires the final (or average) parameter distributions chosen. Beyond the previous discussion, performing a rigorous probabilistic analysis of a Field Inversion problem requires an expensive technique called MCMC (Markov-Chain Monte Carlo) sampling, which results intractable in large-scale systems. As a result, the probabilistic analysis performed by standard Field Inversion optimizers is based on strong Gaussian assumptions, whose effect cannot be immediately quantified.

The overall Field Inversion methodology developed during the present work was further enhanced by creating a Sympy script to derive the algebraic expressions required to perform Field Inversion, which can be imported into any programming language. Thanks to this methodology, the running times registered in Python were improved by a factor of x1000 compared to the use of Automatic Differentiation libraries. Additionally, a strategy was created to build explicit source term corrections ($\boldsymbol{\delta}$) for any RANS turbulence model without compromising the numerical stability of the optimizers. This innovation allowed Field Inversion optimizers to perform an unbiased analysis of any governing equation.

Regarding the Neural Network systems built, the theoretical review performed revealed that logarithmic neurons could be added to the first layer of a Neural Network, such that the optimizer was able to identify the best parameter groups available. Important parameters can be found in this category, such as the Reynolds number, the Prandtl number, or any intermediate combinations (e.g., $Re^{a}Pr^{b}$). As a result, it was possible to build Neural Networks which could automatically discover adequate input features for the subsequent hidden layers, and the overall modelling process was greatly simplified. Furthermore, the present thesis work includes the derivation of a new intelligent relaxation factor methodology, which was created to eliminate any spurious corrections predicted by Neural Networks that may cause divergence. The methodology created scales well to large-scale systems, and it was able to perform an intelligent selection of the best corrections to keep in less than a millisecond during the tests performed.

The final Field Inversion study centered in the Spalart-Allmaras (SA) and the MK turbulence models, since these models presented the lowest and the highest initial modelling errors respectively. The Field Inversion corrections obtained for the SA turbulence model presented a high degree of over-fitting and inconsistent patterns, which did not form a good basis to train Neural Networks. These issues were attributed to the high initial accuracy exhibited by the SA model, which implied that improvements could only be achieved through direct over-fitting. Similar trends were obtained during a secondary study performed for the turbulent Prandtl number (Pr_T), which also presented inconsistent trends marginally deviating from $Pr_T \approx 1$. However, the Field Inversion corrections built for the MK turbulence model presented consistent trends, which were suitable for a Machine Learning study. After an exhaustive analysis, it was detected that the predictions made by Neural Networks for the source terms required by the k-equation of the MK turbulence model resulted more stable, and that the use of the intelligent relaxation factor methodology corresponded to an important asset. The FIML study for the MK turbulence model thus concluded with satisfactory results.

From a global perspective, it was detected that the Field Inversion corrections built for RANS turbulence models presented a high degree of variability, which seems to be caused by their numerical properties instead of the actual flow physics. This insight was based on the derivation of a new turbulence parameter for rectangular channel flows, which was able to collapse the turbulence statistics of all the DNS flows studied into similar trajectory curves. As a result, it was finally concluded that the FIML methodology developed can be employed to perform successful Machine Learning studies, but also that new turbulence parameters should be developed in order to create more reliable predictive systems in the future.

Contents

List of Figures **vii**

List of Tables **xi**

Acknowledgements **xiii**

1 Introduction **1**

1.1 Overview 1

1.2 Thesis Objectives and Contributions 2

1.3 Thesis Outline 3

2 Theoretical Background **5**

2.1 Fluid Mechanics 5

2.1.1 Navier-Stokes Equations 5

2.1.2 Energy Equation 6

2.1.3 Dimensionless Form of the Governing Equations 8

2.1.4 Reynolds-Averaged Navier-Stokes Equations 11

2.1.5 Alternatives to RANS Turbulence Modelling 13

2.1.6 Boundary Layer Structures 15

2.1.7 RANS Turbulence Models 17

2.2 Parameter Optimization 21

2.2.1 General Paradigm 21

2.2.2 Gradient-based Optimization 22

2.2.3 Computation of the Jacobian Matrix 23

2.2.4 Hessian Optimization 25

2.2.5 Jacobian Optimization Algorithms 27

2.3 Machine Learning Applied to CFD 30

2.3.1 Neural Networks 31

2.3.2 Convolutional Neural Networks 33

2.3.3 Long Short Term Memory Neural Networks 34

2.3.4 Logarithmic Neural Networks 35

2.3.5 Model Selection 36

2.3.6 Other Machine Learning Techniques 37

2.3.7 Field Inversion Machine Learning 39

3 Preliminary Examples **41**

3.1 Field Inversion for a 1-D Radiative Heat Transfer Problem 41

3.2 Recovery of a Laminar Viscosity Profile 44

3.3 Field Inversion Optimization for the k-ω Turbulence Model 46

3.4 Deep Dreaming the Existence of the Reynolds Number 49

4 Data-Driven RANS Turbulence Modelling for Variable Property Flows **53**

4.1 RANS Equations Applied to Channel Flows 53

4.2 DNS Database Assembled 54

4.2.1 Performance of Traditional RANS Turbulence Models in the Dataset 56

4.3 Field Inversion Optimization 58

4.3.1 Field Inversion Formulation for the MK Turbulence Model 58

4.3.2 Field Inversion Optimization for the Spalart-Allmaras Turbulence Model . 60

4.3.3 Field Inversion Study for the Turbulent Prandtl Number 61

4.4 Field Inversion Optimization Results . . . 62
4.4.1 MK Turbulence Model . . . 62
4.4.2 Field Inversion Results for the SA turbulence model . . . 70
4.4.3 Field Inversion Results for the Turbulent Prandtl Number . . . 73
4.5 Machine Learning Systems . . . 75
4.5.1 Derivation of an Intelligent Relaxation Factor . . . 75
4.5.2 Neural Network Architectures . . . 78
4.6 Final Remarks . . . 85

5 Alternative Machine Learning Formulation for Turbulent Channel Flows 87
5.1 Mathematical Derivation of a New Turbulence Parameter (Φ) . . . 87
5.2 Machine Learning Predictive System . . . 90
5.3 Final Remarks . . . 94

6 Conclusions and Recommendations 95
6.1 Summary and Conclusions . . . 95
6.2 Recommendations . . . 97

A Source Code for the Examples Presented in Chapter 3 99
A.1 Field Inversion Optimization of a 1-D Radiative Heat Transfer Profile . . . 99
A.1.1 Version A: Build $\boldsymbol{\beta}$ multipliers for P_k (Inefficient) . . . 99
A.1.2 Version B: Build Explicit $\boldsymbol{\delta}$ Source Term Corrections (Efficient) . . . 104
A.2 Recovery of a Dynamic Viscosity Profile . . . 109
A.3 Field Inversion Optimization for the $k-\omega$ Turbulence Model . . . 113
A.3.1 Sympy Code to Derive the Matrices Required by the Discrete Adjoint Method . . . 113
A.3.2 Main Routine for the Field Inversion Optimizer . . . 116
A.4 Deep Dreaming the Reynolds Number . . . 131

B Machine Learning System for the MK Turbulence Model 139
B.1 Source Code for the Intelligent Relaxation Factor Methodology . . . 139
B.2 Extended K-fold Validation Results for the Predicted $\boldsymbol{\delta}_{\mathbf{k}}$ Corrections . . . 144
B.3 Extended K-fold Validation Results for the Predicted $\boldsymbol{\delta}_{\epsilon}$ Corrections . . . 147
B.4 Log-Space Features Created by the Trained Neural Networks . . . 149
B.4.1 Neural Network Weights for the Predicted $\boldsymbol{\delta}_{\mathbf{k}}$ Corrections . . . 149
B.4.2 Neural Network Weights for the Predicted $\boldsymbol{\delta}_{\epsilon}$ Corrections . . . 151

C Additional Results for the Alternative Machine Learning Formulation Presented in Chapter 5 153
C.1 Feature Weights Perceived by the First Layer of Hyperbolic Tangent Neurons . . . 153
C.2 Extended Results of the K-fold Validation Runs. . . . 155

Bibliography 157

List of Figures

2.1 Schematic representation of a wall boundary layer [1] 15
2.2 DNS velocity profile for a turbulent boundary layer at high Reynolds Numbers 15
2.3 Example of an overfitted model employing a high-order polynomial 22
2.4 Example of Gradient Descent optimization across a 3-D landscape 23
2.5 Estimation of the ideal learning rates for an optimization problem based on the Rosenbrock function [2] 27
2.6 Exponential degradation of the convergence rates for the Vanilla Gradient-Descent algorithm during the optimization of a parabolic curve (x^2) 28
2.7 Saddle point obtained at $x = 0$ for a cubic function (x^3) 28
2.8 Performance improvements obtained during the optimization of the Rosenbrock function after adding momentum to the system ($c_1 = 0.9$) 29
2.9 Structure of a densely connected Neural Network [3] 31
2.10 Activation functions commonly employed in Neural Networks 32
2.11 Schematic representation of a traditional Convolutional Neural Network 33
2.12 Schematic representation of a Long Short Term Memory (LSTM) neuron 34
2.13 Schematic representation of a Gate Recurrent Unit (GRU) neuron 34
2.14 Example of the Elbow Method applied to a polynomial fitting system 37
2.15 Example of the results obtained using a Support Vector Machine (SVM) algorithm . . . 38
2.16 Example of the K-means clustering algorithm in a 2-D feature space (X_1,X_2) 39
2.17 Field Inversion Machine Learning (FIML) procedure proposed by Parish & Duraisamy (2016) [4] 40

3.1 Field Inversion optimization results for the introductory problem proposed by Parish & Duraisamy (2016) [5] regarding the reconstruction of a 1-D Radiative Heat Transfer profile 44
3.2 Results for the recovery of a dynamic viscosity profile using Field Inversion optimization 46
3.3 Field Inversion results for the optimization of the $k-\omega$ turbulence model in a 1-D channel flow [5] 48
3.4 Drag coefficient of a smooth cylinder as a function of the Reynolds number 49
3.5 Neural Network system capable of reconstructing the functional relation between the drag coefficient of a cylinder and the Reynolds number ($C_D = f(Re_D)$) 50
3.6 Neural Network architecture employed to infer the existence of the Reynolds number based on the drag coefficient of a smooth cylinder 51
3.7 Results obtained after training the Neural Network architecture presented in Figure 3.6 51

4.1 Schematic representation of a channel flow 53
4.2 Results obtained by the RANS turbulence models listed in Section 2.1.7 for the test set defined by Pecnik et al. (2017) [6] 58
4.3 Turbulence budgets obtained for the MK model [7] during the study of the DNS case $JFM.CRe_\tau^*$ 59
4.4 Initial turbulence budgets for the Spalart-Allmaras model [8] during the study of the DNS case $JFM.CRe_\tau^*$ 61
4.5 Application of the Elbow method to determine the magnitude of I_U during the initial Field Inversion study for the MK turbulence model ($I_k = I_\epsilon = 1$) 63
4.6 Field Inversion corrections ($\boldsymbol{\delta}$) for the MK turbulence model obtained while employing $I_U = 100$ and $I_k = I_\epsilon = 1$ 64
4.7 Effect of the cross-interactions between I_k and I_ϵ for $I_U = 100$ during the Field Inversion study of the MK turbulence model for the DNS case $JFM.CRe_\tau^*$ 65
4.8 Application of the Elbow method to determine the magnitude of I_U during the Field Inversion study of the MK turbulence model while employing only $\boldsymbol{\delta_k}$ corrections ($I_\epsilon = 0$) 66

4.9 Independent set of $\boldsymbol{\delta_k}$ Field Inversion corrections obtained for the MK turbulence model while employing $I_U = 100$, $I_k = 1$ and $I_\epsilon = 0$ . . . 67
4.10 Application of the Elbow method to determine the magnitude of I_U during the Field Inversion study of the MK turbulence model while employing only $\boldsymbol{\delta_\epsilon}$ corrections ($I_k = 0$) 68
4.11 Independent set of $\boldsymbol{\delta_\epsilon}$ Field Inversion corrections obtained for the MK turbulence model while employing $I_U = 100$, $I_\epsilon = 1$ and $I_k = 0$ . . . 69
4.12 Application of the Elbow Method to determine the magnitude of I_U during the Field Inversion study of the SA turbulence model [8] . . . 71
4.13 Field Inversion corrections ($\boldsymbol{\delta_{SA}}$) obtained for the SA turbulence model while employing $I_U = 100$ (blue solid lines) and $I_U = 500$ (red dashed lines) . . . 72
4.14 Application of the Elbow Method to determine the overall magnitude of I_T during the Field Inversion study of the Turbulent Prandtl Number (Pr_T) . . . 73
4.15 Field Inversion results obtained for the Turbulent Prandtl Number (Pr_T) while employing $I_U = 100$ . . . 74
4.16 Comparison between the DNS data analyzed by Patel (2017) [9] (red dashed lines) and the results obtained by the Field Inversion optimizer (blue solid lines) for the turbulent Prandtl number Pr_T in various flow cases . . . 75
4.17 Turbulence budget for the optimized k-equation of the MK turbulence model after applying the independent set of $\boldsymbol{\delta_k}$ corrections obtained in Section 4.4.1 for the DNS case $JFM.CRe_\tau^*$. The fictitious Machine Learning predictions contain the perturbation term: $\Delta = 0.6\ y^3\ sin\,(8\pi y)$ . . . 76
4.18 Results obtained after applying the intelligent relaxation factor methodology to the exercise proposed in Figure 4.17 using $\alpha = 0.6$ . . . 76
4.19 Final framework established for the Field Inversion Machine Learning methodology (FIML) 78
4.20 Neural Network architecture created to predict the Field Inversion corrections ($\boldsymbol{\delta_k}$) required by the MK turbulence model . . . 80
4.21 Selection of the results obtained for the K-fold validation trials at the extreme ends of the test error ranking presented in Table 4.5 during the prediction of the independent sets of $\boldsymbol{\delta_k}$ corrections for the MK turbulence model . . . 81
4.22 Results of the uncertainty quantification process followed for the DNS case $JFM.CRe_\tau^*$ while employing the results of the K-fold validation runs {1, 2, 3, 4, 7, 10} after the prediction of the independent set of $\boldsymbol{\delta_k}$ corrections for the MK turbulence model . . . 82
4.23 Neural Network architecture created to predict the Field Inversion corrections ($\boldsymbol{\delta_\epsilon}$) required by the MK turbulence model . . . 83
4.24 Selection of the results obtained for the K-fold validation trials at the extreme ends of the test error ranking presented in Table 4.5 during the prediction of the independent sets of $\boldsymbol{\delta_\epsilon}$ corrections for the MK turbulence model . . . 84
4.25 Results of the uncertainty quantification process followed for the DNS case $JFM.CRe_\tau^*$ while employing the results of the K-fold validation runs {1, 2, 3, 4, 7, 10} after the prediction of the independent set of $\boldsymbol{\delta_\epsilon}$ corrections for the MK turbulence model . . . 85

5.1 Profiles obtained for the product $(\mu\nabla u)$ during the analysis of the DNS cases defined by Pecnik et al. (2017) [6] . . . 88
5.2 Trajectory curves obtained for the flow parameter $\Phi(Y^*)$ in the DNS cases defined by Pecnik et al. (2017) [6] and the incompressible channel flows studied by Jimenez et al. (2008) [10] . . . 89
5.3 Neural Network architecture chosen to predict the values of Φ in the channel flows studied 90
5.4 Results obtained for the velocity profiles considered in the test sets of the K-fold trial runs located at the extreme ends of the test error ranking presented in Table 5.1 . . . 92
5.5 Results of the uncertainty quantification process followed for the case $JFM.CRe_\tau^*$ after employing the results of the K-fold trial runs {1, 2, 3, 4, 7, 10}, and coupling the thermal energy equation to the system . . . 93

B.1 Results obtained for the K-fold validation trials {1-5} during the prediction of the independent sets of $\boldsymbol{\delta_k}$ corrections for the MK turbulence model . . . 144

B.2 Results obtained for the K-fold validation trials {6-10} during the prediction of the independent sets of $\boldsymbol{\delta}_\mathbf{k}$ corrections for the MK turbulence model 145
B.3 Results obtained for the K-fold validation trials {1-5} during the prediction of the independent sets of $\boldsymbol{\delta}_\epsilon$ corrections for the MK turbulence model 147
B.4 Results obtained for the K-fold validation trials {6-10} during the prediction of the independent sets of $\boldsymbol{\delta}_\epsilon$ corrections for the MK turbulence model 148
B.5 Neural Network architecture created to predict the Field Inversion corrections ($\boldsymbol{\delta}_\mathbf{k}$) required by the MK turbulence model 149
B.6 Neural Network architecture created to predict the Field Inversion corrections ($\boldsymbol{\delta}_\epsilon$) required by the MK turbulence model 151

C.1 Neural Network architecture chosen to predict the values of Φ in the channel flows studied 153
C.2 Results obtained for the velocity profiles considered in the test sets of the K-fold trial runs {1-5} using the Alternative Machine formulation presented in Chapter 5 155
C.3 Results obtained for the velocity profiles considered in the test sets of the K-fold trial runs {6-10} using the Alternative Machine formulation presented in Chapter 5 156

List of Tables

3.1 Errors perceived by the first layer of Hyperbolic Tangent Neurons in the Deep Learning system presented in Figure 3.6 while trying to reconstruct the Reynolds number 51

4.1 DNS flow cases extracted from the PhD thesis of Ashish Patel (2017) [9] and the work of Pecnik et al. (2017) [6] 55
4.2 DNS flow cases simulated by Trettel & Larson (2016) [11] 56
4.3 Incompressible DNS flow cases simulated by Jimenez et al. (2008) [10] 56
4.4 Test cases considered during the implementation of the K-fold validation method for the study of Neural Network architectures 79
4.5 Normalized cost function values obtained for the training and test sets defined in every K-fold validation run during the prediction of $\boldsymbol{\delta}_{\mathbf{k}}$ for the MK turbulence model 80
4.6 Normalized cost function values obtained for the training and test sets defined in every K-fold validation run during the prediction of $\boldsymbol{\delta}_{\epsilon}$ for the MK turbulence model 83

5.1 Maximum velocity error values encountered during the prediction of the DNS flow cases considered in the training set and the test set of every K-fold validation run 91

B.1 Neural Network weights in log-space for each input feature perceived by the Neuron (N-1) presented in Figure B.5 149
B.2 Neural Network weights in log-space for each input feature perceived by the Neuron (N-2) presented in Figure B.5 149
B.3 Neural Network weights in log-space for each input feature perceived by the Neuron (N-3) presented in Figure B.5 150
B.4 Neural Network weights in log-space for each input feature perceived by the Neuron (N-4) presented in Figure B.5 150
B.5 Neural Network weights in log-space for each input feature perceived by the Neuron (N-5) presented in Figure B.5 150
B.6 Neural Network weights in log-space for each input feature perceived by the Neuron (N-1) presented in Figure B.6 151
B.7 Neural Network weights in log-space for each input feature perceived by the Neuron (N-2) presented in Figure B.6 151
B.8 Neural Network weights in log-space for each input feature perceived by the Neuron (N-3) presented in Figure B.6 152

C.1 Neural Network weights in log-space for each input feature perceived by the Neuron (N-1) presented in Figure C.1 153
C.2 Neural Network weights in log-space for each input feature perceived by the Neuron (N-2) presented in Figure C.1 154
C.3 Neural Network weights in log-space for each input feature perceived by the Neuron (N-3) presented in Figure C.1 154
C.4 Neural Network weights in log-space for each input feature perceived by the Neuron (N-4) presented in Figure C.1 154

Acknowledgements

I would like to thank my supervisor Dr. ir. Rene Pecnik for his help and guidance during the present thesis work, as well as for the opportunity to carry out a Machine Learning project. I am also grateful to Gustavo Otero for his advice regarding RANS turbulence modelling, and for the opportunity to collaborate developing CFD solvers and become co-author in the publication: ”Turbulence modelling for flows with strong variations in thermo-physical properties” [12]. I would further like to thank my daily PhD supervisor Stephan Smit for his frequent help and his detailed review of this work.

I would like to thank the MSc thesis committee members for their time and effort in assessing the present thesis report. I would also like to thank Ashish Patel for his exhaustive study of variable property flows and for helping me access the source code of his simulations. I also acknowledge the contributions made by Andrew Trettel, Johan Larsson and Sergio Pirozzoli who kindly shared their DNS databases of supersonic flows with the research group.

I am grateful to my friends and colleagues for their constant support and motivation throughout my studies.

Finally, I would like to thank my family for their unconditional love and support throughout my life. Their faith in me and their positive energy have been a great source of inspiration.

Rafael Diez
Delft, September 2018

1 Introduction

1.1. Overview

Turbulence modelling constitutes one of the greatest unsolved problems in physics and mathematics. While the governing equations of fluids have long been established, Direct Numerical Simulations (DNS) based on these equations require computational resources currently unavailable for real-world engineering applications. Even according to optimistic estimates, the computing power required may not become available until the end of this century [13]. The issues are caused by the chaotic vortex structures created by turbulence, which form an energy cascade ranging from large vortex structures to microscopic dissipation vortices [14–16]. Despite these difficulties, understanding turbulent flows remains critical for applications of great engineering importance, such as turbine electric generators, refrigeration cycles, wind turbine farms, heat exchangers, aviation, fluid pump stations, and the chemical industry. Due to the previous need for practical models, researchers have developed RANS turbulence models, which attempt to describe the average trends observed in turbulent flows. This is physically plausible, since turbulence is bound by global energy constraints and dissipation mechanisms, which limit its scope of action.

One of the greatest limitations of existing RANS turbulence models is given by their focus on isothermal incompressible fluids, which present constant molecular properties. However, many industrial applications present turbulent flows coupled to heat transfer cycles, where strong property changes occur. Since the properties of such flows are largely unknown, the Process & Energy Department at the TU Delft has extensively researched their behavior in order to build better formulations. Several scaling theories have been proposed within this context [6, 9, 17] and large DNS databases have been assembled. The present thesis will thus explore the possibility of employing such DNS datasets in order to build data-driven RANS turbulence models. The use of Machine Learning to improve RANS turbulence modelling has been identified as a potential source of advancement by leading researchers [18], as these models can build complex relations based on existing information. Since the Navier-Stokes equations only correspond to a set of differential equations, it is possible to build alternative mathematical models which can predict the trends observed in these equations.

The analysis performed during the present thesis work was largely based on a technique called Field Inversion Machine Learning (FIML), which was proposed by Parish & Duraisamy (2016) [5]. The framework proposed by these authors consists in building corrections for existing RANS turbulence models instead of attempting to rebuild existing knowledge. The FIML methodology further splits the analysis process into two stages. In the first stage, a data mining process known as Field Inversion is performed, such that an ideal set of corrections for the studied RANS turbulence model can be identified. Then, in a second stage, a Machine Learning system is trained in order to replicate the corrections identified. The main advantage of this procedure is that a Neural Network can be trained without coupling a CFD solver to the optimizer. This helps to improve the efficiency of the procedure by orders of magnitude. During the present thesis work, the formulation of an intelligent relaxation factor methodology was proposed in order to improve the overall results obtained, and to establish a scientific framework to eliminate any spurious corrections which may harm the performance of CFD solvers.

1.2. Thesis Objectives and Contributions

The main objective of the present thesis was to explore the possibility of developing robust Machine Learning (ML) systems capable of improving RANS turbulence models for variable-property flows. The present work corresponds to the first instance where ML is applied within this context. The main tasks considered during the present thesis work were the following:

i. Perform a literature survey of RANS turbulence modelling in variable-property flows, and the Machine Learning techniques which could be applied to improve these models.

ii. Develop robust ML methods for Data-Driven RANS turbulence modelling.

iii. Perform a detailed FIML (Field Inversion Machine Learning) study employing the DNS database assembled, and determine the best overall methodology.

iv. Derive an alternative ML formulation for channel flows and compare the results obtained with respect to the previous FIML study (point iii).

In the previous outline, it can be noted that 50% of the tasks considered during the present thesis required deriving new Machine Learning methods for RANS turbulence modelling. The final contributions made to this field during the present work were the following:

i. Synthesize a robust Field Inversion optimizer for RANS turbulence modelling. The optimizer assembled is efficient, fully-automatic, unconditionally stable and performs well in large-scale systems. The synthesis of this method marks a large contrast with respect to the Bayesian Field Inversion optimizers employed by Parish & Duraisamy (2016) [5], since their algorithm also performs an approximate side-study of the probabilities associated to obtaining certain parameters. This approach can result unnecessarily complex, and further yield distributions whose mathematical consistency cannot be guaranteed in large-scale systems [5]. The FIML methodology can operate by only finding the optimal parameter distributions associated to the cost functions established.

ii. Create a Sympy [19] script capable of building explicit formulas for every solution component required in Field Inversion, and thereby avoid the use of Automatic Differentiation (AD) libraries. During the tests performed, it was detected that employing AD libraries could increase the running times by orders of magnitude, and further impose limitations upon the structure of the main routines.

iii. Create a Field Inversion methodology capable of building explicit source term corrections ($\boldsymbol{\delta}$) for a RANS turbulence model while remaining unbiased and numerically stable.

iv. Introduce logarithmic neurons [20] into the first layer of the Neural Networks created, such that the system could discover appropriate dimensionless groups (e.g., $Re^a Pr^b$).

v. Derive an intelligent relaxation factor methodology to mitigate any spurious corrections predicted by Neural Networks. The algorithm created can be applied to large-scale systems, and it was able to reach decisions in less than a millisecond during the problems considered in the present thesis.

vi. Derive an innovative turbulence parameter for channel flows. The parameter found collapsed all the turbulence statistics observed into similar trajectory curves. Thanks to this formulation, it was possible to assess the differences between the DNS flow cases studied and to build an alternative Neural Network architecture.

Based on the previous analysis, it can be noted that substantial improvements were introduced into the original FIML methodology proposed by Parish & Duraisamy (2016) [5] during the present work. A detailed explanation for each contribution will be given throughout the present report.

1.3. Thesis Outline

Chapter 2 presents a theoretical review of RANS turbulence modelling, CFD optimization, Machine Learning applied to CFD and the general theory of mathematical optimization (e.g., Hessian-free optimizers).

Chapter 3 discusses a total of 4 preliminary examples which were studied in order to develop a robust Machine Learning methodology. Two of the studied problems correspond to the original Field Inversion scenarios proposed by Parish & Duraisamy (2016) [5], whereas the (two) remaining exercises correspond to new problems created.

Chapter 4 presents the application of the FIML methodology to the DNS database of variable-property flows assembled during the present study. The mathematical formulation of the intelligent relaxation factor methodology also corresponds to part of the analysis presented.

Chapter 5 presents an alternative Machine Learning formulation which is based on a new turbulence parameter derived for channel flows. Thanks to this parameter, it was possible to visualize the true complexity of the flows studied in Chapter 4, and to build an innovative Neural Network architecture capable of making highly accurate predictions for any DNS flow well-represented in the training set.

Chapter 6 presents the conclusions found during the present study, along with future recommendations.

2
Theoretical Background

This Chapter will describe the theoretical background employed to analyze turbulent channel flows and to build improved data-driven RANS turbulence models. The governing equations of fluids mechanics and the principles of RANS turbulence modelling will be first presented in Section 2.1, where the different modelling assumptions required to handle turbulent flows subject to strong property-gradients will also be discussed. Then, in Section 2.2, the principles of mathematical optimization for CFD and Machine Learning will be described, along with the required driving algorithms. Many optimization methods described in this Section are also applicable to other fields, such as structural optimization. Finally, in Section 2.3, the most important Machine Learning techniques applicable to RANS turbulence modelling will be described. A special emphasis will be placed in Deep Neural Networks and FIML (Field Inversion Machine Learning), since these are the most relevant techniques. However, the present Chapter contains a rigorous analysis of the global Machine Learning techniques described in the literature, which was deemed necessary since this corresponds to an emerging branch of engineering. As a result, extensive research was required in order to identify the best approaches available.

2.1. Fluid Mechanics

2.1.1. Navier-Stokes Equations

The study of fluid mechanics is primarily based on the Navier-Stokes equations. These equations can be derived from first principles by applying Newton's second law and by noting that mass must also be conserved [15]:

$$\rho_{,t} + \nabla \cdot (\rho\ \mathbf{u}) = 0, \tag{2.1}$$

$$(\rho\ \mathbf{u})_{,t} + \nabla \cdot \left(\rho\ \mathbf{u}\ \mathbf{u}^T\right) = -\nabla P + \nabla \cdot \boldsymbol{\tau} + \mathbf{f}. \tag{2.2}$$

In eqs. (2.1-2.2), $\mathbf{u}$ corresponds to the local (fluid) velocity vector, P to the pressure, ρ to the density, $\boldsymbol{\tau}$ to the viscous stress tensor, and $\mathbf{f}$ to any local body forces (e.g., gravity). The LHS (Left-Hand-Side) of eq. (2.2) corresponds to the expansion of the material derivative $D(\rho\mathbf{u})/Dt$, which measures the momentum change of a particle travelling with the fluid. By making this realization, it can be noted that eq. (2.2) is the result of a direct force balance for an infinitesimal particle. In order to close the Navier-Stokes equations, a model for the viscous stress tensor $\boldsymbol{\tau}$ must be introduced. Based on experimental observations, it has been noted that the majority of pure liquids and gases can be classified as Newtonian fluids [21]. Several fluids of great engineering importance can be found in this category, such as air, water, steam, oil and natural gas [21]. During the present thesis, only Newtonian fluids will be considered, since the study of turbulence in these substances is still an active area of research. From a mathematical perspective, Newtonian fluids are characterized by the following law:

$$\boldsymbol{\tau}_{\mathbf{n}} = \mu \left(\nabla\mathbf{u} + \nabla\mathbf{u}^T\right) + \lambda \left(\nabla \cdot \mathbf{u}\right) \mathbf{I}. \tag{2.3}$$

In eq. (2.3), it can be noted that the viscous stress tensor is a linear function of the velocity gradients ($\nabla\mathbf{u}$). The variable μ corresponds to the dynamic viscosity of fluids, which is a thermodynamic property. The second term present in eq. (2.3) corresponds to a correction employed to accurately model the normal viscous stresses experienced by fluids as they undergo volumetric dilatations (or contractions). Since this term only affects the diagonal entries of $\boldsymbol{\tau}$, the variable $\mathbf{I}$ corresponds to a 3x3 identity matrix. The behavior of $\lambda\,(\nabla\cdot\mathbf{u})\,\mathbf{I}$ is governed by the secondary viscosity λ, which is also a thermodynamic property of fluids. However, since liquids are nearly incompressible, this property is usually studied in the context of gases. Stokes proposed the hypothesis that $\lambda = -2/3\ \mu$, which has proved to be accurate in practice [15, 22, 23]. It must be noted nonetheless that the effects of the secondary viscosity λ are only noticeable in gases undergoing abrupt volumetric changes, and that gases flowing at low subsonic speeds usually behave as incompressible fluids [24]. This has allowed researchers to build turbulent models which are applicable to both liquids and gases in conventional applications.

The system of eqs. (2.1-2.3) formally corresponds to the Navier-Stokes equations for Newtonian fluids. These equations can fully describe the evolution of fluids with constant molecular properties (μ and ρ), such as the flow around a racing boat or a wind turbine. However, in flows where strong property gradients occur, such as those studied during the present thesis, the Navier-Stokes equations must be further coupled with a thermal energy equation in order to reach an equilibrium. The system of equations formed therein constitutes the most general case. The energy equation for fluids will be presented next in Section 2.1.2.

As a final note, it must be noted that understanding the properties of the Navier-Stokes equations constitutes one of the most important research topics in mathematics today. Even proving whether the Navier-Stokes equations always have smooth physical solutions or not has been declared as one of the seven Millennium Prize problems by the Clay Institute of Mathematics (2000) [25]. While the Navier-Stokes equations have always proven to work numerically, a mathematical proof justifying such success is still missing. A counter-example providing a case where the Navier-Stokes equations would fail also constitutes a solution to this Millennium Prize problem.

2.1.2. Energy Equation

The present Section will describe the energy equation of fluids, which keeps track of the evolution in their thermal energy. As it was mentioned above, this equation can be coupled with the Navier-Stokes equations in order to form a complete set of governing equations for fluids. The resulting equations are naturally compatible with any thermodynamic sub-model chosen for the fluids. These models can range from simple relations, such as the perfect gas law, to large databases containing empirical results. The derivation of a thermal energy equation for fluids starts by performing an energy balance over an infinitesimal control volume. This derivation yields an evolution equation for the total energy held by a fluid (E), which takes the following form [15]:

$$(\rho\ E)_{,t} + \nabla\cdot(\rho\ \mathbf{u}\ E) = -\nabla\cdot(P\ \mathbf{u}) + \nabla\cdot(\mathbf{u}\ \boldsymbol{\tau}) + \nabla\cdot(\lambda\ \nabla T) + S_E. \tag{2.4}$$

In eq. (2.4), it can be noted that only three new variables have been added with respect to the Navier-Stokes equations, namely the thermal conductivity (λ), the temperature field (T) and any possible global energy sources (S_E). In general, the total energy E can be understood as $E = I + K$, where I corresponds to the thermal energy stored by the fluid and K to its kinetic energy. While tracking the total energy of a fluid (E) could yield favorable numerical properties in certain applications, it is often ideal to split the flow's total energy into separate components representing the kinetic energy (K) and the thermal energy (I). A direct equation for the evolution of the kinetic energy (K) can be obtained from the Navier-Stokes equations by pre-multiplying these equations with their respective velocity components [15]:

$$(\rho\ K)_{,t} + \nabla\cdot(\rho\ \mathbf{u}\ K) = -\mathbf{u}\cdot\nabla P + \mathbf{u}\cdot(\nabla\cdot\boldsymbol{\tau}) + \mathbf{u}\cdot\mathbf{f}. \tag{2.5}$$

It can be noted that eq. (2.5) does not contain new variables with respect to the Navier-Stokes equations, which is natural since it only corresponds to a sub-product of them. Eq. (2.5) can now be subtracted from eq. (2.4) in order to create a direct equation for the evolution of a fluid's thermal energy (I) [15]:

$$\underbrace{(\rho\ I)_{,t}}_{\text{(i)}}+\underbrace{\nabla\cdot(\rho\ \mathbf{u}\ I)}_{\text{(ii)}}=-\underbrace{P\,(\nabla\cdot\mathbf{u})}_{\text{(iii)}}+\underbrace{\boldsymbol{\tau}:\nabla\mathbf{u}}_{\text{(iv)}}+\underbrace{\nabla\cdot(\lambda\ \nabla T)}_{\text{(v)}}+\underbrace{S_I}_{\text{(vi)}}\,. \tag{2.6}$$

In eq. (2.6), it can be noted that several derivatives present in eq. (2.4) have been expanded and simplified. For example, the local pressure gradients (∇P) do not directly influence the evolution of the fluid's thermal energy (I). This can be interpreted as a natural result, since the evolution of the kinetic energy (K) explicitly depended on ∇P. The source term S_I present in eq. (2.6) corresponds to any possible volumetric heat additions, such as radiative heat transfer or chemical reactions. One simplification which is often made regarding eq. (2.6) is to replace the internal energy (I) by $I = c_p\ T$, where c_p corresponds to the specific heat capacity of the fluid. Since c_p is nearly constant for many applications involving liquids and gases, this simplification creates an energy equation which explicitly models the fluid's temperature. While this simplification was employed during the present thesis, it must be noted that the internal energy of a fluid (I) formally corresponds to a non-linear function of its pressure and temperature: $I = I(P,T)$. Such non-linear dependency can be found in many substances in nature, especially when fluids undergo large changes.

Each term present in eq. (2.6) has a clear physical meaning, which creates distinct effects in nature. The effect of each term will be explained below:

- **Terms (i) and (ii):** These terms, located at the LHS of eq. (2.6), correspond to the material derivative of the thermal energy stored by a fluid particle: $D(\rho\ I)/Dt$. This term is analogue to the momentum change derivative present in the LHS of the Navier-Stokes equations, and it has two distinct consequences in nature. The first consequence is to give fluids thermal inertia. This effect can be clearly appreciated in the oceans, which manage to retain relatively stable temperature profiles throughout a season. The second effect of $D(\rho\ I)/Dt$ is to produce thermal convection. Convection is a phenomenon caused by the ability of fluids to carry energy away simply due to their motion. A simple case of convection can be observed when water is heated by a boiler and transported for service at a distant location. Convection also appears in turbulent flows, when the small eddies present in these flows rapidly transport energy away from a certain location. This process can even occur perpendicular to the mean flow direction, thanks to the interactions between these eddies. However, a complete description of this phenomena is deeply interwoven with the process of molecular heat conduction in fluids, which is given by the fifth term present in eq. (2.6).

- **Term (iii):** The first term present in the RHS (Right-Hand-Side) of eq. (2.6) indicates that the thermal energy stored by a fluid decreases as its volume expands, or vice-versa. Since the thermal energy of a fluid is strongly related to its temperature, the previous statement can also be extended to the temperature of a fluid. This dilatation principle is actively employed in the expansion valves of refrigeration systems, which create artificially low temperatures by expanding their working fluid. Thanks to this principle, modern refrigerators can operate without relying on chemical reactions. Heat pumps also employ the principle of volumetric expansion, although in reverse order. By artificially expanding and compressing their working fluid, heat pumps are able to absorb energy from a cold source and release it at a warmer location.

- **Term (iv):** The fourth term present in eq. (2.6) corresponds to the viscous heating effect experienced by fluids, which is strictly positive. This term converts the energy dissipated by the viscous forces of fluids into thermal energy, thereby contributing to the process of entropy generation and creating irreversible losses. Viscous heating can be clearly observed when meteors melt upon entering the Earth's atmosphere, and it is also relevant for supersonic flows. However, viscous heating rarely plays a role at low subsonic speeds, since the velocity gradients of these flows also tend to be weaker.

- **Term (v):** The fifth term present in eq. (2.6) corresponds to the diffusion of energy in fluids by heat conduction. Even though fluids are constantly moving, this term is identical to the heat conduction term present in solids. In fact, the heat transfer equation of solids can be recovered from eq. (2.6) by substituting $\mathbf{u} = \mathbf{0}$. This term corresponds to only one of the possible mechanisms for heat transport however, since it was previously mentioned that fluids can also transport energy away by convection.

One further consequence of the fifth term (v) present in eq. (2.6) is to create an analogy with the viscous-stress term present in the Navier-Stokes equations. This analogy gave rise to the Prandtl number theory [1, 15], which established a relation between the velocity and the temperature profiles expected in a boundary layer. A boundary layer is a narrow flow region located next to a wall, where a sharp transition takes place between the outer flow velocities and the (zero) non-slip velocity condition imposed by the wall. The Prandtl number analogy will be further described in Section 2.1.3.

- **Term (vi):** Finally, the sixth term present in eq. (2.6) corresponds to any external volumetric heat sources added to the fluid. As it was discussed, these terms are usually linked to radiative heat transfer contributions or local chemical reactions. The fluid itself can also become a source of radiative heat transfer at very high temperatures. However, this effect is rarely found in conventional applications.

The most general form of the governing equations for fluids can be finally summarized as follows:

$$\rho_{,t} + \nabla \cdot (\rho\ \mathbf{u}) = 0, \tag{2.1}$$

$$(\rho\ \mathbf{u})_{,t} + \nabla \cdot \left(\rho\ \mathbf{u}\ \mathbf{u}^T\right) = -\nabla P + \nabla \cdot \boldsymbol{\tau} + \mathbf{f}, \tag{2.2}$$

$$(\rho\ I)_{,t} + \nabla \cdot (\rho\ \mathbf{u}\ I) = -P\,(\nabla \cdot \mathbf{u}) + \boldsymbol{\tau} : \nabla\mathbf{u} + \nabla \cdot (\lambda\ \nabla T) + S_I. \tag{2.6}$$

2.1.3. Dimensionless Form of the Governing Equations

Despite the complexity of the governing equations of fluid mechanics, it can be proven that these equations can be casted into a dimensionless form, which only depends on three dimensionless numbers. Based on these parameters, a distinct response can be expected for every system, and it may be easier to form correlations. In order to start the derivation process, each variable (x) must be first assumed to have a characteristic physical scale X_0 and a dimensionless component $\hat{x}$:

$$\rho = \rho_0 \hat{\rho}, \tag{2.7}$$

$$\mu = \mu_0 \hat{\mu}, \tag{2.8}$$

$$\lambda = \lambda_0 \hat{\lambda}, \tag{2.9}$$

$$\mathbf{u} = U_0 \hat{\mathbf{u}}, \tag{2.10}$$

$$T = T_0 \hat{T}, \tag{2.11}$$

$$\nabla = \frac{1}{L_0} \hat{\nabla}, \tag{2.12}$$

$$\frac{\partial}{\partial t} = \frac{U_0}{L_0} \frac{\partial}{\partial \hat{t}}, \tag{2.13}$$

$$P = \rho_0 {U_0}^2 \hat{P}, \tag{2.14}$$

$$\mathbf{f} = \frac{\rho_0 {U_0}^2}{L_0}\hat{\mathbf{f}}, \tag{2.15}$$

$$\boldsymbol{\tau} = \frac{\mu_0 U_0}{L_0^2}\hat{\boldsymbol{\tau}}, \tag{2.16}$$

$$I = c_{p,0} T_0 \hat{I}, \tag{2.17}$$

$$S_I = \frac{\rho_0 c_{p,0} T_0 U_0}{L_0}\hat{S}_I. \tag{2.18}$$

Regarding the previous physical scales, it can be noted that a reference specific heat value ($c_{p,0}$) was employed in eqs. (2.17) and (2.18). This approach can be justified, since the internal energy of a fluid is almost exclusively a function of its temperature ($I = I(T)$) for a wide variety of engineering applications. Replacing eqs. (2.7-2.18) into the governing equations of fluid mechanics yields the following results:

$$\hat{\rho}_{,\hat{t}} + \hat{\nabla}\cdot(\hat{\rho}\;\hat{\mathbf{u}}) = 0, \tag{2.19}$$

$$(\hat{\rho}\;\hat{\mathbf{u}})_{,\hat{t}} + \nabla\cdot\left(\hat{\rho}\;\hat{\mathbf{u}}\;\hat{\mathbf{u}}^T\right) = -\hat{\nabla}\hat{P} + \frac{1}{Re_0}\;\hat{\nabla}\cdot\hat{\boldsymbol{\tau}} + \hat{\mathbf{f}}, \tag{2.20}$$

$$\left(\hat{\rho}\;\hat{I}\right)_{,\hat{t}} + \nabla\cdot\left(\hat{\rho}\;\hat{\mathbf{u}}\;\hat{I}\right) = -Ec_0\cdot\hat{P}\left(\hat{\nabla}\cdot\hat{\mathbf{u}}\right) + \frac{Ec_0}{Re_0}\;\hat{\boldsymbol{\tau}}:\hat{\nabla}\hat{\mathbf{u}} + \frac{1}{Re_0 Pr_0}\;\hat{\nabla}\cdot\left(\hat{\lambda}\;\hat{\nabla}\hat{T}\right) + \hat{S}_I. \tag{2.21}$$

In eqs. (2.20) and (2.21), Re_0, Pr_0 and Ec_0 correspond to the Reynolds number, the Prandtl number and the Eckert number respectively. These parameters correspond to dimensionless groups given by the following formulas:

$$Re_0 = \frac{\rho_0 U_0 L_0}{\mu_0}, \tag{2.22}$$

$$Pr_0 = \frac{c_{p,0}\mu_0}{\lambda_0}, \tag{2.23}$$

$$Ec_0 = \frac{{U_0}^2}{c_{p,0}T_0}. \tag{2.24}$$

The previous dimensionless numbers are often employed in fluid mechanics to predict the expected response of a system. The quality of the predictions made however depends on the complexity of the problem and the amount of data available. The dimensionless form of the thermodynamic properties considered for fluids (ρ, μ, λ, etc.) have been retained, since these properties will suffer variations during the flow cases studied in the present thesis. The physical meaning of the dimensionless numbers Re, Pr and Ec will be explained in the next Section.

Dimensionless Groups in Fluid Mechanics

The dimensionless numbers found in the governing equations of fluid mechanics (Re, Pr and Ec) hold key physical insights into their behavior. A brief explanation regarding the meaning of each parameter will be presented below:

- **Reynolds number (Re):** The Reynolds number is often the first parameter studied in the context of fluid mechanics, since it directly controls the Navier-Stokes equations. Based on its definition, the Reynolds Number can be understood as the ratio between the inertial and the viscous forces present in a fluid. As the Reynolds number increases for a configuration, the inertial forces start to dominate the flow behavior and turbulence eventually emerges. Turbulence is characterized by a sharp collapse in the smooth streamlines observed in low-speed (laminar) flows and by the formation of a chaotic vortex regime. Despite its apparent anarchy, turbulence can still be bound by global force and energy balances due to its dissipative nature. The emergence of turbulence can also bring a certain degree of disconnection between different flow regions, which is evidenced by the fact that turbulent flows are often scaled by local flow parameters. For example, turbulent wall-bounded flows are scaled by quantities derived from their wall shear stresses, and present a much lower dependence to the outer flow conditions than laminar flows. A detailed description of wall-bounded flows can be found in Section 2.1.6.

- **Prandtl number (Pr):** The Prandtl number corresponds to another important flow parameter, which greatly governs how heat transfer takes place in a system. This number can be defined as the ratio between the kinematic viscosity of a fluid ($\nu = \mu/\rho$) and its perceived thermal diffusivity ($\lambda/(\rho c_p)$). For $Pr \ll 1$, the thermal diffusivity of a fluid is much greater than its kinematic diffusivity, and thus heat transfer takes place mostly by direct heat conduction [1]. This behavior is yet reversed for $Pr \gg 1$, when energy is transported mostly by convection. Unlike the other dimensionless numbers, the Prandtl number corresponds to an explicit thermodynamic property of fluids, and it therefore marks how susceptible is a substance to present the heat transfer characteristics of a solid. The behavior of a solid can be recovered in the limit when $Pr \to 0$.

- **Eckert number (Ec):** The Eckert number of a flow measures the ratio between its kinetic energy (U_0^2) and its thermal inertia ($c_p T_0$). Thanks to this metric, it is possible to estimate whether a flow contains enough kinetic energy to produce significant thermal changes in a fluid or not. If the Eckert number is low, it can be seen that both the viscous heating and the volumetric expansion terms present in eq. (2.21) tend to vanish, since the kinetic energy content of the fluid is deemed insufficient.

Arbitrary Scaling Conventions

Before concluding the analysis of the dimensionless form of the governing equations, it must be noted that employing eqs. (2.19-2.21) directly holds certain drawbacks from a practical perspective. While building a database, this approach would require keeping track of all the dimensionless variables ($\hat{x}$), their physical scales (X_0), the relevant dimensionless numbers (Re, Pr and Ec) and perhaps even the original physical variables (x). Besides the complications created by storing the previous information, it also results unclear which dimensions should be chosen to study a given problem. For example, in order to study a channel flow at $Re = UD/\nu = 100$, the main dimensions could be chosen as $(U, D, \nu) = (100, 1, 1)$, $(U, D, \nu) = (1, 1, 0.01)$, etc. In order to mitigate these issues, it is better to restrict the family of studied problems using the following conventions [9]:

$$U_0 = 1 \implies u = \hat{u}, \tag{2.25}$$

$$L_0 = 1 \implies \nabla = \hat{\nabla}, \tag{2.26}$$

$$\rho_0 = 1 \implies \rho = \hat{\rho}, \tag{2.27}$$

$$T_0 = 1 \implies T = \hat{T}, \tag{2.28}$$

$$\mu_0 = \frac{1}{Re_0}, \tag{2.29}$$

$$c_{p,0} = \frac{1}{Ec_0}, \tag{2.30}$$

$$\lambda_0 = \frac{c_{p,0}\mu_0}{Pr_0} = \frac{1}{Re_0 Pr_0 Ec_0}. \tag{2.31}$$

In eqs. (2.25-2.28), it can be seen that unitary reference scales were chosen such that $x = \hat{x}$ for the most important vector fields. Thanks to these unitary scales, the reference molecular properties (μ_0,

$c_{p,0}$ and λ_0) also became a direct function of the main dimensionless groups (Pr_0, Re_0, Ec_0). The family of studied problems during the present thesis was thus limited to these conventions. Any existing CFD database was first converted into this working space. At this point, it is important to note that Re_0, Pr_0 and Ec_0 were defined based on wall quantities whenever necessary. In cases involving variable-property flows, it is still possible to obtain appropriate profiles for all the thermodynamic properties (μ, ρ, λ, etc.), since their dimensionless forms were not removed from eqs. (2.19) and (2.21).

2.1.4. Reynolds-Averaged Navier-Stokes Equations

Since highly turbulent flows have a well-defined global structure [15], it is possible to define averaging procedures to describe their mean flow behavior. The first technique developed historically is the Reynolds-averaging procedure. This method starts by decomposing every flow variable (x) into $x = \overline{x} + x'$, where $\overline{x}$ corresponds to its mean arithmetic value, and x' to its fluctuating counterpart. In flows with constant properties, the Reynolds-averaging procedure can be applied directly without further complications. However, in flows with variable molecular properties (μ, ρ, λ, etc.), fluctuating cross-interactions can appear in the governing equations, such as $\mu'\nabla\mathbf{u}'$ or $\lambda'\nabla T'$. While sub-models could be developed for these interactions, research still focuses in flows obeying Morkovin's hypothesis. This hypothesis states that only the mean thermodynamic properties of a fluid influence its behavior, and that any fluctuating components have a negligible impact in the flow [17, 26]. Therefore, any terms such as $\mu'\nabla\mathbf{u}'$ are effectively cancelled. Furthermore, it can be proven that the Reynolds-averaging procedure yields identical results than a more advanced averaging technique called Favre-decomposition in flows governed by Morkovin's hypothesis [17].

The derivation of the Reynolds-averaging procedure formally starts by defining the average and the fluctuating components of every flow variable according to the following formulas:

$$\overline{x} = \frac{1}{\Delta T}\int_0^{\Delta T} x\ dt, \tag{2.32}$$

$$x' = x - \overline{x}. \tag{2.33}$$

In eq. (2.32), ΔT refers to the averaging time window employed to measure $\overline{x}$. This time window should be as long as possible, since the quality of the statistics obtained improves asymptotically. However, beyond these considerations, the Reynolds-averaging procedure is often considered insufficient during the study of variable-property flows, since all the variables are pondered equally without any type of physical weighting scheme. In order to overcome this drawback, the Favre-decomposition method defines a density-weighted average for each variable [9]:

$$\tilde{x} = \frac{\int_0^{\Delta T} \rho x\ dt}{\int_0^{\Delta T} \rho\ dt}, \tag{2.34}$$

$$x'' = x - \tilde{x}. \tag{2.35}$$

In eqs. (2.34-2.35), $\tilde{x}$ corresponds to the density-weighted average of each variable, and x'' to its fluctuating counterpart. The difference between the Reynolds and the Favre averaging schemes is given by the following equations [9]:

$$\overline{x} = \tilde{x} - \frac{\overline{\rho' x'}}{\overline{\rho}}, \tag{2.36}$$

$$x' = x'' + \frac{\overline{\rho' x'}}{\overline{\rho}}. \tag{2.37}$$

Based on eqs. (2.36) and (2.37), it can be noted that Morkovin's hypothesis implies that $\overline{x} \approx \tilde{x}$, since the influence of any density fluctuations in the flow can be assumed to be negligible ($\overline{\rho' x'} \approx 0$). As a result, the Favre averaging scheme becomes identical to the Reynolds-averaging method under these conditions ($\tilde{x} \approx \overline{x}$). After a rigorous analysis, it can be proven that the final RANS equations take the following form [9]:

$$\overline{\rho}_{,t} + \nabla \cdot (\overline{\rho}\ \tilde{\mathbf{u}}) = 0, \tag{2.38}$$

$$(\overline{\rho}\ \tilde{\mathbf{u}})_{,t} + \nabla \cdot \left(\overline{\rho}\ \tilde{\mathbf{u}}\ \tilde{\mathbf{u}}^T\right) = -\nabla\overline{P} + \overline{\mathbf{f}} + \nabla \cdot \left(\overline{\boldsymbol{\tau}}_{\mathbf{n}} - \overline{\rho\ \mathbf{u}''\mathbf{u}''^T}\right), \tag{2.39}$$

$$\left(\overline{\rho}\ \tilde{I}\right)_{,t} + \nabla \cdot \left(\overline{\rho}\ \tilde{\mathbf{u}}\ \tilde{I}\right) = -\overline{P}\,(\nabla \cdot \tilde{\mathbf{u}}) + \overline{\boldsymbol{\tau}}_{\mathbf{n}} : \nabla\tilde{\mathbf{u}} + \overline{S}_I + \nabla \cdot \left(\overline{\lambda}\ \nabla\widetilde{T} - \overline{\rho\ \mathbf{u}''\ I''}\right) + \overline{\boldsymbol{\tau}_{\mathbf{n}}{}' : \nabla\mathbf{u}''}. \tag{2.40}$$

In eqs. (2.38-2.40), it can be observed that the general form of the RANS equations is only influenced by three turbulent terms in total. One important identity associated to eq. (2.38) is the fact that $\nabla \cdot \mathbf{u}'' = 0$. This identity can be proven by subtracting eqs. (2.1) and (2.38) while applying Morkovin's hypothesis. It can thus be noted that the general RANS equations would take a far more complex form without the aid of Morkovin's hypothesis. The incompressible nature of the velocity fluctuations ($\mathbf{u}''$) also proves that turbulence is directly linked to the viscous shear stresses, and that turbulent flows can be expected to take similar forms for liquids and gases indeed. In eq. (2.40), it can be further noted that the product $\overline{P' \cdot (\nabla \cdot \mathbf{u}'')}$ has already been cancelled by applying the identity $\nabla \cdot \mathbf{u}'' = 0$.

As an additional note, the RANS equations previously given are also called URANS (Unsteady-RANS) equations in the literature. This nomenclature still persists, since some early RANS models assumed that $\overline{\rho}_{,t}$, $(\overline{\rho}\ \tilde{\mathbf{u}})_{,t}$ and $(\overline{\rho}\ \tilde{I})_{,t}$ were strictly zero. However, these variables can still follow macro patterns in flows such as coastal waves or travelling vortices.

Back in eq. (2.39), it can be noted that the influence of the turbulent fluctuations in the mean flow is restricted to the term $-\overline{\rho\ \mathbf{u}''\mathbf{u}''^T}$, which corresponds to the Reynolds stress tensor. This term is often several orders of magnitude greater than the Newtonian stress tensor $\overline{\boldsymbol{\tau}}_{\mathbf{n}}$ once turbulence emerges. While the Newtonian and the Reynolds stress tensor can appear to be disconnected, it can be proven that their sum is always equal to the total viscous stresses experienced by the mean flow ($\overline{\boldsymbol{\tau}}_{global}$):

$$\overline{\boldsymbol{\tau}}_{global} = \overline{\boldsymbol{\tau}}_{\mathbf{n}} - \overline{\rho\ \mathbf{u}''\mathbf{u}''^T}. \tag{2.41}$$

In flows within confined geometries, the previous formula can be used to obtain explicit bounds for $\overline{\boldsymbol{\tau}}_{global}$. An example of this will be presented in Chapter 4 for channel flows. In order to model the Reynolds stress tensor ($-\overline{\rho\ u_i''u_j''}$), different theories can be employed depending on the complexity of the problem [15]. One simple approach is given by the Boussinesq approximation, which is given by the following formula:

$$-\overline{\rho\ \mathbf{u}''\mathbf{u}''^T} = \nabla \cdot \left(\mu_t\left(\nabla\tilde{\mathbf{u}} + \nabla\tilde{\mathbf{u}}^T\right) - \frac{2}{3}\rho k_t \mathbf{I}\right). \tag{2.42}$$

In eq. (2.42), $\mathbf{I}$ again corresponds to an identity matrix, whereas μ_t and k_t correspond to the eddy viscosity and the turbulent kinetic energy respectively. Both of these quantities must be predicted by a further turbulence sub-model however, since the Boussinesq approximation only corresponds to a framework. It can be noted that eq. (2.42) is able to model the Reynolds stress tensor by only employing two variables, instead of its 9 original entries. While this approach can be restrictive for certain applications, the Boussinesq approximation has been found to work well for a large variety of wall-bounded flows and free-stream diffusion cases [27]. Furthermore, it has been estimated that CFD solvers can still improve substantially before the Boussinesq approximation becomes an obstacle. Duraisamy et al. (2016) [28] studied various 2-D and mildly 3-D flow cases in order to confirm that there existed an eddy viscosity field (μ_t) capable of generating the observed mean flow patterns.

In the case of the Reynolds-averaged energy equation, it can be noted that two turbulent terms now appear, namely $-\overline{\rho\ \mathbf{u}''\ I''}$ and $\overline{\boldsymbol{\tau}' : \nabla\mathbf{u}''}$. The first term corresponds to the turbulent heat transport by convection, which is analogue to the Reynolds stress tensor, whereas the second term represents the presence of turbulent viscous heating. The analogy between the Reynolds stress tensor and the turbulent heat diffusion is often employed to propose the following model:

$$-\overline{\rho\ \mathbf{u}''\ I''} = \frac{c_p \mu_t}{Pr_t}\ \nabla\widetilde{T}. \tag{2.43}$$

In eq. (2.43), Pr_t corresponds to the turbulent Prandtl number of the flow, which often varies between 0.8 and 1.2 in boundary layers [9, 29]. This model can be derived from the K-theory, which states that the diffusion of a scalar in a turbulent flow field is proportional to its mean flow gradients [27]. However, since $I \approx c_p T$, eq. (2.43) also manages to define a turbulent heat conductivity factor $\lambda_t = c_p \mu_t / Pr_t$.

Regarding the turbulent viscous heating term $\overline{\boldsymbol{\tau}'_{\mathbf{n}} : \nabla\mathbf{u}''}$ present in eq. (2.40), it can be proven that modelling this term directly is rather unnecessary, since a free-body diagram for the mean flow indicates that the viscous heating perceived by the flow is bound by the following balance [30]:

$$\overline{\boldsymbol{\tau}}_{global} : \nabla\widetilde{\mathbf{u}} = \overline{\boldsymbol{\tau}}_{\mathbf{n}} : \nabla\widetilde{\mathbf{u}} + \overline{\boldsymbol{\tau}'_{\mathbf{n}} : \nabla\mathbf{u}''}. \tag{2.44}$$

In eq. (2.44), $\overline{\boldsymbol{\tau}}_{global}$ corresponds to the total stresses experienced by the flow, as they were presented back in eq. (2.41). Since this term can be calculated directly from the Navier-Stokes equations, many CFD codes treat the amount of viscous heating experienced by a fluid as an energy source term (S_i) during the solution of the energy equation.

The complete set of RANS equations considering thermal coupling and the Boussinesq approximation can be found listed below:

$$\overline{\rho}_{,t} + \nabla\cdot(\overline{\rho}\ \widetilde{\mathbf{u}}) = 0, \tag{2.45}$$

$$(\overline{\rho}\ \widetilde{\mathbf{u}})_{,t} + \nabla\cdot\left(\overline{\rho}\ \widetilde{\mathbf{u}}\ \widetilde{\mathbf{u}}^T\right) = -\nabla\overline{P} + \nabla\cdot\overline{\boldsymbol{\tau}}_{global} + \overline{\mathbf{f}}, \tag{2.46}$$

$$\left(\overline{\rho}\ \widetilde{I}\right)_{,t} + \nabla\cdot\left(\overline{\rho}\ \widetilde{\mathbf{u}}\ \widetilde{I}\right) = -\overline{P}\left(\nabla\cdot\widetilde{\mathbf{u}}\right) + \overline{\boldsymbol{\tau}}_{global} : \nabla\widetilde{\mathbf{u}} + \nabla\cdot\left(\overline{\lambda}_{eff}\ \nabla\widetilde{T}\right) + \overline{S}_I, \tag{2.47}$$

$$\overline{\boldsymbol{\tau}}_{global} = (\mu + \mu_t)\left(\nabla\widetilde{\mathbf{u}} + \nabla\widetilde{\mathbf{u}}^T\right) - \frac{2}{3}\left(\mu\left(\nabla\cdot\widetilde{\mathbf{u}}\right) + \rho k_t\right)\cdot\mathbf{I}\ \left(= \overline{\boldsymbol{\tau}}_{\mathbf{n}} - \overline{\rho\ \mathbf{u}''\mathbf{u}''^T}\right), \tag{2.48}$$

$$\lambda_{eff} = \lambda + \frac{c_p \mu_t}{Pr_t}. \tag{2.49}$$

In the upcoming Sections, the special notation for the averaged flow variables ($\overline{x}$ and $\tilde{x}$) will be dropped, since it will be assumed that all the equations correspond to mean flow balances.

2.1.5. Alternatives to RANS Turbulence Modelling

While the RANS equations presented in the previous Section constitute one of the leading approaches in turbulence modelling, it can be noted that these equations require the user to specify at least one sub-model for the eddy viscosity μ_t. However, decades of research have proven that predicting the values of μ_t in turbulent flows requires substantial empirical knowledge, and that the development of strong universal models will require innovative ideas [31]. Due to these reasons, researchers must employ other methods to gain insights into the nature of turbulence. The following techniques are often employed by researchers to this end:

- **Direct Numerical Simulations (DNS):** DNS studies correspond to one of the most detailed sources of information regarding the nature of turbulence available. In this approach, the Navier-Stokes equations are discretized directly by using very refined grids, such that turbulence can emerge globally. Thanks to this technique, it is also possible to obtain mathematical correlations that would be impossible to measure experimentally. However, DNS studies are greatly limited by the grid resolution required to model turbulent flows. The scales of the smallest eddies present in turbulent flows were determined in the 1940's by Kolmogorov, who postulated that turbulent flows possess an energy cascade, where large eddies feed energy into smaller eddies, until a nearly isotropic regime is obtained [14–16]. The time and length scales defined by Kolmogorov were the following:

$$\eta = \left(\frac{\nu^3}{\epsilon}\right)^{1/4}, \tag{2.50}$$

$$t_\eta = \left(\frac{\nu}{\epsilon}\right)^{1/2}, \tag{2.51}$$

$$u_\eta = (\nu\epsilon)^{1/4}. \tag{2.52}$$

In eq. (2.50), η corresponds to the length scale of the smallest eddies, whereas ϵ corresponds to the volumetric energy being dissipated by turbulence. Due to the energy cascading process, ϵ is constant across all scales. The variable ν corresponds to the kinematic viscosity of the fluid: $\nu = \mu/\rho$. The associated time (t_η) and velocity scales (u_η) for the smallest eddies are then given by eqs. (2.51) and (2.52) respectively. The exponents present in eqs. (2.50-2.52) can be obtained from a direct dimensional analysis [1]. In order to complete the present derivation, it must be noted that the energy dissipated by turbulence (ϵ) has the following macro scale:

$$\epsilon \approx {U_0}^3/L_0. \tag{2.53}$$

In eq. (2.53), L_0 and U_0 correspond to the length and velocity scales of the largest eddies present in the flow. Replacing this equation into the former Kolmogorov scales reveals the following dependencies on the flow Reynolds number (Re_0):

$$\frac{\eta}{L_0} \approx Re_0^{-3/4}, \tag{2.54}$$

$$\frac{t_\eta}{T_0} = \frac{t_\eta}{L_0/U_0} \approx Re_0^{-1/2}, \tag{2.55}$$

$$\frac{u_\eta}{U_0} \approx Re_0^{-1/4}. \tag{2.56}$$

Based on the previous formulas, the grid requirements for a DNS study can be finally determined, since the ratio between the largest and the smallest scales present in a turbulent flow can be estimated. By noting that the Navier-Stokes equations correspond to a 3-D transient problem, it can be concluded that the complexity of a DNS grid has the following order:

$$\mathcal{O}_{DNS} = \mathcal{O}\left(\left(\frac{L_0}{\eta}\right)^3 \cdot \left(\frac{T_0}{t_\eta}\right)\right) \approx \mathcal{O}\left(Re_0^{11/4}\right). \tag{2.57}$$

Eq. (2.57) indicates that the level of grid refinement required by a DNS study is proportional to ${Re_0}^{2.75}$. However, the overall complexity of the computer simulations may be even higher, since the running times of most CFD solvers also grow non-linearly with the grid size. As a result, performing DNS studies for industrial applications is prohibitive yet, especially within the aerospace industry. However, DNS studies still remain one of the most valuable sources of information regarding the structure of turbulence.

- **Large Eddy Simulations (LES):** LES studies correspond to another CFD modelling technique present in the literature. These models attempt to mitigate the excessive grid requirements found in DNS studies by introducing subgrid-scale models (SGS), which represent the action of the smallest eddies present in the flow. While LES models require less computational resources than DNS, their grid resolution must still be high in order to capture a large portion of the turbulent eddies. Therefore, the issues found in DNS studies are only partially mitigated. The mathematical derivation of LES solvers also differs from the RANS equations, since LES models create spatial filters for the turbulent fluctuations, and only attempt to model those interactions which were missed by the coarser grid. The applications of LES models might grow substantially in the future nonetheless, since the computational resources available grow every year, and Machine Learning techniques are being employed in order to improve SGS models [32].

2.1.6. Boundary Layer Structures

Boundary layers correspond to one of the most important concepts in turbulence modelling, especially within the context of engineering applications. These structures are created when high-speed flows come into contact with walls, as it can be seen in Figure 2.1. Here, the flow must undergo a rapid transition from its non-slip boundary condition at the wall ($u_w = 0$) until the mean flow velocities are reached (u_∞). Although the scheme presented in Figure 2.1 is valid for both laminar and turbulent flows, it has been proven experimentally that the scales governing both cases are dramatically different. While laminar flows tend to form parabolic profiles spawning their respective geometries, turbulent boundary layers are confined to a narrow flow region, which is almost an exclusive function of the wall shear stresses.

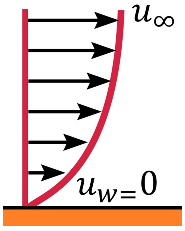


Figure 2.1: Schematic representation of a wall boundary layer [1].

The characteristic velocity profile of a turbulent boundary layer can be found depicted in Figure 2.2 as a function of dimensionless wall coordinates y^+ and U^+ [33]. These variables correspond to the dimensionless wall distance and the scaled velocities respectively. The conversion formulas are the following:

$$y^+ = y\,\frac{u_\tau}{\mu_w/\rho_w}, \tag{2.58}$$

$$u^+ = \frac{u}{u_\tau}, \tag{2.59}$$

$$u_\tau = \sqrt{\frac{\tau_w}{\rho}} = \sqrt{\frac{\mu_w}{\rho}\left|\nabla u\right|_w}. \tag{2.60}$$

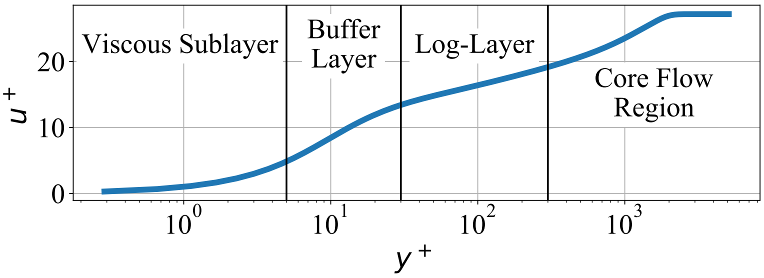


Figure 2.2: DNS velocity profile for a turbulent boundary layer at high Reynolds Numbers. Source: Borrel et al. (2013) [33].

In eqs. (2.58-2.60), u_τ corresponds to the so-called friction velocity, which is a kinematic representation of the wall shear stresses. The quantities ρ_w, μ_w and τ_w naturally refer to the values for the density, the viscosity and the shear stresses located at the walls. The meaning of each flow sub-region presented in Figure 2.2 will be described below:

- **Viscous Sublayer:** The Viscous Sublayer is a narrow flow region located next to the wall, which extends up to $y^+ < 5$. In this region, the turbulent Reynolds stresses are dampened due to the close presence of the wall, and a local laminar profile is obtained. However, the flow rapidly transitions into a turbulent regime as the wall distance increases.
- **Buffer Layer:** The second zone present in Figure 2.2 corresponds to the Buffer Layer ($5 < y^+ < 30$). This zone is characterized by the emergence of turbulence, and it can be proven that it presents the largest velocity gradients for the mean flow. While the Buffer Layer is often considered as a transition zone between the Viscous Sublayer and the Log-Layer, the presence of a sharp break in the velocity gradients for the flow indicates that turbulence reaches its peak of activity in this region.
- **Log-layer:** The third region present in Figure 2.2 corresponds to the Log-layer ($30 < y^+ < 300$). This region presents interesting mathematical properties, since the velocity profile becomes linear in the log-plane. While PDE's (partial differential equations) often present exponential solutions, it is rather remarkable that a system with the complexity of the Navier-Stokes equations still manages to yield a region presenting exponential decay rates. The Log-layer further corresponds to one of the largest zones present in a boundary layer, and it has historically been employed in CFD codes to specify turbulent wall boundary conditions.
- **Core Flow Region:** The last zone present in Figure 2.2 corresponds to the Core Flow Region ($y^+ > 300$). This region is marked by a transition between the log-layer and any far-field conditions. While the log-layer always presents a well-defined velocity profile, the Core Flow Region can suffer significant variations, since it is dependent on the overall flow field. Due to this reason, many CFD codes consider the log-layer to be the last region of a boundary layer and attempt to model the Core Flow Region directly.

Regarding the true accuracy of the velocity profile presented in Figure 2.2, it was noted during the study of several DNS databases published online [29, 33, 34] that boundary layers can still present a secondary dependence to the friction Reynolds number (Re_τ):

$$Re_\tau = \frac{\rho_w \, u_\tau \, L_0}{\mu_w}. \tag{2.61}$$

In eq. (2.61), it can be noted that Re_τ corresponds to a function of the wall shear stresses. While the influence of Re_τ in turbulent boundary layers is small, it is important to remember that including this parameter in Machine Learning systems is necessary in order to improve the baseline approximation shown in Figure 2.2, which represents the asymptotic state of turbulent boundary layers.

Turbulence Modelling in Variable-Property Flows

The study of turbulence modelling in variable-property flows constitutes an emerging branch of research, which has strong applications in industrial settings. Nearly every problem involving heat transfer in fluids has strong property gradients. Even incompressible fluids, such as water, can display variable viscosity (μ) profiles when subject to temperature changes. The research previously done at the TU Delft has determined that a simple scaling law can be established for variable property-flows obeying Morkovin's hypothesis ($\widetilde{a} \gg a''$) [17, 26]. In such flows, the traditional scaling given by the dimensionless wall coordinate (y^+) and the friction Reynolds number (Re_τ) is replaced by:

$$y^* = y^+ \, \frac{\mu_w}{\mu} \sqrt{\frac{\rho}{\rho_w}}, \tag{2.62}$$

$$Re_\tau^* = Re_\tau \, \frac{\mu_w}{\mu} \sqrt{\frac{\rho}{\rho_w}}. \tag{2.63}$$

Furthermore, the following transformation has been proposed to collapse the velocity profiles [9]:

$$u^* = \int_0^y \sqrt{\frac{\rho}{\rho_w}} \left(1 + \frac{y}{Re_\tau^*} \nabla Re_\tau^*\right) \frac{\partial u^+}{\partial y}\, dy. \tag{2.64}$$

In eq. (2.64), the integrand sub-term $\sqrt{\frac{\rho}{\rho_w}\frac{\partial u}{\partial y}}$ corresponds to the van Driest velocity transformation, which was originally developed for supersonic flows [9, 35]. It can thus be noted that the transformations given by eqs. (2.62- 2.64) correspond to an extension of this scaling law. During the present study, it will be proven that the variables y^* and Re_τ^* are particularly good predictors of the regimes adopted by variable-property flows within a Machine Learning context.

2.1.7. RANS Turbulence Models

This Section will present a theoretical review of the traditional RANS turbulence models employed in the literature which can be applied to variable property flows.

Overview

In order to provide a closure model for the RANS equations under the Boussinesq approximation (2.45-2.46), it is necessary to incorporate at least one sub-model for the eddy viscosity μ_t. In order to achieve this goal, several approaches can be defined. One of the first concepts defined historically was given by the Prandtl mixing length theory, which modelled the eddy viscosity as [30]:

$$\mu_t = \rho {l_m}^2 |\nabla \mathbf{u}|. \tag{2.65}$$

In eq. (2.65), l_m corresponds to the Prandtl mixing length itself, which represents a characteristic length scale for turbulence. Even though the values of l_m are highly case-dependent, several models have been developed for classical problems. For example, in turbulent boundary layers, one common approximation is to assume $l_m = \kappa y$, where κ corresponds to the von Karman constant ($\kappa = 0.4187$) [30]. In free shear layers, l_m can be assumed to be proportional to the width of the stream according to some empirical constant [27, 30]. However, it can be noted that l_m only corresponds to a close transformation of μ_t, and that predicting either quantity is essentially the same. Due to this reason, most turbulence models focus on predicting μ_t directly. The next subsections will describe the most important turbulence models found in the literature, which are also applicable to simulating variable-property flows. For all the cases presented, the values of y^+ and Re_τ present in the original models have been replaced by y^* and Re_τ^* respectively. A complete review of turbulence modelling in variable-property flows can be found in the work of Otero et al. (2018) [12], whereas a complete review of the turbulence models developed for incompressible flows can be found in the CFD textbook of Versteeg & Malalasekera (2007) [15].

MK Turbulence Model

The MK turbulence model corresponds to a special variant of the original $k - \epsilon$ turbulence model [36] developed by Myong and Kasagi [7] for low Reynolds numbers. This turbulence model assumes that the behavior of the eddy viscosity (μ_t) is governed by the turbulent kinetic energy present in the flow (k) and its volumetric dissipation rate (ϵ). The full system of equations defined by the MK turbulence model is the following:

$$(\rho\ k)_{,t} + \nabla \cdot (\rho\ \mathbf{u}\ k) = P_k - \rho\ \epsilon + \nabla \cdot \left(\left(\mu + \frac{\mu_t}{\sigma_k} \right) \nabla k \right), \tag{2.66}$$

$$(\rho\ \epsilon)_{,t} + \nabla \cdot (\rho\ \mathbf{u}\ \epsilon) = C_{\epsilon 1}\ P_k\ \frac{\epsilon}{k} - C_{\epsilon 2}\ f_\epsilon\ \rho\ \frac{\epsilon^2}{k} + \nabla \cdot \left(\left(\mu + \frac{\mu_t}{\sigma_\epsilon} \right) \nabla \epsilon \right). \tag{2.67}$$

Where,

$$P_k = \mu_t \ (\nabla \mathbf{u} : \nabla \mathbf{u})\,, \tag{2.68}$$

$$f_\epsilon = \left[1 - \frac{2}{9} e^{-\left(\frac{Re_t}{6}\right)^2}\right]\left[1 - e^{-\left(\frac{y^*}{5}\right)}\right]^2, \tag{2.69}$$

$$f_\mu = \left[1 - e^{-\left(\frac{y^*}{70}\right)}\right]\left[1 + \frac{3.45}{\sqrt{Re_t}}\right], \tag{2.70}$$

$$Re_t = \frac{\rho \ k^2}{\mu \ \epsilon}, \tag{2.71}$$

$$\mu_t = C_\mu \ f_\mu \ \rho \ \frac{k^2}{\epsilon}. \tag{2.72}$$

In eqs. (2.66-2.72), the values of the constants found are: $C_{\epsilon 1} = 1.4$, $C_{\epsilon 2} = 1.8$, $C_\mu = 0.09$, $\sigma_k = 1.4$ and $\sigma_\epsilon = 1.3$. Regarding eq. (2.72), it can be noted that μ_t is finally proportional to the non-linear ratio k^2/ϵ. This implies that there exist infinite corrections for the MK turbulence model which can be used to tweak the values of μ_t, since either k or ϵ could be at fault. At this point, it is important to note that, while k and ϵ are closely related to physical variables, they ultimately correspond to scalar fields which produce the right eddy viscosity profiles $\mu_t(\mathbf{x})$. One fundamental discrepancy present in the MK model is the scaling of k near the wall. While DNS data indicates that the turbulent kinetic energy (k_t) scales with $\mathcal{O}(y^3)$ near the wall, it can be proven that the values predicted by the MK turbulence model scale with $\mathcal{O}(y^2)$. Therefore, forcing the k-equation to match the DNS data would break the delicate balance of the MK model, since the original values predicted for μ_t are reasonably correct for Boundary Layer data. The wall boundary conditions for this model are:

$$k_w = 0, \tag{2.73}$$

$$\epsilon_w = \frac{\mu_w k_{w+1}}{\rho_w y_{w+1}^2}. \tag{2.74}$$

In eq. (2.74), the sub-index (w+1) refers to the first cell adjacent to the wall in a CFD code. These boundary conditions may take different forms depending on whether a Finite Volume or a Finite Difference scheme is employed. The differences between these methods can be found in references [15, 22].

SST Turbulence Model

The SST (Shear Stress Transport) model corresponds to an advanced turbulence model developed by Menter (1993) [37], which combines the best numerical properties of the $k - \epsilon$ [36] and $k - \omega$ [38] turbulence models. The main difference between these two model families is that $k - \omega$ models replace the ϵ-equation by the evolution of the eddy dissipation frequency ω. The equations for this model are the following:

$$(\rho \ k)_{,t} + \nabla \cdot (\rho \ \mathbf{u} \ k) = P_k^{lim} - \beta^* \ \rho \ k \ \omega + \nabla \cdot ((\mu + \sigma_k \ \mu_t) \nabla k)\,, \tag{2.75}$$

$$(\rho \ \omega)_{,t} + \nabla \cdot (\rho \ \mathbf{u} \ \omega) = \frac{\alpha \ \rho}{\mu_t} \ P_k - \beta \ \rho \ \omega^2 + (1 - F_1)\, CD_{k\omega} + \nabla \cdot ((\mu + \sigma_\omega \ \mu_t) \nabla \omega)\,. \tag{2.76}$$

With,

$$P_k^{lim} = min\,(P_k, 20 \ \beta^* \ \rho \ \omega \ k)\,, \tag{2.77}$$

$$CD_{k\omega} = 2 \frac{\rho \ \sigma_{\omega 2}}{\omega} \ (\nabla k : \nabla \omega)\,, \tag{2.78}$$

$$F_1 = tanh\left[min\left(max\left[\gamma_1,\gamma_2\right],\gamma_3\right)^4\right], \tag{2.79}$$

$$F_2 = tanh\left[min\left(2\ \gamma_1,\gamma_2\right)^2\right], \tag{2.80}$$

$$\gamma_1 = \frac{\sqrt{k}}{\beta^*\ \omega\ y}, \tag{2.81}$$

$$\gamma_2 = \frac{500\ \mu}{\rho\ y^2\ \omega}, \tag{2.82}$$

$$\gamma_3 = \frac{4\ \rho\ \sigma_{\phi 2} k}{y^2\ max\left(CD_{k\omega}, 10^{-20}\right)}, \tag{2.83}$$

$$\begin{bmatrix}\alpha_1\\ \alpha_2\end{bmatrix} = \frac{1}{\beta^*}\begin{bmatrix}\beta_1\\ \beta_2\end{bmatrix} - \frac{\kappa^2}{\sqrt{\beta^*}}\begin{bmatrix}\sigma_{\omega 1}\\ \sigma_{\omega 2}\end{bmatrix}, \tag{2.84}$$

$$\begin{bmatrix}\alpha\\ \beta\\ \sigma_k\\ \sigma_\omega\end{bmatrix} = \begin{bmatrix}\alpha_1\\ \beta_1\\ \sigma_{k1}\\ \sigma_{\omega 1}\end{bmatrix} F_1 + (1-F_1)\begin{bmatrix}\alpha_2\\ \beta_2\\ \sigma_{k2}\\ \sigma_{\omega 2}\end{bmatrix}, \tag{2.85}$$

$$\Omega = \parallel \mathbf{u} \parallel, \tag{2.86}$$

$$\mu_t = \frac{\rho\ C_\mu\ k}{max\left(C_\mu\ \omega\ ,\ \Omega\ F_2\right)}. \tag{2.87}$$

In can be noted that the previous set of equations is notoriously more complex than the MK model, and that it is marked by the presence of multiple threshold functions. The latter is due to the switch behavior employed by the SST model, which allows this model to retain the best numerical properties of both the $k-\epsilon$ and the $k-\omega$ turbulence models. The constants found in eqs. (2.75-2.87) take the following values: $\sigma_{k1} = 0.85$, $\sigma_{k2} = 1.0$, $\sigma_{\omega 1} = 0.5$, $\sigma_{\omega 2} = 0.856$, $\beta_1 = 0.075$, $\beta_2 = 0.0828$, $\beta^* = 0.09$ and $C_\mu = 0.31$. Furthermore, the wall boundary conditions for the SST model are:

$$k_w = 0, \tag{2.88}$$

$$\omega_w = \frac{60\mu_w}{\rho_w \beta_1 y_{w+1}^2}. \tag{2.89}$$

$v'^2 - f$ Turbulence Model

The $v'^2 - f$ model, proposed by Durvin in 1995 [39], corresponds to an advanced version of the standard $k-\epsilon$ model, which employs an additional transport equation for the wall-normal velocity fluctuations (v'^2), and an elliptic relaxation function f. The resulting equations are the following:

$$\left(\rho\ k\right)_{,t} + \nabla\cdot\left(\rho\ \mathbf{u}\ k\right) = P_k - \rho\ \epsilon + \nabla\cdot\left(\left(\mu + \frac{\mu_t}{\sigma_k}\right)\nabla k\right), \tag{2.90}$$

$$\left(\rho\ \epsilon\right)_{,t} + \nabla\cdot\left(\rho\ \mathbf{u}\ \epsilon\right) = \frac{1}{T_t}\left[f_{\epsilon 1}P_k - C_{\epsilon 2}\rho\epsilon\right] + \nabla\cdot\left(\left(\mu + \frac{\mu_t}{\sigma_\epsilon}\right)\nabla\epsilon\right), \tag{2.91}$$

$$\left(\rho\ v'^2\right)_{,t} + \nabla\cdot\left(\rho\ \mathbf{u}\ v'^2\right) = \rho k f - 6\frac{\rho v'^2}{k}\epsilon + \nabla\cdot\left(\left(\mu + \mu_t\right)\nabla v'^2\right), \tag{2.92}$$

$$L_t^2\nabla^2 f - f = \frac{1}{T_t}\left[\left(C_{f1} - 6\right)\frac{v'^2}{k} - \frac{2}{3}\left(C_{f1} - 1\right)\right] - C_{f2}\frac{P_k}{\rho\ k}, \tag{2.93}$$

$$T_t = max\left(\frac{k}{\epsilon}, 6\sqrt{\frac{\mu}{\rho\epsilon}}\right), \tag{2.94}$$

$$L_t = 0.23\ max\left(\frac{k^{3/2}}{\epsilon}, 70\left(\frac{\mu^3}{\rho^3\epsilon}\right)^{1/4}\right), \tag{2.95}$$

$$f_{\epsilon 1} = C_{\epsilon 1}\left(1 + 0.045\sqrt{\frac{k}{v'^2}}\right), \tag{2.96}$$

$$\mu_t = C_\mu \rho v'^2 T_t. \tag{2.97}$$

The constants found in eqs. (2.90-2.97) are $\sigma_k = 1.0$, $\sigma_\epsilon = 1.3$, $C_{\epsilon 1} = 1.4$, $C_{\epsilon 2} = 1.9$, $C_\mu = 0.22$, $C_{f1} = 1.4$ and $C_{f2} = 0.3$. Regarding the formula given for the eddy viscosity (μ_t), it can now be seen that the model depends on a time scale T_t, which is subject to a threshold function. The boundary conditions specified for this model are:

$$k_w,\ v'^2_w,\ f_w = 0, \tag{2.98}$$

$$\epsilon_w = \frac{\mu_w k_{w+1}}{\rho_w y^2_{w+1}}. \tag{2.99}$$

Spalart Allmaras (SA) Turbulence Model

The Spalart Allmaras (SA) turbulence model [8] corresponds to an innovative model which builds a direct transport equation for a surrogate eddy viscosity ($\hat{v}$). Despite its simplicity, the SA model can offer better or equal performance than the previous turbulence models in a wide variety of aerospace engineering applications [15]. This model has also been found to work exceptionally well for variable-property flows, since its governing equations are solved in a kinematic transformed space [12]. The governing equations of the SA turbulence model are the following:

$$\hat{v}_{,t} + \nabla\cdot(\mathbf{u}\ \hat{v}) = c_{b1}\hat{S}\hat{v} - c_{w1}f_w\left(\frac{\hat{v}}{y}\right)^2 + \frac{c_{b2}}{c_{b3}}(\nabla\hat{v})^2 + \frac{1}{c_{b3}}\nabla\cdot((v+\hat{v})\nabla\hat{v}), \tag{2.100}$$

$$\hat{S} = S + \frac{\hat{v}}{\kappa^2 y^2} f_{v2}, \tag{2.101}$$

$$S = \sqrt{2W_{ij}W_{ij}}, \tag{2.102}$$

$$W_{ij} = \frac{1}{2}\left(\nabla\mathbf{u} - (\nabla\mathbf{u})^T\right), \tag{2.103}$$

$$f_{v2} = 1 - \frac{\chi}{1 + \chi f_{v1}}, \tag{2.104}$$

$$f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3}, \tag{2.105}$$

$$\chi = \frac{\hat{v}}{v}, \tag{2.106}$$

$$f_w = g\left(\frac{1 + c_{w3}^6}{g^6 + c_{w3}^6}\right)^{1/6}, \tag{2.107}$$

$$g = r + c_{w2}\left(r^6 - r\right), \tag{2.108}$$

$$r = \frac{\hat{\nu}}{\hat{S}\kappa^2 y^2}, \tag{2.109}$$

$$\mu_t = \rho\hat{\nu} f_{v1}, \tag{2.110}$$

$$c_{w1} = \frac{c_{b1}}{\kappa^2} + \frac{1 + c_{b2}}{c_{b3}}. \tag{2.111}$$

The constants for the SA turbulence model are $c_{b1} = 0.1355$, $c_{b2} = 0.622$, $c_{b3} = 2/3$, $c_{v1} = 7.1$, $c_{w2} = 0.3$, $c_{w3} = 2.0$ and $\kappa = 0.41$. The only wall boundary condition required for this model is $\hat{\nu}_w = 0$, which can be derived from eq. (2.110) by noting that μ_t is also zero at the wall.

Cess Algebraic Model

The Cess model corresponds to an algebraic correlation which was originally developed to predict the eddy viscosity profiles (μ_t) of fully-developed pipe flows [12, 40], yet it was later extended to channel flows by Hussain and Reynolds [12, 41]. The Cess model takes a different form in each case, since a correlation needs to be formed for every studied geometry. The expression developed for channel flows by Hussain was the following:

$$\frac{\mu_t}{\mu} = \frac{1}{2}\left[1 + \frac{\kappa^2 {Re_\tau^*}^2}{9}\left(2y - y^2\right)^2\left(3 - 4y + 2y^2\right)^2\left(1 + e^{-y^*/A}\right)^2\right]^{1/2} - \frac{1}{2}. \tag{2.112}$$

In eq. (2.112), the constants κ and A take the values of 0.41 and 25.4 respectively. Regarding the overall applicability of algebraic models, it must be noted that these models constitute an excellent alternative to synthesize the results found in DNS simulations. Furthermore, these models can also be incorporated into larger CFD codes as advanced wall Boundary Conditions or internal sub-models for specific features. Therefore, the elaboration of algebraic correlations constitutes an important modelling resource.

2.2. Parameter Optimization

2.2.1. General Paradigm

This Section will describe the general principles of parameter optimization, which are applicable to both Machine Learning and CFD optimization. In each of these fields, it is often necessary to optimize models containing a large distribution of $\boldsymbol{\beta}$ parameters. The parameters under study can range from the weights of a Neural Network to the corrections required by a RANS turbulence model. Regardless of the specific application, these problems can be formulated as finding an optimal parameter distribution $\boldsymbol{\beta}$ which minimizes a given cost function $\mathcal{J}$. The cost function $\mathcal{J}$ should reflect the problem objectives as clearly as possible. If multiple optimization targets exist, one common practice is to create an individual cost function for every objective ($\mathcal{J}_i$), and then to define the total cost function as a weighted sum:

$$\mathcal{J}_{total} = \sum_{i}^{N} \alpha_i\, \mathcal{J}_i. \tag{2.113}$$

Where the optimal parameter distribution ($\boldsymbol{\beta}_{opt}$) is given by:

$$\boldsymbol{\beta}_{opt} = argmin\left[\, \mathcal{J}_{total}\left(\boldsymbol{\beta}\right)\,\right]. \tag{2.114}$$

In eq. (2.113), the terms α_i correspond to the weights assigned to each objective. These objectives can include the fulfillment of mathematical conditions that may be hard to define otherwise, such as reducing the magnitude of the $\boldsymbol{\beta}$ parameters. The latter technique is known as regularization, and it is often employed to prevent an issue known as over-fitting, where the $\boldsymbol{\beta}$ parameters of a system grow excessively in order to produce marginal improvements. An example of this issue can be found in Figure 2.3, which presents a parabolic distribution containing noise. Here it can be seen that the data-fitting system wrongly employed a high-order polynomial only to capture the noise, and thus obtained an incorrect solution presenting wide oscillations. The previous example corresponds to a clear case of over-fitting, which can be mitigated through the use of regularization. Similar cases of over-fitting can be found while training Deep Neural Networks or building corrections for a RANS turbulence model. The formulation of a cost function employing regularization is the following:

$$\mathcal{J}_{total} = \sum_{i}^{N} \alpha_i \, \mathcal{J}_i + \lambda \, \|\boldsymbol{\beta}\|^2 . \tag{2.115}$$

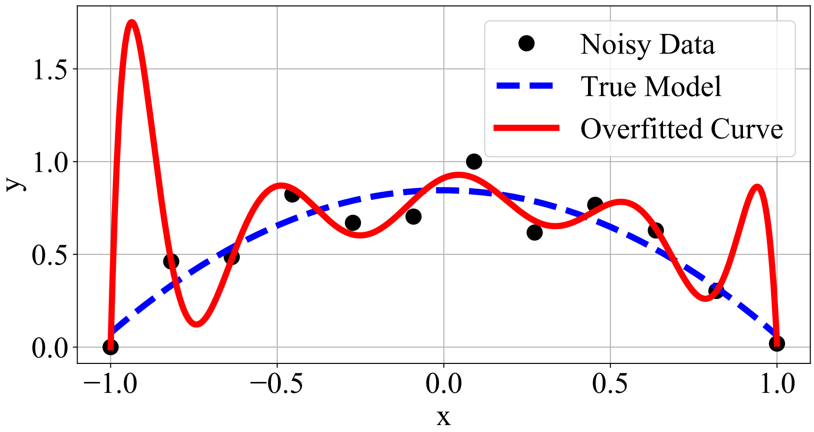


Figure 2.3: Example of an overfitted model employing a high-order polynomial.

In eq. (2.115), λ corresponds to the regularization hyper-parameter being employed. The values of λ are often subject to a secondary optimization problem, since they are highly problem-sensitive. The different techniques employed in Machine Learning to tune the hyper-parameters of a model will be discussed in Section 2.3.5. The techniques presented therein can also be employed to perform model selection, i.e., to choose the right architecture for a problem.

In order to find the optimal parameter distribution ($\boldsymbol{\beta}_{opt}$) required by a system, several algorithms can be employed. The most common algorithms employed in the literature corresponds to Gradient-descent and Hessian optimization. These techniques will be presented next in Sections 2.2.2 and 2.2.4 respectively.

2.2.2. Gradient-based Optimization

Gradient-based optimization corresponds to one of the leading approaches today in parameter optimization, especially given its success in deep learning applications [42]. These algorithms scale exceptionally well to large-scale systems and are able to find optimum solutions through a simple iterative approach. The basic intuition behind gradient-based optimization can be found depicted in Figure 2.4. In this figure, it can be noted that the optimizer continuously travels downhill across the landscape created by the cost function $\mathcal{J}_{total}$ until a local minimum is reached. While the previous problem appears to work intuitively, there exists a mathematical proof that gradient-based optimization also works in N-dimensional spaces [43]. Furthermore, many advanced optimization algorithms reverse to gradient-descent optimization when facing convergence difficulties [5]. Gradient-based optimization is also known as Jacobian optimization, since the Jacobian matrix/vector of a system corresponds to the gradient $\nabla_{\beta}\mathcal{J}$.

One of the greatest obstacles encountered in optimization problems is the presence of local minima, as it can be seen in Figure 2.4, where the algorithm narrowly avoided the wrong valley. This issue is not only restricted to Jacobian optimization however, since it is also applicable to other optimization algorithms. The only alternative to prove that a system has reached its global minimum would be to build an analytic expression for the cost function $\mathcal{J}_{total}$, such that the roots of its Jacobian matrix could be evaluated. However, even if such expression could be built, a mathematical proof providing the number of roots present in the system would still be required. The previous approach is clearly intractable in large-scale optimization problems, especially when threshold functions or advanced mathematical constraints are involved. The extreme non-linear nature of the RANS equations also poses a problem in CFD optimization problems. Therefore, other statistical methods are employed in practice to assess whether a function has reached its global minimum or not. One of the best methods is to define multiple initialization trials for the $\boldsymbol{\beta}$ parameters, and then to observe whether the system converges to the same local minimum or not. This procedure can be repeated N-times, until a solution can be accepted as the global minimum. The algorithms employed to compute the Jacobian matrix of a system will be presented in the next Section.

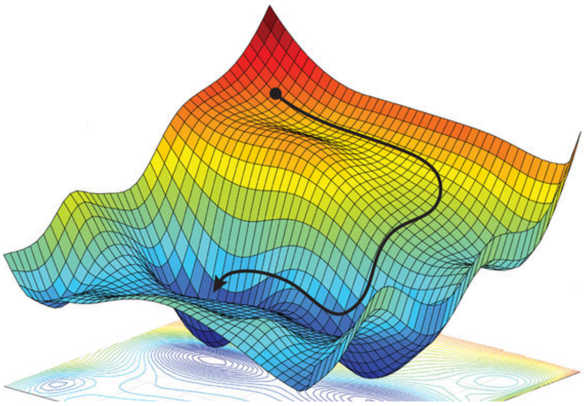


Figure 2.4: Example of Gradient Descent optimization across a 3-D landscape. Source: Hutson (2018) [44].

2.2.3. Computation of the Jacobian Matrix

From a practical perspective, the implementation of gradient-based optimization requires the computation of the Jacobian matrix/vector $\nabla_{\beta}\mathcal{J}$, which is required to drive the process. Since building the Jacobian matrix of a system can be challenging, several automatic methods have been developed. For example, in CFD optimization, a special procedure called Discrete Adjoint Method can be employed to compute the Jacobian matrix of a system constrained by governing equations. Thanks to this procedure, it is possible to find an optimization direction that complies with the RANS equations automatically. A detailed presentation of the Discrete Adjoint method can be found in the next sub-section.

In a general Machine Learning context, a computational procedure called Automatic Differentiation (AD) is often employed to calculate the Jacobian matrix of a system. AD algorithms track the operations required to calculate the cost function $\mathcal{J}_{total}$ in a computer program, and then differentiate the machine code in order to obtain the Jacobian matrix directly. While this procedure is computationally expensive in applications involving the solution of linear equations, AD libraries have proven to be extremely effective in deep learning applications. Another possible method is to employ symbolic algebra solvers, such as Sympy, in order to build a cost function and to obtain its Jacobian matrix. During the present thesis, this approach was partially employed in order to build various solution components required by CFD optimization algorithms, as it will be seen in Chapter 3. Finally, it must be noted that the Jacobian matrix $\nabla_{\beta}\mathcal{J}$ can also be built through a brute-force finite difference approach:

$$\left[\frac{\partial \mathcal{J}}{\partial \beta_i}\right] \approx \left[\frac{\mathcal{J}(\boldsymbol{\beta}+\Delta_i)-\mathcal{J}(\boldsymbol{\beta}-\Delta_i)}{2\Delta_i}\right]. \tag{2.116}$$

In eq. (2.116), the variable Δ_i correspond to an infinitesimal increment applied to every parameter β_i whose gradient is being calculated. While the values of Δ_i should be as small as possible in principle, it must be noted that infinitesimal Δ_i increments can clash with the Floating-Point Arithmetic Precision of a computer, and thereby produce noisy gradients. Beyond these difficulties, the use of eq. (2.116) is greatly avoided since it corresponds to a brute-force approach which does not scale well to N-dimensions. It was confirmed in practice that eq. (2.116) was orders of magnitude slower than the leading methods employed in CFD optimization. However, eq. (2.116) remains useful for validation purposes, since only the cost function $\mathcal{J}_{total}$ needs to be defined.

Discrete Adjoint Method

The present Section will describe the Discrete Adjoint Method, which can be used to obtain the Jacobian matrix of any optimization problem constrained by PDE's (Partial Differential Equations). This method is required in CFD systems, since they are constrained by the RANS equations. Other applications of this method can also be found in structural mechanics.

In order to start the derivation process, the governing equations present in the system must be first expressed as:

$$\mathcal{R}\left(\mathbf{W}\left(\boldsymbol{\beta}\right),\boldsymbol{\beta}\right) = 0. \tag{2.117}$$

In eq. (2.117), the vector $\mathcal{R}$ contains the residuals obtained for every governing equation defined, whereas the vector $\mathbf{W}$ contains all the physical DOF's (Degrees-of-Freedom) present in the system. In CFD applications, $\mathbf{W}$ must contain variables such as the pressures, the velocities and any further turbulence scalars being modelled. Since these variables can change according to the solution parameters ($\boldsymbol{\beta}$), the functional dependency $\mathbf{W}\left(\boldsymbol{\beta}\right)$ can be justified in practice. The residuals $\mathcal{R}$ should naturally be zero at every solution stage, since otherwise the governing equations would be unfulfilled. At this point, it is important to note that the CFD governing equations can be discretized either at the start or at the end of the present derivation process. The results obtained will be the same, as long as the discretization schemes employed are identical. During the present thesis, the Finite Difference Method was employed to discretize the CFD systems studied. An excellent presentation of this method can be found in ref. [22].

The Discrete Adjoint Method now continues by assuming that $\mathcal{R}$ corresponds to a vector which can be differentiated. The chain rule can be applied therein to prove that eq. (2.117) implies the following:

$$\frac{\partial \mathcal{R}}{\partial \mathbf{W}}\frac{\partial \mathbf{W}}{\partial \boldsymbol{\beta}} + \frac{\partial \mathcal{R}}{\partial \boldsymbol{\beta}} = 0. \tag{2.118}$$

In eq. (2.118), it must be noted that $\partial\mathcal{R}/\partial\mathbf{W}$ and $\partial\mathcal{R}/\partial\boldsymbol{\beta}$ correspond to explicit derivatives of the governing equations which can be obtained through a rigorous derivation process. These matrices are often computed in the literature by employing Automatic Differentiation libraries, although in the present thesis they were generated by a Sympy script. The matrix $\partial\mathbf{W}/\partial\boldsymbol{\beta}$ corresponds to an array of unknown gradients which can be calculated directly from eq. (2.118). However, the Discrete Adjoint Method avoids such process, since it is computationally expensive. In order to understand the possible alternative, the Jacobian matrix of the system must be first defined as:

$$\nabla_{\beta}\mathcal{J}\left(\mathbf{W}\left(\boldsymbol{\beta}\right),\boldsymbol{\beta}\right) = \frac{\partial \mathcal{J}}{\partial \mathbf{W}}\frac{\partial \mathbf{W}}{\partial \boldsymbol{\beta}} + \frac{\partial \mathcal{J}}{\partial \boldsymbol{\beta}}. \tag{2.119}$$

It can be noted that eq. (2.119) corresponds to a direct application of chain rule differentiation to the cost function, which is similar to the process followed in eq. (2.118) for $\mathcal{R}$. The Discrete Adjoint Method now employs the Adjoint State Theorem from linear algebra to prove that $\nabla_{\beta}\mathcal{J}$ can also be calculated using a vector of adjoint variables ($\boldsymbol{\Psi}$) according to the following equation:

$$\nabla_{\beta}\mathcal{J} = \boldsymbol{\Psi}\cdot\frac{\partial \mathcal{R}}{\partial \boldsymbol{\beta}} + \frac{\partial \mathcal{J}}{\partial \boldsymbol{\beta}}. \tag{2.120}$$

Where $\boldsymbol{\Psi}$ can be obtained from the following system of equations:

$$\left[\frac{\partial \mathcal{R}}{\partial \mathbf{W}}\right]^T \cdot \boldsymbol{\Psi}^T = -\left[\frac{\partial \mathcal{J}}{\partial \mathbf{W}}\right]^T. \tag{2.121}$$

It can be proven that the transposition operations specified in eq. (2.121) are consistent with the dimensions of the system. By analyzing eq. (2.121), it is also possible to prove that $\mathbf{\Psi}$ strictly corresponds to a vector, whereas $\partial\mathcal{R}/\partial\mathbf{W}$ corresponded to a matrix of larger size. Therefore, the Discrete Adjoint Method brings a substantial performance improvement to the system.

Regarding the meaning of the Adjoint State Theorem, it must be noted that this method is an application of the Duality Principle in mathematics. This principle states that there exist two possible approaches to an optimization problem. In the present case, the cost function $\mathcal{J}\left(\mathbf{W}\left(\boldsymbol{\beta}\right),\boldsymbol{\beta}\right)$ can also be written as $\mathcal{J}\left(\mathbf{W}\left(\mathcal{R}\right),\boldsymbol{\beta}\right)$. The latter implies that the values of $\mathbf{W}$ is guided by the residual vectors $\mathcal{R}$, which is true in practice. Based on the last definition, eq. (2.120) can be obtained as a direct product of chain rule differentiation, given that $\mathbf{\Psi} = \partial\mathcal{J}/\partial\mathcal{R}$. Eq. (2.121) is also consistent with such assumption. A formal proof of the Adjoint State Theorem can be found in the next sub-section.

Proof of the Adjoint State Theorem

In order to prove the identity behind eqs. (2.120,2.121), the required linear algebra proof was reconstructed. However, before addressing this proof, it is better to rewrite the previous process using a simplified notation. Under this context, eq. (2.118) can be first reformulated as:

$$A \cdot x = -\,f. \tag{2.122}$$

Where x corresponds to the former matrix of unknowns $\partial\mathbf{W}/\partial\boldsymbol{\beta}$, A to the matrix $\partial\mathcal{R}/\partial\mathbf{W}$ and f to the matrix $\partial\mathcal{R}/\partial\boldsymbol{\beta}$. The expression for the Jacobian matrix can be further rewritten as:

$$\nabla_{\beta}\mathcal{J}\left(\mathbf{W}\left(\boldsymbol{\beta}\right),\boldsymbol{\beta}\right) = s \cdot x + d. \tag{2.123}$$

Where s and d corresponds to $\partial\mathcal{J}/\partial\mathbf{W}$ and $\partial\mathcal{J}/\partial\boldsymbol{\beta}$ respectively. Based on the new notation, eqs. (2.120-2.121) stated that:

$$\nabla_{\beta}\mathcal{J}\left(\mathbf{W}\left(\boldsymbol{\beta}\right),\boldsymbol{\beta}\right) = \Psi \cdot f + d. \tag{2.124}$$

Where,

$$A^{T} \cdot r^{T} = -s^{T}. \tag{2.125}$$

In eqs. (2.124-2.125), Ψ corresponds to the vector of adjoint-state variables as before. The Adjoint State Theorem will now be proven using this simplified notation. At this point, it can be noted that the current proof reduces to showing that $s \cdot x = \Psi \cdot f$. In order to prove this identity, it is convenient to use eq. (2.122) to write $(s \cdot x)$ as:

$$s \cdot x = s \cdot \left(-A^{-1} \cdot f\right) = (-s) \cdot A^{-1} \cdot f. \tag{2.126}$$

By transposing eq. (2.125), it can be further shown that:

$$-\,s = \Psi \cdot A. \tag{2.127}$$

Replacing eq. (2.127) back in eq. (2.126) yields a clear proof of the Adjoint State theorem:

$$s \cdot x = (-s) \cdot A^{-1} \cdot f = (\Psi \cdot A) \cdot A^{-1} \cdot f = \Psi \cdot f. \tag{2.128}$$

In eq. (2.128), it can be seen that the Adjoint State Theorem has been finally proven, which concludes this Section.

2.2.4. Hessian Optimization

While Jacobian optimization corresponds to the leading approach in Machine Learning, Hessian optimization also constitutes a strong alternative in certain applications. The Hessian matrix of a system corresponds to the square tensor formed by all the second derivatives of the cost function ($\nabla^{2}_{\beta}\mathcal{J}$). Thanks to this matrix, it is possible to predict how the gradients of a system will change. In cases where the Hessian matrix is invertible, Newton's Method [45] can be employed to find the location of the closest local minimum:

$$\boldsymbol{\beta}_n = \boldsymbol{\beta}_{n-1} - \alpha\left[\nabla^2_{\beta}\mathcal{J}_{n-1}\right]^{-1} \cdot \nabla_{\beta}\mathcal{J}_{n-1}. \tag{2.129}$$

In eq. (2.129), the sub-index $(n-1)$ refers to the properties of the system at the previous iteration, whereas the variable α refers to the relaxation factor being employed. The introduction of a relaxation factor is optional under this context, since Newton's method can still converge to a local minimum while employing $\alpha = 1$. However, the predictions made without a relaxation factor in highly non-linear problems can lead to excessive oscillations, or even to divergence. It has been proven that the initial conditions given to a Hessian optimizer can strongly compromise its convergence properties [45].

Beyond Newton's method, other Hessian optimization algorithms can be found in the literature. Among these methods, the BFGS (Broyden–Fletcher–Goldfarb–Shanno) algorithm is often considered as one of the best alternatives, since it can even optimize problems where the exact Hessian is not invertible [5, 46]. However, the BFGS algorithm presents certain practical drawbacks. The first issue is that this method employs a relaxation factor α which is obtained through explicit line-search, which is a form of brute-force evaluation. Therefore, the BFGS algorithm can spend significant time searching for appropriate parameters, especially in CFD systems subject to multiple RANS equations. Another issue is that the approximate inverse Hessian matrix constructed by the BFGS method can produce unnecessary oscillations in systems containing spatially distributed parameters, since the average effect of the projected changes can still be perceived as positive by the optimizer. This issue was confirmed in practice while optimizing RANS turbulence models, and it can also be found reported in [47]. In contrast, Jacobian optimization algorithms tend to produce smooth spatial distributions, since the gradients of a smooth initial distribution rarely present perturbations.

Regarding the computation of the Hessian matrix, it must be noted that obtaining this matrix is orders of magnitude more expensive than calculating the Jacobian matrix/vector. This can be understood by noting that the Hessian matrix is obtained by re-differentiating every entry present in the Jacobian matrix by each parameter $\boldsymbol{\beta}$ present in the system. Storing the resulting Hessian Matrix in the RAM memory of a computer can also pose a challenge in large-scale systems. Regarding the running times, the tests carried out during the present thesis showed that the calculation of a Hessian matrix could take 10-20 seconds, whereas the Jacobian matrix could be obtained in a few milliseconds. The complexity of the Hessian matrix is not only limited to its high number of entries, but is also due to the nature of the operations involved. For example, in CFD applications, Giannakoglou et al. (2012) [48] showed that the fastest alternative to compute the Hessian matrix of a RANS model is given by the following equation:

$$\begin{aligned}\nabla^2_{\beta}\mathcal{J}_{n-1} = \frac{D^2\mathcal{J}}{D\beta_i D\beta_j} = \frac{\partial^2\mathcal{J}}{\partial\beta_i\partial\beta_j} + \Psi_n\frac{\partial^2\mathcal{R}_n}{\partial\beta_i\partial\beta_j} + \left(\frac{\partial^2\mathcal{J}}{\partial W_k\partial W_m} + \Psi_n\frac{\partial^2\mathcal{R}_n}{\partial W_k\partial W_m}\right)\frac{\partial W_k}{\partial\beta_i}\frac{\partial W_m}{\partial\beta_j}\\ + \left(\frac{\partial^2\mathcal{J}}{\partial\beta_i\partial W_k} + \Psi_n\frac{\partial^2\mathcal{R}_n}{\partial\beta_i\partial W_k}\right)\frac{\partial W_k}{\partial\beta_j} + \left(\frac{\partial^2\mathcal{J}}{\partial\beta_j\partial W_k} + \Psi_n\frac{\partial^2\mathcal{R}_n}{\partial\beta_j\partial W_k}\right)\frac{\partial W_k}{\partial\beta_i}.\end{aligned} \tag{2.130}$$

The nomenclature employed in eq. (2.130) is consistent with the derivation of the Discrete Adjoint Method for the Jacobian matrix presented in Section 2.2.3. The Einstein notation of the original publication was kept due to the high complexity of the equation presented. It can be noted that computing the Hessian matrix of a CFD system requires obtaining several third-rank order tensors, such as $\partial^2\mathcal{R}_n/\partial W_k\partial W_m$. These tensors are highly expensive to compute, and require extremely efficient sparse matrix methods. Even for a basic CFD model containing 10 000 discrete points, the tensor $\partial^2\mathcal{R}_n/\partial W_k\partial W_m$ would have at least 1 trillion entries. Furthermore, it can be noted that the equation for the Hessian matrix requires the direct computation of $\partial\mathbf{W}/\partial\boldsymbol{\beta}$ from eq. (2.118), which had previously been avoided.

In summary, Hessian optimization corresponds to a strong alternative to Jacobian optimization in problems where the Hessian matrix is invertible, since Newton's method can offer exceptional convergence rates. In such cases, the computational overhead per iteration given by Hessian methods can be easily overcome. However, none of the problems studied during the present thesis were well-posed for Hessian optimization. Machine Learning systems often present ill-posed Hessian matrixes, since many of its rows are nearly zero due to the small contributions made by secondary parameters to the solution ($\partial\mathcal{J}/\partial\beta_i \approx 0$). Furthermore, the threshold operators $f(x) = max(a + bx,\ c + dx)$ common in Machine Learning always present null Hessian matrices, despite their sharp non-linearity. In the CFD optimization problems formulated in Chapters 3 and 4, the resulting Hessian matrixes were also non-invertible, since many spatially distributed parameters had a negligible effect on the cost function. The BFGS method was further discarded due to the (confirmed) risk of obtaining non-smooth spatial $\boldsymbol{\beta}$ distributions. As a result, only the Gradient-based optimizers described in the next Section were considered during the present thesis.

2.2.5. Jacobian Optimization Algorithms

The present Section will discuss the leading algorithms employed today in Jacobian optimization, starting from their basic components. The algorithms presented in this Section also constitute the leading alternatives in Hessian-free optimization, since they can operate based only on the Jacobian matrix. As it was discussed before, Hessian optimization presented several practical drawbacks, such as being computationally expensive and not always being able to yield benefits. Therefore, the use of Jacobian optimizers is greatly favored in the Machine Learning community.

Vanilla Gradient-Descent

The Vanilla Gradient-Descent algorithm corresponds to the first method defined in Jacobian optimization. This technique states that the optimum solution for a system can be found by marching downhill against the gradients of the cost function at a fixed learning rate (α) [3]:

$$\boldsymbol{\beta}_n = \boldsymbol{\beta}_{n-1} - \alpha \left(\nabla_{\beta}\mathcal{J}\right)_{n-1}. \tag{2.131}$$

In eq. (2.131), the sub-index ($n-1$) refers to the properties found in the previous iteration, as it was mentioned before. Despite its simplicity, Vanilla Gradient-Descent corresponds to one of the few optimization algorithms which is mathematically guaranteed to make progress given a small learning rate α. Several complex optimization libraries reverse back to this technique when failing to make progress [5]. However, Vanilla Gradient-Descent algorithms presents several important drawbacks. The first issue is the challenge of finding an appropriate learning rate α. The learning rate has to be large enough in order to let the optimizer make progress, yet it cannot compromise the stability of the algorithm. Searching for an appropriate learning rate (α) can become a hyper-parameter optimization problem on its own right. Furthermore, it has been proven that the optimal learning rate for an optimization problem is not constant. An example of this issue is presented in Figure 2.5, where the optimal learning rates for an optimization problem based on the Rosenbrock function [2] were calculated using the Bold Drive Method presented in Section 2.2.5. Here, it can be seen that the optimal learning rate decreased as the iterations advanced. Due to this reason, many optimization libraries allow the user to define a decaying exponent for the learning rate (α) [42]. However, calibrating such exponent is a challenging task even for dedicated hyper-parameter optimization algorithms.

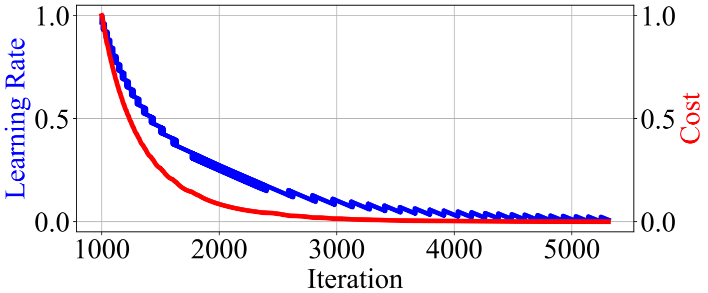


Figure 2.5: Estimation of the ideal learning rates for an optimization problem based on the Rosenbrock function [2].

The greatest issue found in Vanilla Gradient-descent algorithms however is the exponential degradation of its convergence properties. In Figure 2.6, it can be seen that the algorithm quickly stagnates as it approaches the global minimum of a parabolic curve (x^2). Defining a convergence precision for Vanilla Gradient-Descent algorithms can also be challenging, since it cannot be mathematically proven that the stagnation suffered by the cost function is not caused by the small learning rates being employed. Furthermore, Vanilla Gradient-descent algorithms are prone to become stuck in saddle points; if they exist. A graphical description of saddle points can be found in Figure 2.7 for a cubic function (x^3). Here, it can be seen that the function suffers a concavity change at $x = 0$, which is marked by the presence of a zero-gradient although this function does not present a global minimum. This issue can be overcome by adding momentum to the gradient-descent algorithm, as it will be seen in the next sub-section. Alternatively, an aggressive learning rate (α) could also manage to by-pass the saddle point by accident. Once a saddle-point is by-passed, the solution quickly diverges away from the local stagnation point.

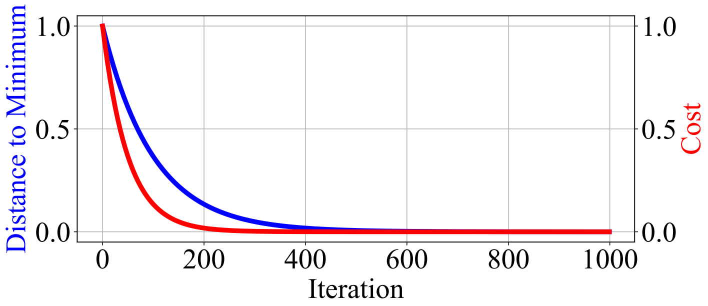


Figure 2.6: Exponential degradation of the convergence rates for the Vanilla Gradient-Descent algorithm during the optimization of a parabolic curve (x^2).

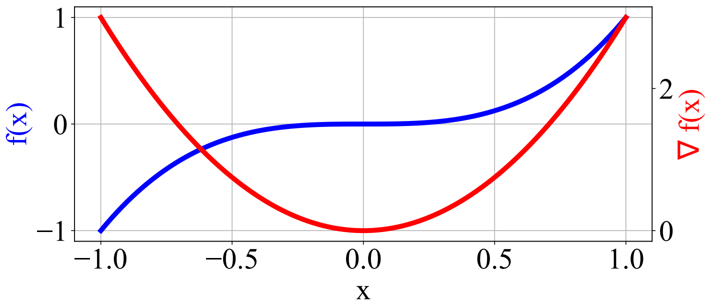


Figure 2.7: Saddle point obtained at $x = 0$ for a cubic function (x^3).

Gradient-Descent with Added Momentum

Since Vanilla Gradient-Descent algorithms present an exponential degradation in their convergence rates, several alternative solutions have been studied. One of the simplest solutions proposed has been to add momentum or gradient-inertia to the optimizers. The equations obtained using this modification are the following [3]:

$$\boldsymbol{\beta}_n = \boldsymbol{\beta}_{n-1} - \alpha \, \mathbf{m}_n, \tag{2.132}$$

$$\mathbf{m}_n = \frac{c_1 \mathbf{m}_{n-1} + (1 - c_1) \nabla_\beta \mathcal{J}_n}{1 - c_1^n}. \tag{2.133}$$

In eq. (2.132), it can be seen that the system is now driven by the accumulated gradient $\mathbf{m}_n$, which is calculated from eq. (2.133). The inertial memory of the system is controlled by the parameter c_1 present in eq. (2.133). For $c_1 = 0$, the original Vanilla Gradient-Descent algorithm is recovered, whereas in the limit $c_1 \rightarrow 1$, the system can remember the strongest gradients for a long time. Both extreme values for c_1 leads to poor convergence properties however, and the recommended value of

c_1 is often taken to be 0.9 [49]. The denominator present in eq. (2.133) only corresponds to a bias correction term, which helps the algorithm to initialize its gradient-memory using the Jacobian matrices obtained during the first iterations. The true effects of gradient-descent with added momentum can be observed in Figure 2.8, where the optimization example for the Rosenbrock function is revisited. Here, it can be seen that the addition of momentum to the system substantially improves its convergence properties. In general, it has been estimated that momentum can increase the convergence rates quadratically [49]. During the present thesis, the addition of momentum to the CFD optimizers employed decreased the running times by a factor of 3, even after introducing aggressive search schemes for the optimal learning rates. As a result, the addition of momentum to gradient-descent algorithms has established itself as an essential strategy for most researchers. However, more advanced optimization algorithms are required, since the addition of momentum is only a partial solution to the problems encountered in Vanilla Gradient-Descent.

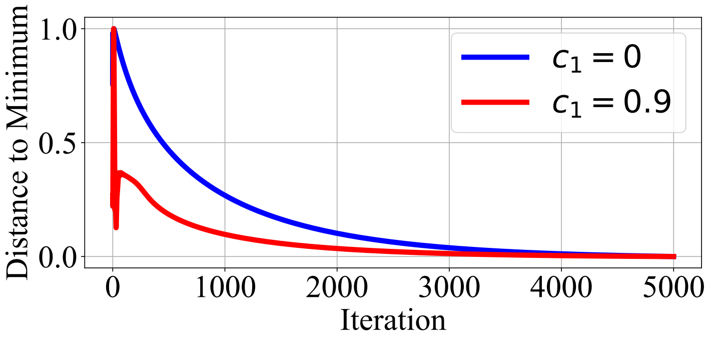


Figure 2.8: Performance improvements obtained during the optimization of the Rosenbrock function after adding momentum to the system ($c_1 = 0.9$).

Adam Optimizer

The Adam optimizer corresponds to one of the leading alternatives employed today in Machine Learning optimization, and it is often recommended as the default choice for Deep Learning applications [3]. This algorithm achieves a robust behavior by combining the numerical properties of other advanced optimizers, such as RMSprop and AdaGrad [50]. The equations of the Adam algorithm are the following [50]:

$$\boldsymbol{\beta}_n = \boldsymbol{\beta}_{n-1} - \frac{\alpha}{\sqrt{\mathbf{v}_n} + \epsilon}\mathbf{m}_n, \tag{2.134}$$

$$\mathbf{m}_n = \frac{c_1 \mathbf{m}_{n-1} + (1 - c_1)\,\nabla_\beta \mathcal{J}_n}{1 - c_1^n}, \tag{2.135}$$

$$\mathbf{v}_n = \frac{c_2 \mathbf{v}_{n-1} + (1 - c_2)\left(\nabla_\beta \mathcal{J}_n\right)^2}{1 - c_2^n}. \tag{2.136}$$

In eqs. (2.135-2.136), it can be noted that the main difference between the Adam optimizer and the former Gradient-Descent algorithm with Added Momentum is given by the scaling term for the learning rate ($\sqrt{\mathbf{v}_n} + \epsilon$). This term is based on the squared-gradient inertial term $\mathbf{v}_n$, which helps the system to remember which parameters have been changing the most. As a result, the learning process is accelerated for those β terms which have been subject to slow changes, whereas any parameters subject to drastic changes are stabilized. Thanks to this behavior, the Adam algorithm manages to achieve smooth changes for every parameter considered during the optimization process. The influence of the learning rate α in the convergence rates is also mitigated due to the presence of this new scaling term. The default values recommended for the Adam optimizer are $\alpha = 0.001$, $c_1 = 0.9$, $c_2 = 0.999$, and $\epsilon = 10^{-8}$ [50]. The previous values can be subject to changes depending on the application however. For example, in computer vision systems, the default value of ϵ can be as high as 1.0 [42]. The learning rate (α) can also be tuned for specific applications, although the default values provided have been reported to work well for a wide variety of engineering applications.

Bold Drive Method

The Bold Drive Method [51] corresponds to an extension of the Vanilla Gradient-Descent algorithm, which continuously adapts the learning rates (α) being employed. The overall routine employed by this algorithm is the following:

$$\boldsymbol{\beta}^* = \boldsymbol{\beta}_{n-1} - \alpha_{n-1} \left(\nabla_\beta \mathcal{J}\right)_{n-1}, \tag{2.137}$$

$$f_1 = \begin{cases} 1 & \text{if } \mathcal{J}\left(\boldsymbol{\beta}^*\right) < \mathcal{J}\left(\boldsymbol{\beta}_{n-1}\right) \\ 0 & \text{otherwise} \end{cases}, \tag{2.138}$$

$$\boldsymbol{\beta}_n = f_1 \; \boldsymbol{\beta}^* + (1 - f_1) \; \boldsymbol{\beta}_{n-1}, \tag{2.139}$$

$$\frac{\alpha_n}{\alpha_{n-1}} = f_1 \; k^+ + (1 - f_1) \; k^-. \tag{2.140}$$

In eqs. (2.137-2.140), it can be noted that the algorithm first calculates a trial update $\boldsymbol{\beta}^*$. If this update yields positive results, the logical gate f_1 is set to 1, such that the results of the previous iteration can be overwritten. Otherwise, the results of the previous iteration are kept ($f_1 = 0$). Likewise, the constants $k^+ > 1$ and $k^- < 1$ indicate whether the learning rate should be increased or not based on the logical gate f_1. Common values for the learning rate updates are $k^+ = 1.2$ and $k^- = 0.5$. Thanks to this simple approach, it is possible to obtain an algorithm which always marches forward at an optimum learning rate. The behavior of the scheme when $f_1 = 0$ resembles a line-search algorithm, since it seeks for the optimal learning rate which produces convergence. A suitable stopping criterion for this algorithm is $\alpha < 10^{-12}$, since the learning rate quickly vanishes once a local minimum has been found. Advanced versions of this scheme can also be created, such that momentum is added to the system or the parameters of the Adam optimizer are tuned in real-time.

Regarding the overall applicability of this algorithm, it was found that the Bold Drive Method was particularly suitable for CFD optimization problems, since finding an appropriate learning rate for such problems can require extensive calibration procedures otherwise. The Bold Drive algorithm could thus operate with absolute independence, and was able to improve the running times by a factor of x10. Furthermore, the Bold Drive Method was perhaps the only algorithm which reasonably guaranteed convergence.

Despite the previous success, it must be noted that the performance of the Bold Drive algorithm was inferior to that of the Adam optimizer in Deep Learning applications. The main issue is that the scaling procedure adopted by the Adam optimizer for the learning rate is unmatched by other algorithms in this context, since Deep Learning systems contain large arrays of subtle parameters. It was further observed that the default hyper-parameters for the Adam optimizer managed to train Deep Learning systems with minimal supervision as well. However, the results obtained in CFD optimization using the Adam optimizer were far less optimistic, since the learning rates found by the Bold Drive Algorithm substantially accelerated the learning process. As a result, both algorithms remain useful in different contexts.

2.3. Machine Learning Applied to CFD

This Section will discuss different emerging techniques in Machine Learning and their potential applications to CFD. A special emphasis will be placed in Deep Neural Networks (DNN), since this approach has been shown to outperform any other alternatives in complex tasks [3]. DNN's have been recently achieved great success in tasks such as Computer Vision, Natural Language Processing (NLP) and data classification [3]. The success of DNN's in different fields is greatly due to their flexibility, which allows these systems to adopt different forms depending on their application. Based on this context, it is possible that the DNN's developed in the future for RANS turbulence modelling will present completely new architectures. However, it will only be possible to assess the possibilities once large turbulence databases are assembled. The emergence of such datasets could also reveal new physical parameters present in turbulent flows. For example, it may become possible to build a working description of Kolmogorov's energy cascade, or a model based on the non-linear eigenmodes of turbulence.

2.3.1. Neural Networks

Neural Networks correspond to one of the leading techniques employed in Machine Learning today. These systems were originally inspired by advances in neuroscience, which postulated that neurons were organized hierarchically [3, 52, 53]. Based on these insights, mathematicians soon developed Artificial Neural Networks, which employed large stacks of non-linear activation functions resembling the action of neurons. The basic structure of Fully-Connected Neural Networks can be found in Figure 2.9. Here, it can be seen that the set of input features ($\mathbf{X}$) is passed through different neurons, whose output is a non-linear function of the previous layer contents. The process is repeated until a set of output units $\mathbf{Y}$ is obtained. The most common types of neurons employed in Machine Learning can be found in Figure 2.10. In this Figure, $z = \mathbf{w} \cdot \mathbf{x} + b$ represents a generalized linear transformation for any input vector $\mathbf{x}$. An important property of the neurons presented in Figure 2.10 is that they have proven to fulfill the Universal Approximation Theorem. This theorem states that, if there exists a function $\mathbf{Y} = f(\mathbf{X})$, then a sufficiently large neural network can approximate any such function [54, 55]. However, no promises are made regarding the size of the Neural Network nor the amount of training data required. Regarding these points, it has further been proven that the total number of neurons required by a problem can be exponentially decreased by stacking multiple hidden layers [56]. The previous concept has inspired the raise of Deep Neural Networks (DNN), which present large hierarchical structures. The performance of these networks can be further enhanced by creating systems which are well-tuned for their respective applications.

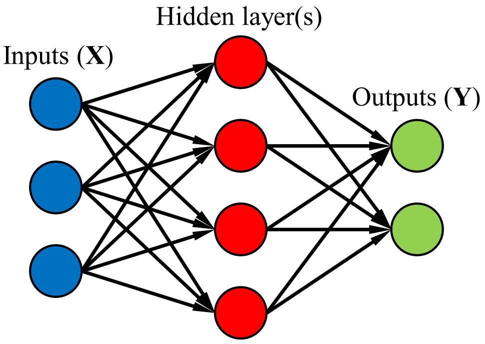


Figure 2.9: Structure of a densely connected Neural Network [3].

Regardless of the specific application, Neural Networks can be trained employing the following generic cost function:

$$\mathcal{J}(\mathbf{X}, \boldsymbol{\beta}) = \sum_{i=0}^{N} \mathcal{L}(Y(\mathbf{X}_i, \boldsymbol{\beta}) - Y_{0,i}). \tag{2.141}$$

In eq. (2.141), $\mathbf{Y}(\mathbf{X}, \boldsymbol{\beta})$ represents the output of the neural network given a stack of input features X and a vector of parameters $\boldsymbol{\beta}$. The labels $Y_{0,i}$ represent the training examples provided to the system, whereas the operator $\mathcal{L}$ corresponds to the loss function employed to evaluate the system's performance. Different loss functions can be employed depending on the nature of the problem. For example, in classification tasks such as anomaly detection, logarithmic loss functions tend to be employed. However, in data fitting tasks, quadratic loss functions are preferred ($\mathcal{L}(x) = x^2$). The main advantage of quadratic functions is that they present linear gradients ($2x$), which facilitates the overall training procedure. Some authors occasionally employ absolute value loss functions ($\mathcal{L}(x) = |x|$), since these functions introduce smaller penalties into the system due to the presence of outliers. However, the gradient of absolute value functions cannot be fully linearized, and thus they can result difficult to train.

Regarding the properties of the neurons presented in Figure 2.10, it must be noted that modern systems tend to employ RELU functions. These functions present sharp non-linearities given by their

On/Off behavior, yet they are fairly easy to train once they fall into place. However, the discontinuous nature of these operators makes them highly sensitive to the initial weights given to the Neural Network, since any neurons initialized to negative values cannot be trained directly. Furthermore, in data fitting problems, the use of RELU operators yields discontinuous curves which can be undesirable. For example in CFD, the velocity gradients ($\nabla \mathbf{u}$) can be as important as the velocity magnitudes ($\mathbf{u}$) in certain calculations. A similar argument can be made regarding other physical properties, such as the pressure (P) or the temperature (T). Given this context, the use of Hyperbolic Tangent neurons may be preferable in fluid mechanics. While these neurons are often considered hard to train, they present several advantages, such as yielding a continuous output and being fully anti-symmetric. Moreover, it can be proven that Hyperbolic Tangent neurons are easier to train than Sigmoid functions, since their zero average makes them less prone to become saturated [57]. The use of Hyperbolic Tangent neurons in fluid mechanics can be found documented in the work of Milanos & Koumoutsakos (2002) [58], which is one of the first studies that applied Neural Networks to CFD.

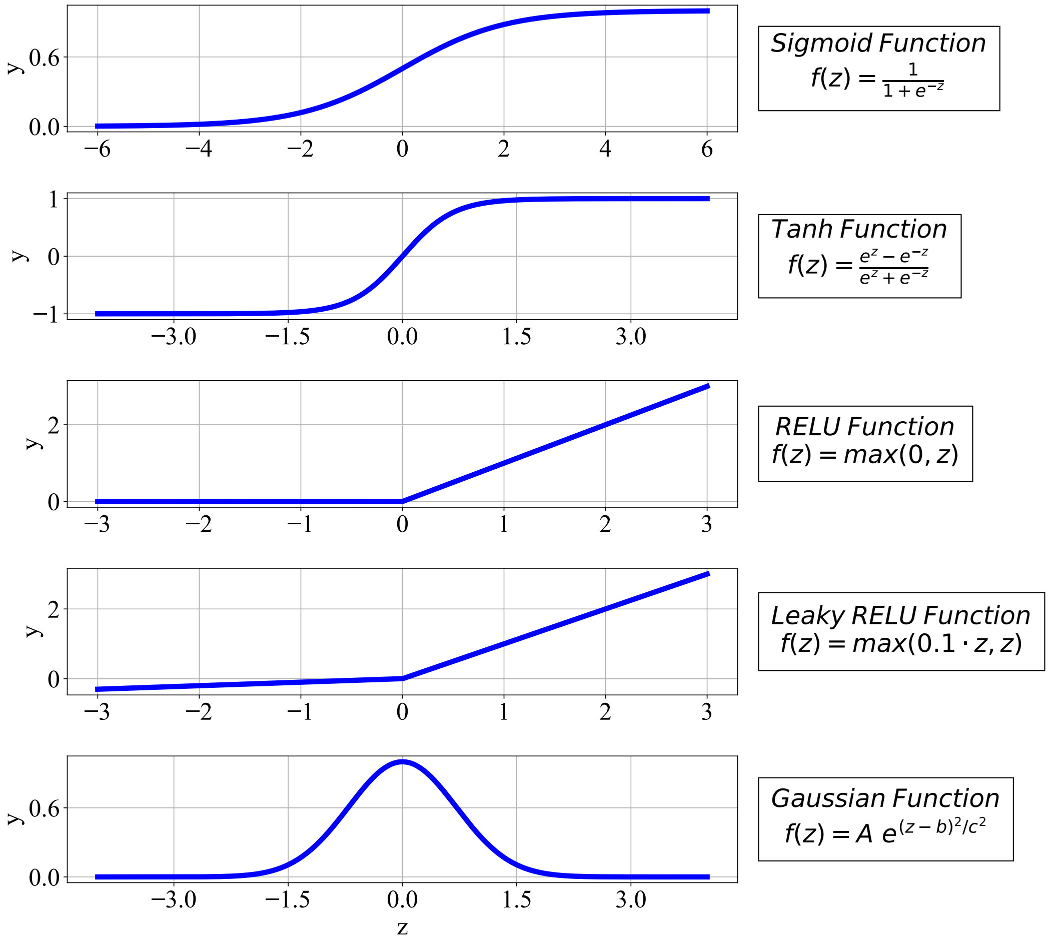


Figure 2.10: Activation functions commonly employed in Neural Networks.

As a final note, it is important to note that one of the most successful applications of Deep Neural Networks in CFD is given by the study performed by Ling et al. (2016) [59]. In this study, a Deep Learning system with embedded Galilean invariance was proposed, which was capable of reconstructing the full Reynolds Stress (anisotropy) tensor $\overline{\rho}\ \overline{\mathbf{u}'\mathbf{u}'^{T}}$ for two test cases regarding the flow across a wavy wall and the flow through a square duct. The Neural Networks were trained employing six different flow conditions, which involved the flow around a square cylinder, an internal duct flow, a channel flow, a perpendicular jet in cross-flow, a converging-diverging channel and an inclined jet in cross-flow. A total of 4000 CPU hours (5.6 months) were required to train the final Machine Learning architecture created by Ling et al. (2016) [59]. Based on the successful predictions established by these authors, it was concluded that Deep Learning systems can be employed to predict turbulence, although larger datasets need to be assembled before building general CFD solvers.

2.3.2. Convolutional Neural Networks

Convolutional Neural Networks (CNN's) are one of the most important architectures developed in Machine Learning. These systems are able to recognize patterns in segments of data, such as images or recorded speech, and further process these results in order to produce decisions. The study of CNN's was greatly inspired by the Nobel prize work of Hubbel and Wiesel in 1962 [53], who studied the visual neural cortex of cats. Hubbel and Wiesel determined that neurons were organized hierarchically, such that simple neurons first detected the presence of small visual features (S-cells), and deeper neurons detected the presence of more complex objects (C-cells) [53]. After years of research, these insights let to the first Convolutional Neural Network proposed in 1998 by Yann LeCun et al. [60]. Their neural network, called LeNet-5, managed to recognize handwritten digits with remarkable accuracy even under the presence of noise and other effects. Today, the CNN holding the world record in hand-written digit recognition reaches 99.79% accuracy based on similar principles [61]. A schematic representation of the structure employed in the original LeNet-5 network can be found depicted in Figure 2.11. Here, it can be seen that the network first filters an image looking for small patterns, such as edges or parts of a digit. The results obtained by each filter are then passed to deeper layers. The operation called sub-sampling in the LeNet-5 network corresponds to a max-pooling operation. Even though more advanced pooling operators have been developed [3], these operators allow CNN's to consider various alternatives for detecting an event, and only retain the strongest detected signals. For example, in vehicle detection systems, the first convolutional filters can try to detect the presence of various types of wheels, such that a max-pooling operator can decide whether a wheel was present in the picture or not. If a wheel appeared in the image, one of the filters will raise a strong signal, and otherwise all the filters will yield zero. The previous process of convolutions and sub-sampling is repeated until the last layers of the CNN are reached. Here, fully-connected neurons are employed in order to detect high level concepts, such as the true presence of a hand-written digit. Despite being an early prototype, modern CNN's have remained significantly close to the concepts introduced in the LeNet-5 network. Several modelling advances have been introduced to improve the efficiency of these systems however. One of the most important inventions has been the use of Dropout regularization [62], where a random subset of the connections present in a CNN is suppressed each iteration, such that the system does not become over-reliant on small details. Dropout regularization is considered as one of the strongest techniques developed in Machine Learning, since it allows the user to introduce a strong form of regularization without further tuning hyper-parameters. Another important innovation in CNN's includes Szegedy's Inception modules [63], which allow CNN's to consider various convolutional filters and a sub-sampling operator at the same time. The optimizer is thus responsible for selecting the most appropriate operations between these alternatives, and a suitable architecture for a problem can be developed automatically.

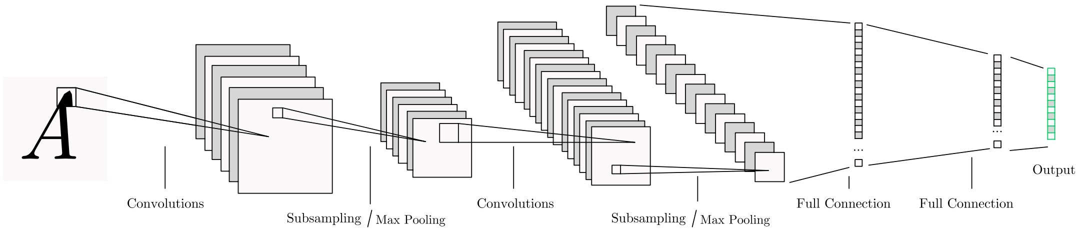
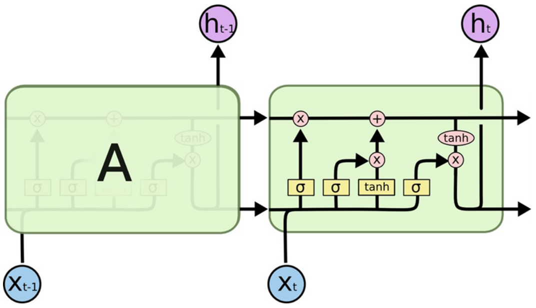


Figure 2.11: Schematic representation of a traditional Convolutional Neural Network. Source: Wesley (2017) [64].

The application of CNN's to CFD has yielded several interesting results. For example, in the work of Tompson et al. (2017) [65], Convolutional Neural Networks were employed to simulate raising smoke plumes. Zhang et al. (2018) [66] alternatively employed CNN's to predict the lift coefficient of airfoils. More applications of CNN's in CFD will appear in the future, since these systems have an inherent ability to detect macro patterns and apply corrections. However, it is important to remember that the predictive capabilities of CNN's should be harnessed within appropriate physical frameworks, such that their work can be simplified. These systems should never be employed to replace physical constraints which could be easily hard-coded [65]. As a rule of thumb, every additional prediction expected from a Deep Learning system increases the amount of training data required substantially.

2.3.3. Long Short Term Memory Neural Networks

Long Short Term Memory (LSTM) neural networks correspond to one of the most abstract systems employed in Artificial Intelligence today. These networks employ an architecture closely resembling an electronic circuit, as it can be seen in Figure 2.12. Despite the complexity of these systems, their use has massified due to their impressive ability to perform complex tasks, such as performing accurate language translations. One of the key features in LSTM networks is their ability to decide whether any given inputs should be transformed, remembered or forgotten. Thanks to these long-term dependencies, LSTM's can manage to navigate through sentences, character by character, and perform accurate translations. Language translation formally corresponds to a branch of Natural Language Processing (NLP), which attempts to build AI systems capable of processing human language. One important discovery made in this field is that there exist simpler units which can achieve a similar performance to LSTM units. These units are called Gate Recurrent Units (GRU). Their overall architecture can be found depicted in Figure 2.13. Here, it can be noted that GRU's present far less parameters than LSTM units indeed. This in turn implies that larger systems can be built, such that the overall system performance is increased for the same total number of parameters [67]. Another important advance made in NLP is the use of bi-directional LSTM networks, which allow computers to process sentences from start to finish, and vice-versa. This has proven useful while performing translations between languages which invert the sentence order, such as English vs. Spanish. Beyond their success in language translation tasks, LSTM networks present a remarkable ability to learn complex data patterns. Even simple LSTM networks have managed to generate Shakespearean sonnets [3].



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

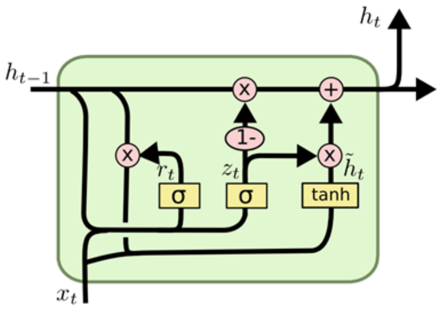
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 2.12: Schematic representation of a Long Short Term Memory (LSTM) neuron. Source: Olah (2015) [68].



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2.13: Schematic representation of a Gate Recurrent Unit (GRU) neuron. Source: Olah (2015) [68].

The application of LSTM networks in CFD could yield important results, since their ability to capture complex data patterns could be used to predict turbulence parameters based on raw sets of global features. However, training LSTM networks for CFD applications can be complex, since every DNS simulation corresponds to only one data sequence, and thus an exceptionally large database is required. A detailed assessment of the numerical properties of LSTM networks in CFD can be found in the works of White (2017) [69] and Mohan et al. (2018) [70]. The results obtained by these authors indicate that further research is needed in order to design LSTM architectures that are well-suited for CFD modelling. A research project applying LTSM networks to the prediction of μ_t in the RANS equations appears to be missing at the moment.

2.3.4. Logarithmic Neural Networks

While studying the application of Neural Networks in CFD, it was detected that standard Deep Learning systems lacked the ability to discover appropriate dimensionless numbers contained in their input data. For example, in heat transfer problems, a Neural Network should be able to select the best dimensionless groups in order to build a correlation (e.g. $Re^a Pr^b$). A similar argument could be made regarding the existence of the Semi-Locally Scaled Reynolds Numbers (Re_τ^*) in variable-property flows, or other dimensionless groups present in fluid mechanics. While the previous problems may be easy to solve numerically, discovering appropriate features to use in complex systems can be far more challenging. The issue of selecting appropriate features is not only limited to providing the required information, but also to proving that the features employed facilitated the work of the Neural Network.

One simple solution to the previous problem is to employ logarithmic neurons, as they were proposed by Hines (1996) [20]. These neurons are able to build correlations with the following form:

$$Y = \prod_{i=1}^{N} \mathbf{X}_i^{w_i}. \tag{2.142}$$

In eq. (2.142), the vectors $\mathbf{X}_i$ correspond to each of the input features given to the neuron, whereas the weights w_i correspond to the exponents assigned by the Neural Network. The relation between these neurons and fluid mechanics is evident, since they can form groups such as: $Re^a Pr^b$, $Re_\tau^* = Re_\tau \sqrt{\rho/\rho_w}\,(\mu_w/\mu)$, etc. While the optimization problem given in eq. (2.142) can be approached directly, the following architecture is preferred for logarithmic neurons:

$$Y = e^{\mathbf{w}\cdot ln(\mathbf{X})} = e^{\mathbf{w}\cdot\mathbf{X}'}. \tag{2.143}$$

The main advantage of eq. (2.143) is that the term $e^{\mathbf{w}\cdot\mathbf{X}'}$ corresponds to a simple neuron with an exponential activation function, which is therefore easier to implement using existing Machine Learning libraries. Furthermore, the original stack of input features ($\mathbf{X}$) can be pre-transformed into log-space ($\mathbf{X}'$) before starting the Machine Learning process. The log-decomposition procedure can be further employed to realize that any dimensionless group in fluid mechanics (Y) can be written as:

$$ln(Y) = a \cdot ln(U) + b \cdot ln(\mu) + c \cdot ln(\rho) + d \cdot ln(L) + e \cdot ln(c_p)... \tag{2.144}$$

Eq. (2.144) implies that a Neural Network may be able to discover any dimensionless group in fluid mechanics based on experimental measurements. An example of this will be presented in Chapter 3. However, it is always preferable to feed dimensionless features into the Neural Network, since this guarantees that the results will also be dimensionless.

From a mathematical perspective, it can be proven that the feature selection procedure created by logarithmic neurons corresponds to an instance of non-linear PCA (Principal Component Analysis). The proof can be derived from the properties of the linear neurons present in eq. (2.143), which operate in log-space. Mathematicians have proven that adding an initial layer of linear neurons to a DNN is equivalent to performing linear PCA, i.e., to choosing only the ”N” most relevant vector directions [71]. Therefore, logarithmic neurons inherently tend to choose the best ”N” vector components in log-space, which are then transformed into dimensionless groups.

In conclusion, the use of logarithmic neurons can help users to determine which are the best features to employ in a Neural Network, and thereby reduce the development times significantly. Other alternative approaches for the discovery of data-driven closure models were recently explored by Pan & Duraisamy (2018) [72]. However, their results indicate that further research is required.

2.3.5. Model Selection

In Machine Learning, several model selection techniques have been developed in order to prevent a problem known as over-fitting, which was introduced back in Figure 2.3. The development of general methods has been fundamental in fields such as computer vision or machine translation, where the Neural Networks employed are fairly abstract. The different methods developed will be presented below:

- **Holdout Method:** In the presence of big data, the preferred model selection technique in Machine Learning is to split the global dataset into three sub-sets, which are called Training set, Cross-Validation (CV) set and Test set. Under this approach, the Training set is employed to calibrate the parameters of a Neural Network, whereas the CV set is employed either to evaluate the system's performance or to tune any model hyper-parameters. The data of the CV set should never be used to train the model directly, since it is only intended to provide a general benchmark of the model's response to untrained samples. Finally, the Test set is intended to provide a final evaluation of the model's performance in unseen data. The Deep Learning system should never be changed according to the Test set, since it would then become part of the CV set. The data split proportions employed for this approach can vary depending on the amount of data available. For moderate datasets, a data distribution of 80-10-10 can be employed for the Training set, Cross-Validation set and Test set respectively. However, more aggressive data split proportions can be chosen in large databases, such as 98-1-1.

  Regarding the Holdout approach, it is important to note that other conclusions can also be obtained regarding the problems studied. For example, if the training error is too large, it can be concluded that the DNN employed is not capable of representing the problem well. This problem is formally known as high bias [3]. Conversely, when a problem presents a low training error but a large CV error, the model is said to present high variance (or over-fitting). As a rule of thumb, over-fitting is usually associated to DNN's containing a large number of parameters, which nearly managed to interpolate the training data. Therefore, the risk of over-fitting can be usually mitigated by choosing the smallest model which manages to obtain a low training error. The lower the number of parameters present in the model, the lower also tends to be the risk of over-fitting. However, the previous intuition must always be validated employing a formal Test set.

- **Elbow Method:** The Elbow Method corresponds to another technique employed in Machine Learning, which allows the user to draw conclusions from either the Training set or the Cross-Validation set. The intuition behind this method can be found depicted in Figure 2.14, which presents the training error of a model using different polynomial approximations. Here, it can be seen that the overall error for the model decreased dramatically once a polynomial of 5^{th} order was chosen, which corresponds to the true order of the model. Higher order polynomials present similar training errors under this context, since there exists little margin for improvement. As it turns out, the trends observed in Figure 2.14 are also representative of similar problems encountered in Deep Neural Networks and CFD optimization [4]. Given this context, the Elbow Method states that the optimal size of a model is defined by the location of the sharp break in its error curve as it is presented in Figure 2.14. Thanks to this method, it is possible to estimate the appropriate size of a model even in highly abstract scenarios. However, not every problem presents a sharp break in its error curve as expected by the Elbow method. In such cases, other model selection techniques must be employed.

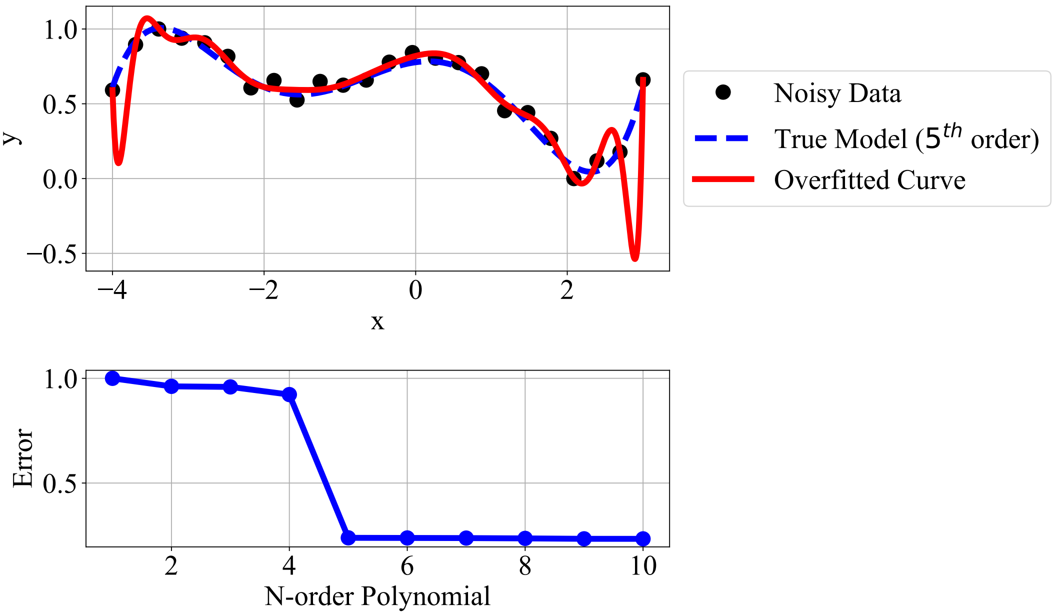


Figure 2.14: Example of the Elbow Method applied to a polynomial fitting system. True model: 5^{th} order polynomial.

- **K-fold Validation:** The K-fold validation method corresponds to an extension of the original Holdout Method, where the database is split into K different sub-sets for validation. For each of these K trials, the Machine Learning model must be trained and validated. The use of a CV set may become optional, depending on whether the model contains hyper-parameters or not. The K-fold validation method can be used to estimate the sensitivity of the model's parameters, since the results obtained using different datasets can be assessed. Furthermore, this method can also allow the user to examine the influence of arbitrary data samples in the model. However, it is often preferred to employ random data splits. While the K-fold validation method is computationally expensive, it must be noted that this method corresponds to one of the most rigorous approaches defined in Machine Learning, and that it provides the greatest insights regarding the model's performance. Traditional implementations of the K-fold validation procedure tend to choose K=10.

The methods previously described correspond to the standard techniques employed in Machine Learning for model selection. One case which is not completely covered by the previous methods however is the selection of models for small databases. For these cases, it is hard to reach consensus regarding the best alternatives available. One possible approach is to choose a reasonable model based on previous experience, and then to train the model parameters using a scientific approach, such as the K-fold validation method. However, the conclusions drawn from a small database can never be conclusive.

2.3.6. Other Machine Learning Techniques

While the previous Sections focused on discussing different aspects of Deep Neural Networks, it is important to note that there exist other approaches in Machine Learning which are actively employed. A brief description of the most relevant techniques will be given below:

- **DNN's with Multi-valued Neurons and Frequency Domain Features:** Multilayer Neural Networks with Multi-valued Neurons and Frequency Domain Features (MLMVN-FDF) correspond to an emerging technique in Machine Learning, which employs an alternative formulation to Convolutional Neural Networks. Instead of processing images as arrays of pixels, MLMVN-FDF systems perform a Fourier transformation, where every image is decomposed into a set of frequencies with a given phase and magnitude. The mathematical formulation of Neural Networks is thus adapted to work with complex numbers, since this is the most natural representation of Fourier spaces. In a recent article published at the time of writing this thesis, Aizenberg & Gonzalez [61] showed that a MLMVN-FDF system could achieve a 100% recognition rate (human-level performance) during the classification of over 10 000 handwritten digit images contained in the Test Set of the MNIST database. Due to this reason, MLMVN-FDF systems are likely to play an important role in the AI systems developed in the future. The far-reaching consequences of this breakthrough will certainly constitute an active area of research.

- **Decision Trees:** Decision Trees correspond to one of the most employed alternatives to Neural Networks. These systems train hard classifiers based on their stacks of input features. Unlike Deep Neural Networks, which build complex non-linear relations between the input features, Decision Trees only consider classification processes given by If-Else conditions. Despite their simplicity, Decision Trees can classify information with reasonable accuracy. However, the simple classification process adopted by these algorithms lowers their performance compared to Neural Networks in complex tasks. Example applications of Decision Trees in CFD can be found in the works of Kaandorp (2018) [73] or Ladický et al. (2015) [74].

- **Support Vector Machines:** Support Vector Machines (SVM) correspond to advanced algorithms which try to establish the widest classification margins possible between different label groups, as it can be seen in Figure 2.15. These algorithms can also be modified to operate in non-linear spaces by employing Kernel functions, which are used to define the problem. Google's Tensorflow library for example incorporates built-in Gaussian Kernels among other options.

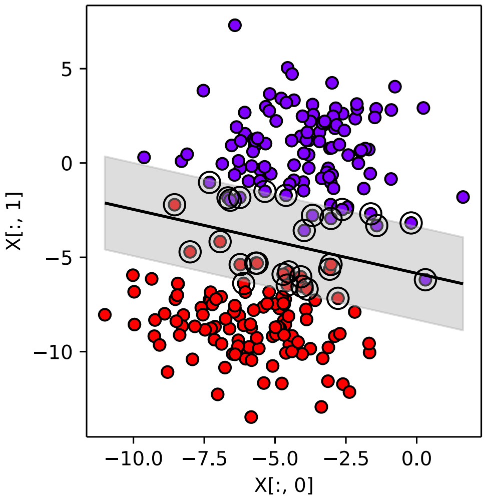


Figure 2.15: Example of the results obtained using a Support Vector Machine (SVM) algorithm. Source: Baker (2018) [75].

- **K-means clustering:** The K-means algorithm corresponds to one of the simplest algorithms defined in Machine Learning. In this method, K-centers are randomly defined within the input feature space. The positions of the K-clusters are updated based on the average position of the samples which were assigned to each cluster according to their minimum distance. The overall K-means clustering procedure can be repeated for N initialization trials in order to guarantee convergence. During a formal study, an appropriate number of clusters can be defined using the Elbow Method. Despite its simplicity, the K-means method has allowed researchers to classify different types of feature groups present in a problem, as it can be seen in Figure 2.16. If the samples are close enough in the feature space, the original samples can even be replaced by one of the K-mean cases. This technique can also reveal correlations between the studied features, and provide an indication of how far is a new test case from the original training samples. Due to these reasons, the K-means clustering procedure is potentially useful in CFD.

  Finally, it must be noted that the K-means clustering algorithm corresponds to one of the few successful instances of unsupervised learning, where an algorithm is able to draw conclusions regarding the input data without employing explicit labels (Y). However, these conclusions are only limited to determining the types of samples present in the problem.

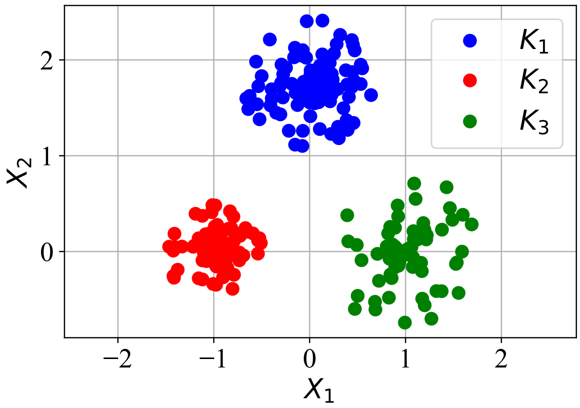


Figure 2.16: Example of the K-means clustering algorithm in a 2-D feature space (X_1,X_2).

2.3.7. Field Inversion Machine Learning

The Field Inversion Machine Learning (FIML) scheme proposed by Parish & Duraisamy (2016) [5] for CFD constitutes one of the leading approaches employed in data-driven RANS turbulence modelling. The general scheme employed in this approach can be found in Figure 2.17. Here, it can be seen that the solution procedure is split in two stages. The first stage corresponds to the Field Inversion procedure, where an optimal set of ($\boldsymbol{\beta}$) corrections for a RANS turbulence model is obtained. These corrections are then used to train a Machine Learning model, such that a new predictive system can be built. The original problem studied by Parish & Duraisamy (2016) [5] consisted in building corrections for the production of turbulent kinetic energy in the original $k - \omega$ model, as it will be seen later in Chapter 3. Their Field Inversion procedure was based on the following cost function:

$$\mathcal{J} = \sum_{i=1}^{N} \left(u - u^*\right)^2 . \tag{2.145}$$

In eq. (2.145), u corresponds to the CFD velocities obtained using the $k-\omega$ turbulence model, whereas u^* corresponds to the DNS velocity profiles. Despite the simplicity of the cost function employed by Parish & Duraisamy (2016) [5], it is interesting to note that their $\boldsymbol{\beta}$ corrections managed to reach a clear local minimum, since most optimization problems without a regularization term grow indefinitely. The original optimization procedure implemented by these authors was based on Bayesian inference and required the computation of the Hessian matrix. However, it was discovered during the present work that a simple Jacobian optimizer could be used to replicate the final $\boldsymbol{\beta}$ corrections obtained by these authors, and that the additional complexity introduced by probabilistic optimization approaches could be completely avoided. As a result, only Jacobian optimization algorithms were employed to perform Field Inversion during the present thesis. Several additional improvements to the original Field Inversion procedure will be presented in Chapters 3 and 4.

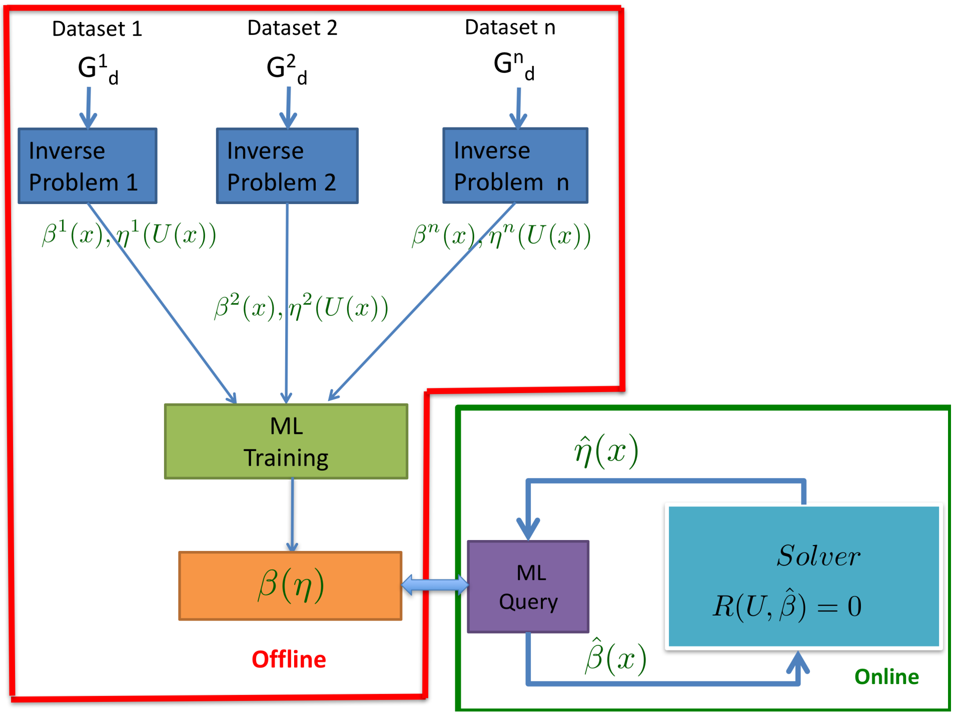


Figure 2.17: Field Inversion Machine Learning (FIML) procedure proposed by Parish & Duraisamy (2016) [4]. Image extracted from the original publication.

Regarding the overall applicability of the FIML methodology developed by Parish & Duraisamy (2016) [5] , it is important to note that Field Inversion optimization can be used to infer large sets of corrections based on scarce information. For example, Singh et al. (2017) [76] showed that simple measurements for the lift coefficient of an airfoil could be used to build 2-D corrections for a RANS turbulence model. This is a clear case of high-dimensional inference, which is made possible by the fact that the optimal parameter distribution ($\boldsymbol{\beta}_{\mathbf{opt}}$) presents a zero-gradient for every variable once it converges to a local minimum ($\partial\mathcal{J}/\partial\beta_i = 0$). As a result, an independent equation can be built for every DOF (Degree-of-Freedom) present in the system, regardless of the original formulation of the cost function. A detailed analysis of the Field Inversion problems created by Parish et al. (2016) [5] can be found in Sections 3.1 and 3.3 from Chapter 3.

3

Preliminary Examples

In order to gain experience building Machine Learning models for CFD, four preliminary examples were considered. The first exercise consisted in applying the principles of Field Inversion and the Discrete Adjoint Method to an introductory problem proposed by Parish & Duraisamy (2016) [5], which consisted in reconstructing a 1-D radiative heat transfer profile. This problem was followed by a second exercise, where the Field Inversion methodology was applied to a laminar CFD solver with the objective of recovering a dynamic viscosity profile using velocity measurements. The last problem studied regarding Field Inversion consisted in replicating the results obtained by Parish & Duraisamy (2016) [5] during the optimization of the $k-\omega$ turbulence model in a channel flow. This problem held great importance, since it constituted the basis of the analysis presented later in Chapter 4 for variable-property flows. Finally, in order to gain experience building Deep Neural Networks, it was decided to create a Machine Learning system capable of predicting the Drag coefficient of a cylinder and inferring the existence of the Reynolds number. Despite the small nature of this problem, it was possible to prove that logarithmic neurons can quickly filter information and converge to appropriate solutions. These insights were later used to automate the feature selection processes during Chapters 4 and 5. As a result, the preliminary examples studied in this Chapter correspond to an integral part of the analysis developed, where robust Machine Learning methods were identified for the present thesis work. A complete version of the Python code employed to solve every exercise presented in this Chapter can be found in Appendix A.

3.1. Field Inversion for a 1-D Radiative Heat Transfer Problem

The first Field Inversion problem considered during the present thesis consisted in replicating the results obtained by Parish & Duraisamy (2016) [5] during the reconstruction of a 1-D radiative heat transfer profile. The true unknown model considered by these authors was based on the following equation:

$$\frac{\partial^2 T}{\partial z^2} = \epsilon(T)(T^4 - T_\infty^4) + h(T - T_\infty). \tag{3.1}$$

In eq. (3.1), $\epsilon(T)$ corresponds to the emissivity coefficient for the model, which was given by the following non-linear relation:

$$\epsilon(T) = \left(1 + 5\ sin\left(\frac{3\pi}{200}T\right) + e^{0.02T}\right) \cdot 10^{-4}. \tag{3.2}$$

It can be seen that eq. (3.2) corresponds to a fairly complex expression, whose results must be inferred by the optimizer. Back in eq. (3.1), T_∞ corresponded to the environment temperature, which was assumed to be $T_\infty = 50$ during the nominal case considered by Parish et al. (2016) [5]. The variable h corresponded to a constant convective heat transfer coefficient with a value of 0.5. Homogeneous Dirichlet boundary conditions were also considered for the system at $z = \{0, 1\}$. Based on this ground-truth model, Parish & Duraisamy generated temperature profile measurements (T^*), which were later given to the optimizer. However, the original model presented in eqs. (3.1-3.2) was otherwise ”forgotten”, and only the following approximate model was assumed to be known:

$$\frac{\partial^2 T}{\partial z^2} = \epsilon_0 (T^4 - T_\infty^4). \tag{3.3}$$

In eq. (3.3), the emissivity ϵ_0 now corresponds to a constant with a value of $5 \cdot 10^{-4}$. The optimization problem proposed by Parish & Duraisamy (2016) [5] thus consisted in building an array of $\beta(z)$ modulations for ϵ_0, such that the true temperature profiles (T^*) could be recovered:

$$\frac{\partial^2 T}{\partial z^2} = \beta(z)\ \epsilon_0\ (T^4 - T_\infty^4). \tag{3.4}$$

By comparing eqs. (3.1) and (3.4), it can be noted that the ground-truth labels for $\beta(z)$ are the following:

$$\beta^*(z) = \frac{1}{\epsilon_0}\left(1 + 5sin\left(\frac{3\pi}{200}T\right) + e^{0.02T}\right) \cdot 10^{-4} + \frac{h}{\epsilon_0}\frac{T - T_\infty}{T^4 - T_\infty^4}. \tag{3.5}$$

The values given by eq. (3.5) were only kept for validation purposes, and they were not employed during the Field Inversion process. While the previous optimization problem appears to be well-posed, it will be proven that finding $\beta(z)$ corrections is orders of magnitudes slower than optimizing the following problem:

$$\frac{\partial^2 T}{\partial z^2} = \epsilon_0\ (T^4 - T_\infty^4) + \delta(z). \tag{3.6}$$

In eq. (3.6), the problem now consists in finding additive corrections $\delta(z)$ for the model, which represent potential missing terms in the equation. The ground-truth labels for this alternative problem are the following:

$$\delta^*(z) = \left[\left(1 + 5sin\left(\frac{3\pi}{200}T\right) + e^{0.02T}\right) - \epsilon_0\right]\left(T^4 - T_\infty^4\right) + h\,(T - T_\infty)\,. \tag{3.7}$$

In eq. (3.7), it can be noted that the expression given for $\delta^*(z)$ does not contain a denominator for the convective term, as it occurred back in eq. (3.5). Beyond these considerations, the main advantage of this alternative formulation is given by the simplification of the matrices present in the Discrete Adjoint Method. The introduction of additive corrections $\delta(z)$ implies that the matrix $\partial\mathcal{R}/\partial\mathbf{W}$ is decoupled from the Field Inversion corrections, and that $\partial\mathcal{R}/\partial\boldsymbol{\delta} = 1$, which is a substantial simplification. While an optimizer seeking $\beta(z)$ corrections could fail to converge after 12+ million iterations, a similar optimizer searching for $\delta(z)$ corrections could converge to floating point precision in less than 6500 iterations. Both versions of the code are presented in Appendix A.1. For future reference, the conversion between $\beta(z)$ and $\delta(z)$ corrections is given by the following formula:

$$\beta(z) = 1 + \frac{\delta(z)}{\epsilon_0\left(T^4 - T_\infty^4\right)}. \tag{3.8}$$

The previous formula can also be used to prove that building $\boldsymbol{\beta}$ multipliers is equivalent to building $\boldsymbol{\delta}$ corrections, although the convergence properties of both schemes differ. In order to start the Field Inversion process, Parish & Duraisamy (2016) [5] considered a cost function based on the squared residual losses for the temperature profiles:

$$\mathcal{J} = \sum_{i=1}^{N}\left(T_i - T_i^*\right)^2. \tag{3.9}$$

The Discrete Adjoint Method presented in Section 2.2.3 was then applied to this problem, since it constitutes a case of constrained optimization. By analyzing the equations presented, it can be noted that the following expressions can be built:

$$\mathcal{R} = \epsilon_0\ (\mathbf{T}^4 - T_\infty^4) + \boldsymbol{\delta} - \nabla^2 T\ \ (= 0)\,, \tag{3.10}$$

$$\frac{\partial\mathcal{R}}{\partial\mathbf{W}} = \frac{\partial\mathcal{R}}{\partial\mathbf{T}} = 4\epsilon_0\mathbf{T}^3 - \nabla^2(), \tag{3.11}$$

$$\frac{\partial \mathcal{R}}{\partial \boldsymbol{\delta}} = \mathbf{1}, \tag{3.12}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}} = \frac{\partial \mathcal{J}}{\partial \mathbf{T}} = 2\left(\mathbf{T} - T_{\infty}\right), \tag{3.13}$$

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{\beta}} = 0. \tag{3.14}$$

The expressions given by eqs. (3.10-3.14) define all the required components to build the Jacobian matrix of the system employing the Discrete Adjoint Method. As it was mentioned in Chapter 2, the systems of equations considered can be discretized either at the start or the end of the derivation process using the Finite Difference Method [22]. The results will be identical, as long as the interpolation schemes employed are consistent. Regarding this context, the stand-alone Laplacian operator found in eq. (3.11) is intended to be replaced by a finite difference matrix $G^{(2)}$ such that $\nabla^2 T = G^{(2)} \cdot \mathbf{T}$. The presence of such operator can be derived from first principles, although it may be easier to understand this term by analyzing the discretized version of the residual operator ($\widetilde{\mathcal{R}}$):

$$\widetilde{\mathcal{R}} = \boldsymbol{\beta}\, \epsilon_0\, \left(\mathbf{T}^4 - T_{\infty}^4\right) - G^{(2)} \cdot \mathbf{T}. \tag{3.15}$$

For a uniform grid, the matrix $G^{(2)}$ can be approximated using second-order accurate Taylor expansions according to the following scheme [22]:

$$G^{(2)} = \frac{1}{\Delta z^2} \begin{bmatrix} 1 & -2 & 1 & 0 & \dots & & & & \\ 0 & 1 & -2 & 1 & \dots & & & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ & & & & \dots & 1 & -2 & 1 & 0 \\ & & & & \dots & 0 & 1 & -2 & 1 \end{bmatrix}. \tag{3.16}$$

Based on the previous equations, it can be clearly seen that differentiating $\widetilde{\mathcal{R}}$ with respect to $\mathbf{T}$ will yield $G^{(2)}$ as a stand-alone entity, since it corresponds to a linear operator. In order to obtain a fast and reliable optimizer for the problems considered during the present thesis, the Bold Drive Method presented in Section 2.2.5 was modified in order to incorporate momentum inertia. Thanks to this modification, the optimization times were reduced by factor of 3 even in complex scenarios. The optimizer synthesized was given by the following equations:

$$\boldsymbol{\delta}^* = \boldsymbol{\delta}_{n-1} - \alpha_{n-1}\, \mathbf{m}^*, \tag{3.17}$$

$$\mathbf{m}^* = \frac{c_1 \mathbf{m}_{n-1} + \left(1 - c_1\right) \nabla_{\delta} \mathcal{J}_n}{1 - c_1^n}, \tag{3.18}$$

$$f_1 = \begin{cases} 1 & \text{if } \mathcal{J}\left(\boldsymbol{\delta}^*\right) < \mathcal{J}\left(\boldsymbol{\delta}_{n-1}\right) \\ 0 & \text{otherwise} \end{cases}, \tag{3.19}$$

$$\begin{bmatrix} \boldsymbol{\delta}_n \\ \mathbf{m}_n \\ \alpha_n / \alpha_{n-1} \end{bmatrix} = f_1 \begin{bmatrix} \boldsymbol{\delta}^* \\ \mathbf{m}^* \\ k^+ \end{bmatrix} + \left(1 - f_1\right) \begin{bmatrix} \boldsymbol{\delta}_{n-1} \\ \nabla_{\delta} \mathcal{J}_n \\ k^- \end{bmatrix}. \tag{3.20}$$

In eq. (3.20), it is important to note that m_n must be forced to match the local gradients once divergence is detected, since the addition of momentum to the system can create occasional issues. The final optimization results obtained during this exercise can be found in Figure 3.1. Here, it can be seen that the Field Inversion procedure successfully managed to reconstruct the ground-truth $\boldsymbol{\beta}^*$ profiles, as well as the measured temperature profiles ($\mathbf{T}^*$). The conversion from $\boldsymbol{\delta}(\mathbf{z})$ to $\boldsymbol{\beta}(\mathbf{z})$ corrections was performed after finishing the Field Inversion process using eq. (3.8). It is important to note that in this problem, the introduction of a regularization hyper-parameter (λ) was not necessary, because there existed a finite solution for the $\boldsymbol{\beta}$ corrections. However, the $\boldsymbol{\beta}$ corrections obtained for many problems in nature can grow to infinity due to the presence of noise or a fundamental incompatibility with the experimental results. For example, the Re_{τ}^* scaling theory for variable-property flows may explain

95%+ of the observed measurements, yet there will always exist a small degree of incompatibility between any modelling assumptions and reality. Before concluding this Section, it must be noted that the original problem considered by Parish & Duraisamy (2016) [5] contained a noise term in the "true" emissivity coefficients $\epsilon(T)$. However, such term was omitted in order to create a straight-forward introductory exercise. The optimization algorithm given by eqs. (3.17-3.20) will be employed throughout the remainder of the present thesis. A complete version of the code employed during the present Section can be found in Appendix A.1.

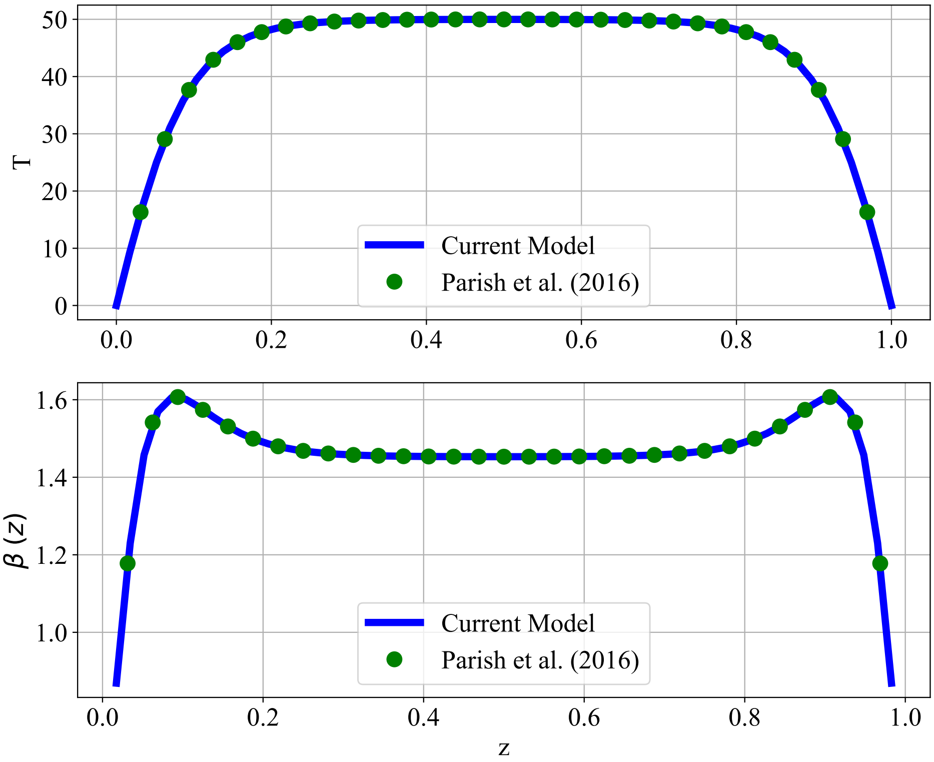


Figure 3.1: Field Inversion optimization results for the introductory problem proposed by Parish & Duraisamy (2016) [5] regarding the reconstruction of a 1-D Radiative Heat Transfer profile.

3.2. Recovery of a Laminar Viscosity Profile

The second Field Inversion problem studied consisted in re-building a dynamic viscosity profile employing existing velocity measurements. While trivial solutions to this problem exist, the main idea was to gain experience applying the Discrete Adjoint Method in a CFD context. The optimization problem was based on the following 1-D equation for a fully-developed channel flow:

$$\frac{\partial}{\partial y}\left(\mu(y)\frac{\partial u}{\partial y}\right) = \frac{\partial P}{\partial x}. \tag{3.21}$$

In eq. (3.21), $\partial P/\partial x$ corresponds to the pressure gradient found in the channel, which is constant for a fully-developed channel flow. The variable $\mu(y)$ corresponds to the dynamic viscosity profile, which is the immediate optimization target. The wall boundary conditions for this problem are given by $u_w = 0$, as it can be expected. The total length for the channel was assumed to be one in order to simplify the numerical discretization. The cost function considered for this problem was based on the squared velocity residuals:

$$\mathcal{J} = \sum_{i=1}^{N}\left(u_i - u_i^*\right)^2. \tag{3.22}$$

Based on eqs. (3.21) and (3.22), the matrices required to drive the Discrete Adjoint Method can be inferred:

$$\mathcal{R} = \boldsymbol{\mu}\nabla^2 u + \nabla\mu \cdot \nabla u - \nabla P, \tag{3.23}$$

$$\frac{\partial \mathcal{R}}{\partial \mathbf{u}} = \boldsymbol{\mu}\nabla^2() + \nabla\mu \cdot \nabla(), \tag{3.24}$$

$$\frac{\partial \mathcal{R}}{\partial \boldsymbol{\mu}} = \nabla^2 u + \nabla u \cdot \nabla(), \tag{3.25}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{u}} = 2\left(\mathbf{u} - \mathbf{u}^*\right), \tag{3.26}$$

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{\mu}} = 0. \tag{3.27}$$

The previous equations can now be discretized using Finite Difference schemes in order to obtain the final results. The discretization procedure presents similar characteristics to those encountered in Section 3.1, although now the gradient operator $\nabla() \approx G^{(1)}$ must be included in the equations. The following alternatives are available to discretize this operator in uniform grids:

$$G^{(1)}_{O_1} = \frac{1}{\Delta z}\begin{bmatrix} -1 & 1 & 0 & 0 & \dots & & & & \\ 0 & -1 & 1 & 0 & \dots & & & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ & & & & \dots & 0 & -1 & 1 & 0 \\ & & & & \dots & 0 & 0 & -1 & 1 \end{bmatrix}, \tag{3.28}$$

$$G^{(1)}_{O_2} = \frac{1}{2\ \Delta z}\begin{bmatrix} -1 & 0 & 1 & 0 & \dots & & & & \\ 0 & -1 & 0 & 1 & \dots & & & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ & & & & \dots & -1 & 0 & 1 & 0 \\ & & & & \dots & 0 & -1 & 0 & 1 \end{bmatrix}. \tag{3.29}$$

The operators $G^{(1)}_{O_1}$ and $G^{(1)}_{O_2}$ correspond to the first and second-order accurate Taylor series expansions of the $G^{(1)}$ operator. These operators are also known in the literature as upwind and central difference schemes respectively [15]. While the use of central difference schemes is generally preferred due to their lower discretization error, it must be noted that the $G^{(1)}_{O_2}$ operator is prone to create checkerboard patterns. In order to understand this issue, it can be noted for example that the vector $x = [0,\ 10,\ 0,\ 10]$ would present a zero-gradient according to the $G^{(1)}_{O_2}$ operator, whereas $G^{(1)}_{O_1}$ would perceive a significant zig-zag pattern. Different solutions have been developed historically to mitigate checkerboard pattern issues, such as the use of artificial diffusivity schemes [15]. These advanced schemes were developed since the use of fist-order upwind schemes is considered intractable in large-scale problems, due to the increased grid discretization requirements associated. However, the use of the $G^{(1)}_{O_1}$ operator was deemed feasible during the present exercise because of its small scale. As a result, the risk of developing checkerboard patterns was effectively discarded.

In order to generate the reference velocity profile ($\mathbf{u}^*$), the following dynamic viscosity profile ($\boldsymbol{\mu}^*$) was created:

$$\frac{\mu^*}{\mu_w} = 3 - cos^2\left(\pi x\right) - cos^2\left(2\pi x\right). \tag{3.30}$$

In eq. (3.30), μ_w corresponds to the value of the dynamic viscosity at the wall, which was taken to be unity. The reference velocity profile u^* was thus generated by inserting the expression developed for μ^* back into eq. (3.21). The viscosity profile finally recovered using Field Inversion can be found in Figure 3.2, along with the corresponding velocity profile. Here, it can be noted that the results are completely identical, and that the Field Inversion process can recover different scalar fields present in a system of PDE's. The Python code employed during this example can be found in Appendix A.2.

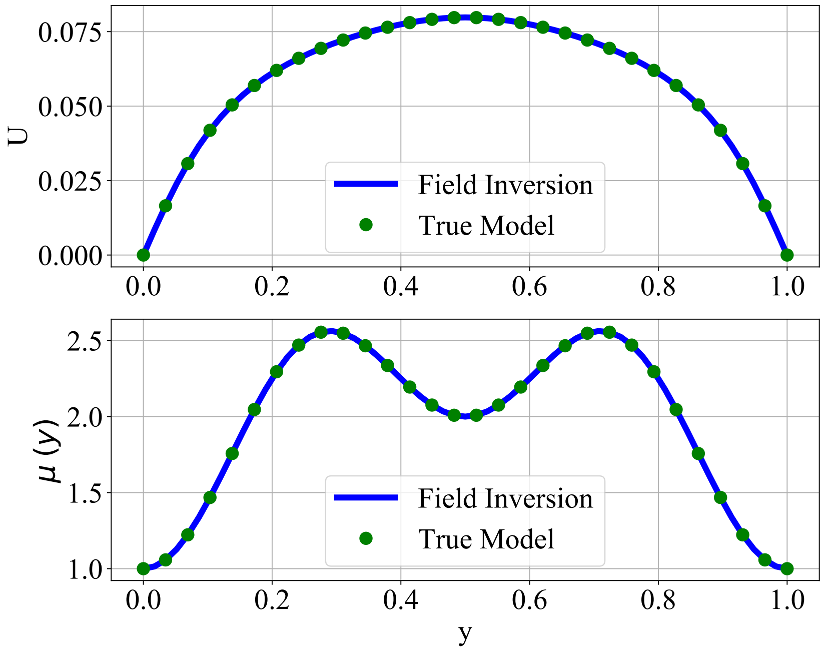


Figure 3.2: Results for the recovery of a dynamic viscosity profile using Field Inversion optimization.

3.3. Field Inversion Optimization for the k-ω Turbulence Model

In order to conclude the study of Field Inversion optimization, the last problem considered by Parish & Duraisamy (2016) [5] was replicated. The study case defined by these authors consisted in building corrections for the production of turbulent kinetic energy in the $k-\omega$ turbulence model [38]. The geometry considered for this problem corresponded to a fully-developed channel flow, which was governed by the following equations:

$$\nabla \cdot \left(\left(\nu + \nu_t \right) \nabla u \right) = \nabla P / \rho, \tag{3.31}$$

$$\beta(z)\ \nu_t \left(\nabla u \right)^2 - \alpha_k\ k\ \omega + \nabla \cdot \left[\left(\nu + \nu_t \frac{\sigma_k}{C_\mu} \right) \nabla k \right] = 0, \tag{3.32}$$

$$\gamma \left(\nabla u \right)^2 - \alpha_\omega\ \omega^2 + \nabla \cdot \left[\left(\nu + \nu_t \frac{\sigma_\omega}{C_\mu} \right) \nabla \omega \right] = 0, \tag{3.33}$$

$$\nu_t = C_\mu \frac{k}{\omega}. \tag{3.34}$$

The constants present in eqs. (3.31-3.34) take the following values: $\alpha_k = 0.09$, $\sigma_k = 0.6$, $\gamma = 0.52$, $\alpha_\omega = 0.0708$ and $\sigma_\omega = 0.5$. It can be noted that the Field Inversion variable ($\boldsymbol{\beta}$) has already been added to the $k - \omega$ turbulence model. This decision was taken, since the ground-truth model for this problem is either unknown or too expensive to compute (3-D DNS simulations). The velocity profiles employed to drive the Field Inversion process were taken from the DNS database assembled by Jimenez et al. (2008) [10], who studied channel flows at various Reynolds numbers (Re_τ). While Parish & Duraisamy (2016) [5] analyzed the different cases considered in this database, the present introductory example was restricted to studying the case where $Re_\tau = 950$, since the remaining cases can be trivially replicated by changing the input values given to the optimizer. As it was mentioned back in Section 2.3.7, the present optimization problem was driven by a simple cost function based on the squared velocity residuals:

$$\mathcal{J} = \sum_{i}^{N} \left(u_i - u_i^*\right)^2 . \tag{3.35}$$

In order to obtain the Jacobian matrix associated to eq. (3.35), the Discrete Adjoint Method must be applied as in the previous examples. However, the matrices required to drive this method no longer take trivial forms, due to the complexity of the RANS equations. Each of the governing equations (3.31-3.33) must now be differentiated with respect to the vector $\mathbf{W} = [\mathbf{u}, \mathbf{k}, \boldsymbol{\omega}]$. In the original implementation created by Parish & Duraisamy [5], this problem was handled by employing an Automatic Differentiation (AD) library to build the required matrixes. However, during the present exercise, a Sympy [19] script was created in order to build explicit expressions to compute every required matrix (e.g., $\partial\mathcal{R}/\partial\mathbf{W}$). The code presented in Appendix A.3 is able to yield code snippets, which can be copy-pasted into other programs, such as C++, Matlab or Fortran. The script can be trivially adapted to work with other governing equations as well. Thanks to this technique, it was possible to build an optimization program which was free of the limitations imposed by AD packages. Furthermore, the code generated by Sympy proved to be x1000 times faster than Python's Autograd library [77]. Therefore, the creation of this script was a larger contribution to the overall optimization procedure than the use of the Bold Drive Method with Added Momentum given by eqs. (3.17-3.20). This performance boost was also employed to run the simulations in high-levels programming languages such as Matlab or Python. The use of Python was finally preferred however, due to its open-source license.

Regarding the numerical discretization of the present problem, it is important to note that a non-uniform grid is required, since turbulent boundary layers develop in a narrow flow region. Furthermore, the $k - \omega$ turbulence model requires the first element to be placed well within the viscous sub-layer ($y^+_{w+1} \leq 1$). As a result, the grid must be highly stretched. The discretization procedure was based on the schemes developed by Otero et al. (2018) [12], where it was proven that a turbulent channel flow can be discretized employing around 100 grid points. The finite difference matrices $G^{(1)}$ and $G^{(2)}$ required for this problem now take an asymmetric form, which can be calculated using advanced Taylor expansions. A second Sympy script was created to compute the coefficients present in these matrices:

$$\begin{bmatrix} G^{(1)}_{i,i-1} & G^{(1)}_{i,i} & G^{(1)}_{i,i+1} \end{bmatrix} = \begin{bmatrix} \frac{-\Delta Y^+}{\Delta Y^-(\Delta Y^+ + \Delta Y^-)} & \frac{\Delta Y^+ - \Delta Y^-}{\Delta Y^+ \cdot \Delta Y^-} & \frac{\Delta Y^-}{\Delta Y^+(\Delta Y^+ + \Delta Y^-)} \end{bmatrix}, \tag{3.36}$$

$$\begin{bmatrix} G^{(2)}_{i,i-1} & G^{(2)}_{i,i} & G^{(2)}_{i,i+1} \end{bmatrix} = \begin{bmatrix} \frac{2}{\Delta Y^-(\Delta Y^+ + \Delta Y^-)} & \frac{-2}{\Delta Y^+ \cdot \Delta Y^-} & \frac{2}{\Delta Y^+(\Delta Y^+ + \Delta Y^-)} \end{bmatrix}. \tag{3.37}$$

The variables ΔY^+ and ΔY^- present in eqs. (3.36-3.37) correspond to the (positive) grid spacing values found at the north and south of each discretized point (i). While the complete channel geometry can be discretized employing the equations provided, it is important to notice the existence of a symmetry boundary condition at the channel center. This condition can be exploited in order to reduce the grid size by half, thereby speeding-up the simulations. One of the best alternatives to achieve this goal is to introduce the following changes into the differential operators $G^{(1)}$ and $G^{(2)}$:

$$\begin{bmatrix} G^{(1)}_{c,c-1} & G^{(1)}_{c,c} \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix}, \tag{3.38}$$

$$\begin{bmatrix} G^{(2)}_{c,c-1} & G^{(2)}_{c,c} \end{bmatrix} = \begin{bmatrix} \frac{2}{(\Delta Y^-)^2} & \frac{-2}{(\Delta Y^-)^2} \end{bmatrix}. \tag{3.39}$$

In eqs. (3.38-3.39), the sub-index (c) refers to the grid position at the channel center. The previous conditions represent the only required changes in the numerical discretization, since it can be proven that the linearized system of RANS equations remains invertible. Similar conditions can also be found for fully-developed pipe flows and other symmetric flows in nature.

The final results obtained during this exercise can be found in Figure 3.3, which presents the final $\boldsymbol{\beta}$ corrections, the dimensionless velocity profiles (U^+) and the modified k-equation production terms ($\beta(y)\ P_k$). The first curve, presented in blue, corresponds to the results found by the optimizer starting from $\boldsymbol{\beta} = 1$. It can be noted that this distribution managed to replicate the DNS velocity profile while applying lower $\boldsymbol{\beta}$ corrections than the other alternatives. Therefore, it can be concluded that this solution presented the lowest degree of over-fitting. The second solution curve presented in Figure 3.3, plotted in red, corresponds to the $\boldsymbol{\beta}$ distribution obtained after initializing the optimizer using the corrections obtained by Parish et al. (2016) [5]. While this curve should have remained identical to its initial values, a small drift was observed in the solution until a local minimum was found. The differences between both solutions can be caused either by the numerical properties of the schemes employed, or by the convergence criterion applied. Both of these solutions present a higher degree of over-fitting according to the present analysis nonetheless, since the $\boldsymbol{\beta}$ corrections employed grow substantially without producing a substantial change in the velocity profiles. The modifications encountered in the production of turbulent kinetic energy in the $k-\omega$ turbulence model can also be found in Figure 3.3. Here, it can be seen that the optimizer trimmed the production of turbulent kinetic energy in the buffer layer ($5 < y^+ < 30$). These corrections are natural, since it can be noted that the standard solution of the $k-\omega$ turbulence model presents a significant deviation with respect to the DNS data in this region. However, it is interesting to note that the optimizer managed to change the behavior of the model in the core flow region by only applying corrections in the log-layer ($30 < y^+ < 300$).

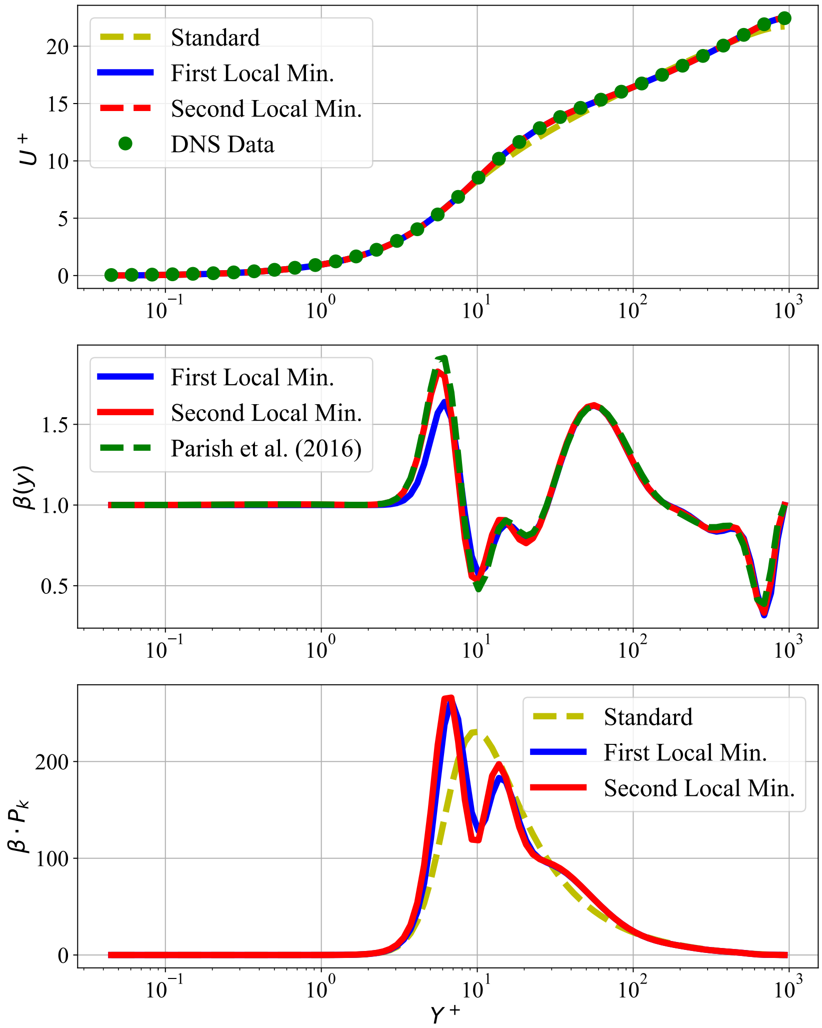


Figure 3.3: Field Inversion results for the optimization of the $k-\omega$ turbulence model in a 1-D channel flow [5].

Finally, it must be noted that an explicit analytic solution for the $\boldsymbol{\beta}$ corrections can be found in this exercise by imposing $u = u^*$ upon the system. However, the corrections obtained therein would present a high degree of over-fitting, since it can be observed in Figure 3.3 that marginal improvements in the solution must be compensated by substantial changes in the $\boldsymbol{\beta}$ corrections. As a result, it is never recommended to impose direct DNS data upon a turbulence model.

3.4. Deep Dreaming the Existence of the Reynolds Number

This Section will present the Deep Learning system created to predict the drag coefficient of a cylinder (C_D) after inferring the existence of the Reynolds number (Re_D). The data considered for this exercise can be found in Figure 3.4, which presents the drag coefficient of a smooth cylinder as a function of the Reynolds number ($C_D = f(Re)$) [24]. While a clear relation exists between the two plotted variables, it is important to note that the curve presented cannot be reduced into a simple correlation, since the process is marked by a transition from a laminar to a turbulent flow regime. Regarding the sampling process of the distribution presented in Figure 3.4, it is important to note that the original Bézier curve created by White (2011) [24] was imported into the code presented in Appendix A.4 thanks to the use of the Python library Svgpathtools [78]. As a result, negligible numerical errors were associated to this process.

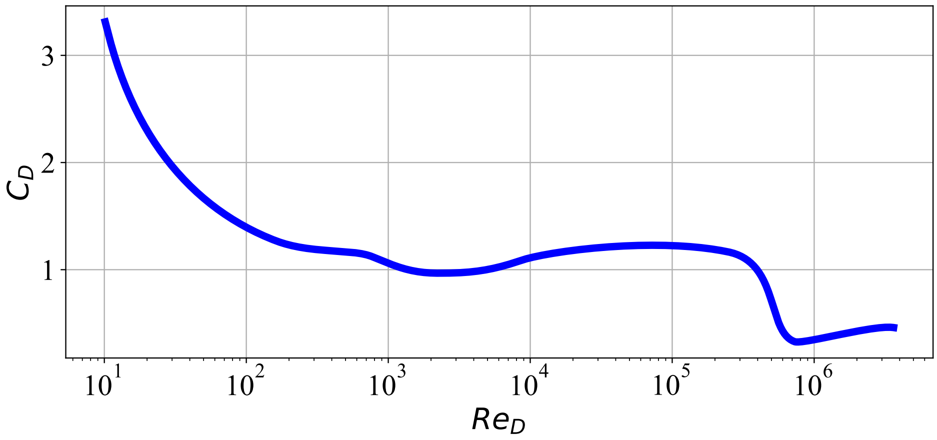


Figure 3.4: Drag coefficient of a smooth cylinder as a function of the Reynolds number. Source: White (2011) [24].

Before attempting to infer the existence of the Reynolds number based on the drag coefficient of a cylinder, it was decided to search for a Neural Network architecture capable of representing the relation $C_D = f(Re)$. The architecture found during this process can be found depicted in Figure 3.5. Here, it can be noted that the Neural Network only presents three hidden layers. Choosing a small number of hidden layers is considered ideal for small datasets, since Deep Neural Networks (DNN) present an issue known as vanishing gradients [57]. This problem arises due to the properties of chain rule differentiation, which imply that the gradients of the first neurons must be multiplied by the gradients of each subsequent activation function. As a result, the first neurons in a DNN can be difficult to train even while presenting an erroneous configuration. However, this phenomenon is unlikely to arise in a Neural Network presenting only three hidden layers. It can be further noted that Hyperbolic Tangent Neurons were employed in Figure 3.5. As it was mentioned in Chapter 2, these neurons constitute a strong alternative in fluid mechanics, since they are able to create smooth transitions and result easier to train than standard Sigmoid neurons. The magnitude of the drag coefficient (C_D) does not constitute a problem, since the training data can be re-scaled to fit the range [-1,1]. The Reynolds number was pre-scaled using a logarithmic transformation as well, since it can be noted in Figure 3.4 that $C_D = f(log(Re))$ constitutes a better representation of the problem than $C_D = f(Re)$. Further applying a linear transformation to the input features was deemed unnecessary, since the weights of the first hidden layer in the Neural Network can also develop adequate conversion factors.

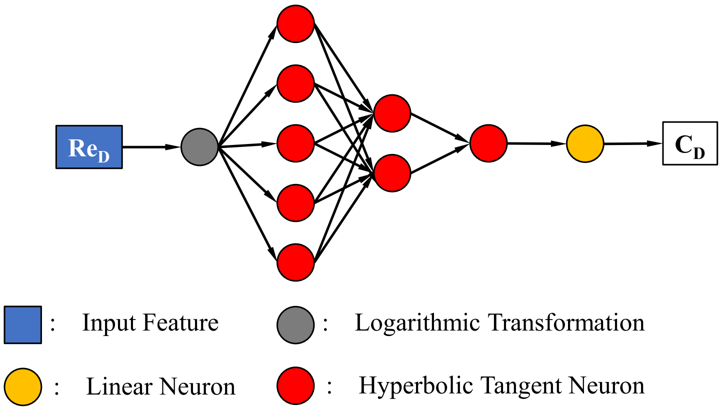


Figure 3.5: Neural Network system capable of reconstructing the functional relation between the drag coefficient of a cylinder and the Reynolds number ($C_D = f(Re_D)$).

In order to carry out the final exercise, the Neural Network presented in Figure 3.5 was modified in order to incorporate several input features ($\mathbf{X}$) representing the initial experimental data. The configuration obtained can be found in Figure 3.6, where it can be seen that the Neural Network now receives pairs of feature points (U, D, ν, ρ). Since $Re_D = U \cdot D/\nu$, it can be noted that the density feature (ρ) corresponds to uncorrelated information, and that the Neural Network is thus tasked with filtering out this quantity. Furthermore, it can be noted that the second neuron present in the first layer of Figure 3.6 is also redundant, since only one neuron is required to form the Reynolds number. The cost function employed to drive the training process was given by the following expression:

$$\mathcal{J} = \sum_{i=1}^{N} \left(C_{D,i} - C^*_{D,i}\right)^2 . \tag{3.40}$$

In eq. (3.40), the variable $\mathbf{C}^*_{\mathbf{D}}$ corresponds to the ground-truth labels employed to drive the training process. The data labels employed during the present exercise contained a 30% noise level with respect to the original curve presented in Figure 3.4. Thanks to this consideration, it was possible to prove that Deep Learning systems containing logarithmic neurons are able to find the mean trends in noisy distributions. Another important detail added to the model was the introduction of multiple initialization trials for the training procedure. The main idea behind this technique is to let the computer train different configurations for a moderate number of iterations, and then to only retain the configuration which achieved the lowest training error. Since Jacobian optimization algorithms present exponential convergence rates during the first iterations, the previous technique can even be used to assess the maximum performance which can be expected from a Neural Network. The use of multiple initialization trials constitutes a mandatory practice nonetheless, since erroneous initial configurations can present null gradients due to the saturation of their activation functions [57].

The results obtained during the final training process can be found in Figure 3.7, where it can be seen that the Neural Network managed to accurately reconstruct the relation $C_D = f(Re)$ even under the presence of noise. The errors found in the first layer of Hyperbolic Tangent Neurons while trying to reconstruct the Reynolds number can be found listed in Table 3.1. Here, it can be noted that the Neural Network system effectively managed to cancel the influence of the density feature $\boldsymbol{\rho}$. The proportion required to form the Reynolds number in log-space corresponds to the vector $[1, 1, -1, 0]$ according to the stack of features (U, D, ν, ρ). The scaling factors α present in Table 3.1 correspond to the overall multipliers chosen by the Neural Network for each column of weights. This property arises since $C_D = f(Re)$ can also be expressed as $C_D = g(Re^{\alpha})$. The Neural Network was thus able to choose the scaling factors (α) which facilitated the training procedure. As a result, the final predictive system had the form $C_D = f(Re) = g(Re^{\alpha_1}, Re^{\alpha_2}, ...)$.

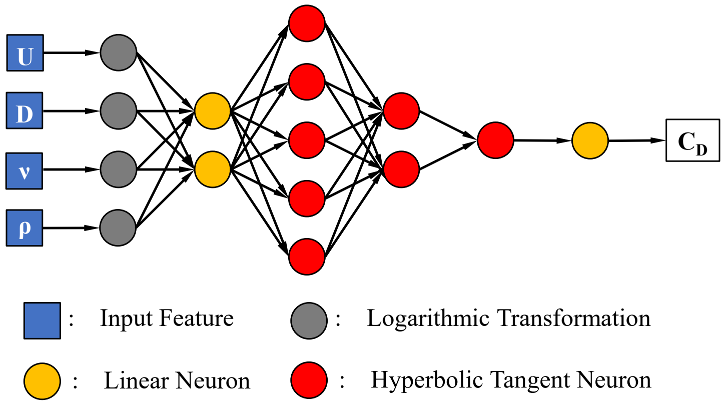


Figure 3.6: Neural Network architecture employed to infer the existence of the Reynolds number based on the drag coefficient of a smooth cylinder.

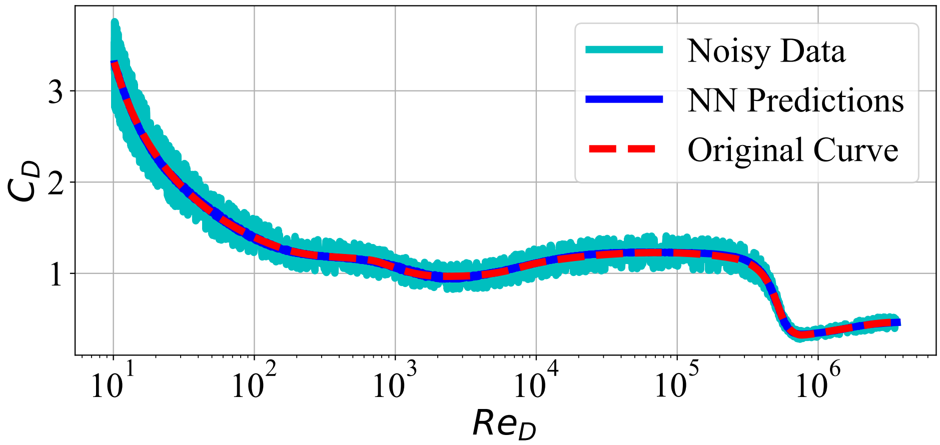


Figure 3.7: Results obtained after training the Neural Network architecture presented in Figure 3.6.

	Neuron 1	Neuron 2	Neuron 3	Neuron 4	Neuron 5
log (U)	-0.1%	0.0%	1.1%	0.6%	-0.2%
log (D)	0.0%	-0.1%	-0.2%	-0.1%	0.0%
log (ν)	-0.2%	0.0%	0.9%	0.5%	-0.3%
log (ρ)	-0.2%	-0.1%	0.2%	0.0%	-0.2%
Scale factor (α)	-3.827	7.541	-1.223	1.554	-1.857

Table 3.1: Errors perceived by the first layer of Hyperbolic Tangent Neurons in the Deep Learning system presented in Figure 3.6 while trying to reconstruct the Reynolds number. The exact weights required to reconstruct the Reynolds number are given by the vector $[1, 1, -1, 0]$.

Based on the overall robustness exhibited by logarithmic neurons, it was concluded that these neurons could be used in fluid mechanics to perform automatic feature selection. This property was actively employed to simplify the selection of the Neural Network architectures employed in Chapters 4 and 5. Furthermore, the results obtained during the present exercise indicate that logarithmic neurons could prove useful in heat transfer problems, since these neurons can automatically discover relations based on dimensionless groups such as the Prandtl or the Grashof number [1] given only experimental data.

4

Data-Driven RANS Turbulence Modelling for Variable Property Flows

The present Chapter will describe the application of the principles of Field Inversion Machine Learning (FIML) to channels flows subject to strong property variations. In order to start the analysis, the application of the RANS equations to fully-developed channel flows will be first discussed in Section 4.1. Then, in Section 4.2, the DNS database of turbulent channel flows assembled will be presented, along with a review of the performance of traditional RANS turbulence models within this dataset. The Field Inversion methodology developed during the present thesis will be introduced in Section 4.3. Based on the formulation established, the Field Inversion results obtained for various configurations will be presented in Section 4.4, where the feasibility of replicating the patterns observed in each set will also be discussed. Finally, in Section 4.5, different Neural Network systems will be built for the most promising sets of Field Inversion corrections previously found, and a new intelligent relaxation factor methodology for these corrections will be derived. A global discussion of the results obtained through the use of FIML will be presented at the end of this Chapter.

4.1. RANS Equations Applied to Channel Flows

The present Section will describe how the RANS equations presented in Chapter 2 can be applied to fully-developed channel flows. The overall geometry of these flows can be found depicted in Figure 4.1, which is also representative of the flow configurations encountered back in Sections 3.2 and 3.3 from Chapter 3. Here, it can be noted that the mean flow reaches a steady-state configuration along the channel, such that gradients only appear in the wall-normal direction (y). The previous fact can be transformed into the following identities to simplify the RANS governing equations:

$$\frac{\partial \alpha}{\partial x} = \frac{\partial \alpha}{\partial z} = \frac{\partial \alpha}{\partial t} = 0. \tag{4.1}$$

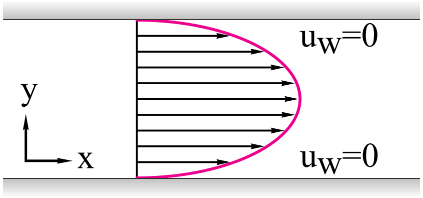


Figure 4.1: Schematic representation of a channel flow. Image adapted from White (2011) [24].

In eq. (4.1), the scalar α corresponds to any mean flow variable present in the RANS equations. The only exception to this rule is given by the pressure gradient $\partial P/\partial x$, which corresponds to a constant in the configuration presented in Figure 4.1. Furthermore, it can be noted that the velocity profile for a channel flow has the following vector form:

$$\mathbf{u} = \begin{bmatrix} u \\ 0 \\ 0 \end{bmatrix}. \tag{4.2}$$

Eq. (4.2) states that the mean flow can only advance in the x-direction, which is consistent with the coordinates presented in Figure 4.1. One important consequence of this relation is that eqs. (4.1-4.2) can be combined in order to prove that the mean convective transport terms are strictly zero in a channel flow. As a result, the flow present in these channels can be transformed into a 1-D problem. The results obtained after simplifying the original set of RANS equations (2.45-2.49) under the Boussinesq approximation take the following form:

$$\frac{\partial}{\partial y}\left((\mu + \mu_t)\,\frac{\partial u}{\partial y}\right) = \frac{\partial P}{\partial x}, \tag{4.3}$$

$$\frac{\partial}{\partial y}\left(\left(\lambda + \frac{c_p \mu_t}{Pr_t}\right)\frac{\partial T}{\partial y}\right) + \tau(y)\frac{\partial u}{\partial y} + S_I = 0, \tag{4.4}$$

$$\begin{aligned} \tau(y) &= (\mu + \mu_t)\,\frac{\partial u}{\partial y} \\ &= \tau_w\left(1 - \frac{y}{H}\right). \end{aligned} \tag{4.5}$$

In eqs. (4.3-4.5), it can be noted that the influence of turbulence in channel flows is confined to the scalar field μ_t, which is always well-defined for a 1-D velocity profile. The previous equations can alternatively be derived from a free-body diagram, by assuming that the different streamlines present in the flow interact with each other according to an effective viscosity ($\mu_{eff} = \mu + \mu_t$). Beyond these considerations, it can be noted that eq. (4.5) imposes a remarkably strong bound upon turbulent channel flows, since it implies that the chaotic energy cascade of turbulence must adapt itself such that a linear shear stress profile is always obtained.

4.2. DNS Database Assembled

This Section will describe the DNS database assembled to study turbulent channel flows subject to strong property gradients. The DNS cases considered were obtained from the PhD thesis work of Ashish Patel (2017) [9] and the supersonic flow database shared by Trettel & Larson (2016) [11]. The different parameters employed in each database can be found listed in Tables 4.1 and 4.2 respectively. Regarding these flow cases, it is important to note that both databases handle the energy equation of fluids differently. The DNS cases considered by Patel (2017) [9] employed a low-Mach number approximation, under which the effects of viscous heating were explicitly cancelled. The temperature gradients found in this database were thus given by volumetric heat source terms $S_i = \phi/(Re_\tau Pr)$ calculated according to the values presented in Table 4.1. The DNS flow cases considered by Trettel & Larson (2016) [11] employed the opposite assumptions, since only supersonic flows subject to strong viscous heating effects were modelled. These differences helped to create a diverse dataset nevertheless, where it was possible to identify patterns cross a wide variety of DNS flows.

Regarding the DNS cases presented in Table 4.1, it must be noted that the flow cases (1-11) follow the nomenclature presented in Table 3.1 from the PhD thesis of Ashish Patel (2017) [9], whereas the DNS cases (12-15) correspond to the additional simulations introduced in Table 6.1 of his thesis. The final DNS cases (16-18), marked by the prefix JFM, refer to the simulations presented by Pecnik et al. (2017) [6], which correspond to variants of the original models subject to stronger heat source terms. The coefficients listed in Tables 4.1 and 4.2 for μ, ρ and λ refer to the exponents employed to build temperature-dependency laws with the form $x/x_w = (T/T_w)^k$. These relations were simplified however, since the arbitrary scaling conventions presented in Section 2.1.3 indicate that $T_w = 1$ and $\rho_w = 1$. The values of c_p presented in Tables 4.1 and 4.2 were also scaled such that $c_p = 1/Ec$.

Finally, it must be mentioned that the global DNS database was augmented by employing the incompressible DNS flow cases published by Jimenez et al. (2008) [10], who simulated channel flows at various Reynolds Numbers. The different flow cases considered in this database can be found listed in Table 4.3. Thanks to these reference flows, it was possible to clearly assess the influence of the Reynolds number in the simulations.

ID	**Case**	$\boldsymbol{\rho/\rho_w}$	$\boldsymbol{\mu/\mu_w}$	$\boldsymbol{\lambda/\lambda_w}$	$\mathbf{Re}_\tau$	**Pr**	$\mathbf{c_p}$	$\boldsymbol{\phi}$
1	$CP395$	0	0	0	395	1	1	17.55
2	CRe_τ^*	-1	-0.5	0	395	1	1	17.55
3	$SRe_{\tau GL}^*$	0	1.2	0	395	1	1	18.55
4	GL	-1	0.7	0	395	1	1	17.55
5	$LL1$	0	-1	0	150	1	1	29
6	$SRe_{\tau LL}^*$	0.6	-0.75	0	150	1	1	31.5
7	$SRe_{\tau Cv}^*$	0	-0.5	0	395	1	1	17.55
8	Cv	-1	-1	0	395	1	1	16
9	$LL2$	0	-1	0	395	1	1	17.55
10	$CP150$	0	0	0	150	1	1	0
11	$CP550$	0	0	0	550	1	1	0
12	$CRe_\tau^*CPr^*$	-1	-0.5	-0.5	395	1	1	17.55
13	$GLCPr^*$	-1	0.7	0.7	395	1	1	17.55
14	$V\lambda SPr_{LL}^*$	0	0	1	395	1	1	17.55
15	$CP395_{Pr4}$	0	0	0	395	4	1	34
16	$JFM.CRe_\tau^*$	-1	-0.5	0	395	1	1	95
17	$JFM.GL$	-1	0.7	0	950	1	1	75
18	$JFM.LL$	0	-1	0	150	1	1	62

Table 4.1: DNS flow cases extracted from the PhD thesis of Ashish Patel (2017) [9] and the work of Pecnik et al. (2017) [6].

ID	Case	ρ/ρ_w	μ/μ_w	λ/λ_w	Re_τ	Pr	c_p
19	M0.7R400	-1	0.75	0.75	437	0.7	1743
20	M0.7R600	-1	0.75	0.75	652	0.7	1927
21	M1.7R200	-1	0.75	0.75	322	0.7	357
22	M1.7R400	-1	0.75	0.75	663	0.7	418
23	M1.7R600	-1	0.75	0.75	972	0.7	468
24	M3.0R200	-1	0.75	0.75	650	0.7	210
25	M3.0R400	-1	0.75	0.75	1232	0.7	239
26	M3.0R600	-1	0.75	0.75	1876	0.7	266
27	M4.0R200	-1	0.75	0.75	1017	0.7	179

Table 4.2: DNS flow cases simulated by Trettel & Larson (2016) [11].

ID	Case	Re_τ
28	$IC.Re180$	180
29	$IC.Re550$	550
30	$IC.Re950$	950
31	$IC.Re2000$	2000
32	$IC.Re4200$	4200

Table 4.3: Incompressible DNS flow cases simulated by Jimenez et al. (2008) [10].

4.2.1. Performance of Traditional RANS Turbulence Models in the Dataset

Before starting the Field Inversion optimization process, it was decided to assess the preliminary performance of various RANS turbulence models in the dataset, such that the best optimization targets could be established. The benchmark was based on the 5 DNS cases chosen for comparison by Pecnik et al. (2017) [6]. Under the present nomenclature, the simulations employed in such publication correspond to the following DNS cases: $CP395$, $JFM.CRe_\tau^*$, $JFM.GL$, $JFM.LL$ and $M4.0R200$. These cases represent a wide variety of flow scenarios, since they include a constant-property case (CP), a gas-like fluid (GL), a liquid substance (LL) and the strongest supersonic flow (M4) studied by Trettel & Larson (2016) [11]. The results obtained during the benchmark can be found in Figure 4.2, which presents a comparison of the predictions made by each RANS turbulence model described back in Section 2.1.7. The code employed to drive the simulations was based on the CFD solvers developed by Otero et al. (2018) [12], who studied various alternatives for modelling variable-property flows. The diffusivity corrections proposed therein were omitted, however, since the Field Inversion optimization process should produce a new set of modifications. Beyond these considerations, it is important to mention that the molecular properties (ρ and μ) employed during the RANS turbulence modelling benchmark were extracted from the DNS database and kept frozen during the simulations. This alternative is often employed in the literature, since it allows researchers to assess the performance of a RANS turbulence model under optimal conditions. However, it is important to remember that any final CFD solver must incorporate the energy equation of fluids as well.

Regarding the trends observed in Figure 4.2, it can be first noted that the Cess algebraic model [40, 41] described back in Section 2.1.7 achieved the best overall performance. While these results can seem natural, since this model was explicitly calibrated for the geometry considered, it is still important to notice how the modified dimensionless wall distance y^* and the SLS (Semi-Locally Scaled) Reynolds number Re_τ^* managed to achieve good results. The second model with the highest level of performance is given by the Spalart-Allmaras (SA) turbulence model [8], which solves for a transformed kinematic eddy viscosity $\hat{\nu}$. During the review performed by Otero et al. (2018) [12], it was theorized that the SA turbulence model performed well in variable-property flows since its governing equation is purely kinematic, which implies a certain degree of local scaling. The results obtained by the MK turbulence model can seem contradictory, since this model reached the highest overall error in the $JFM.CRe_{\tau}^{*}$ case, yet it managed to achieve a good performance in the rest of the test cases. During the benchmark of the SST turbulence model, it could be observed that this model achieved an unsatisfactory performance in nearly every flow case tested, whereas the $v'^{2}-f$ model offered similar levels of performance except for the liquid-like case ($JFM.LL$).

Based on the previous analysis, it was decided to employ the Spalart-Allmaras and the MK turbulence model as targets for Field Inversion optimization. Optimizing the MK turbulence model poses an interesting challenge, since this model can range from requiring negligible corrections to exhibiting the worst performance found in the dataset ($JFM.CRe_{\tau}^{*}$ case). As a result, a Neural Network system building corrections for this model must actively determine which conditions trigger large deficits in the turbulence budgets of the RANS equations. The Spalart-Allmaras turbulence model also constitutes an important target for optimization, since its high preliminary performance could allow a Neural Network to reach exceptional levels of accuracy. Furthermore, the SA turbulence model has been regarded by previous authors as a good target for Field Inversion optimization [76], since only one set of $\boldsymbol{\beta}$ corrections is required for its governing equation, and thus the Elbow Method can be readily applied to reach a decision [76]. Further optimizing the SST and the $v'^{2}-f$ turbulence model was deemed unnecessary, since it would not add new insights to the present study. Moreover, the optimization cases considered under the present circumstances already required defining over 700 Field Inversion problems, and over a month of CPU hours were employed to build the final FIML system.

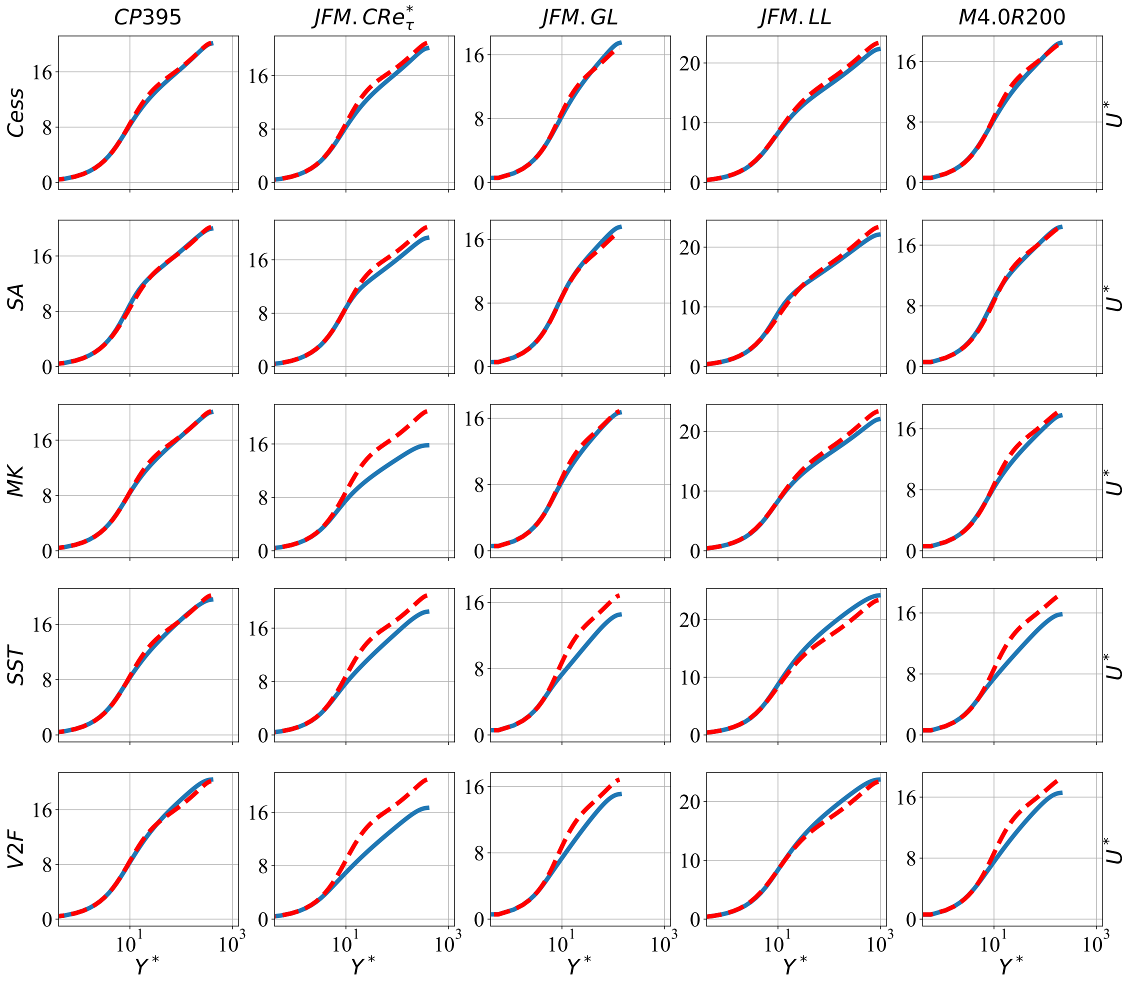


Figure 4.2: Results obtained by the RANS turbulence models listed in Section 2.1.7 for the test set defined by Pecnik et al. (2017) [6]. The blue curves correspond to the RANS simulations performed, whereas the red dashed lines represent the original DNS Data.

4.3. Field Inversion Optimization

This Section will describe the different Field Inversion techniques employed during the analysis of variable property flows. First, in Section 4.3.1, the Field Inversion process developed for the MK turbulence model will be described. This formulation constitutes one of the most challenging aspects studied, since two possible sets of corrections can be obtained for the $k-\epsilon$ turbulence model. Then, in Section 4.3.2, the Field Inversion methodology developed for the SA turbulence model will be presented. The velocity profiles obtained during the Field Inversion study for the MK turbulence model will be further used to create an independent Field Inversion analysis for the turbulent Prandtl number (Pr_t) in Section 4.3.3. Thanks to this analysis, it will be possible to determine whether a Field Inversion optimizer can manage to recover the DNS values of Pr_t or not.

4.3.1. Field Inversion Formulation for the MK Turbulence Model

The present Section will describe the Field Inversion methodology created for the MK turbulence model, along with a careful review of the different possibilities available. Before starting the derivation process, it is important to define the nature of the corrections that will be applied to the problem. As it was mentioned in Chapter 3, it is possible to either build explicit $\boldsymbol{\delta}$ source term corrections for a model, or to employ $\boldsymbol{\beta}$ multipliers for specific terms. Regarding these choices, the numerical analysis performed revealed that building $\boldsymbol{\delta}$ corrections created an unstable solver while trying to optimize a

RANS turbulence model. In order to illustrate this, Figure 4.3 presents the turbulence budgets of the MK turbulence model for the case $JFM.CRe_{\tau}^{*}$. Here, it can be observed that every term present in the RANS turbulence budgets vanishes as the channel center is reached. Due to this reason, the zones marked by rectangles in Figure 4.3 correspond to areas where the model can diverge due to the presence of explicit $\boldsymbol{\delta}$ source term corrections, such as the triangular spikes presented. The previous risk of divergence was confirmed during the numerical experiments performed using the Bold Drive Method and the BFGS algorithm, which attempted to accelerate the optimization process by building aggressive sets of corrections. The solvers in fact diverged within the first global iterations. Further references to the issues caused by adding explicit $\boldsymbol{\delta}$ corrections can be found in the work of Duraisamy et al. (2017) [79].

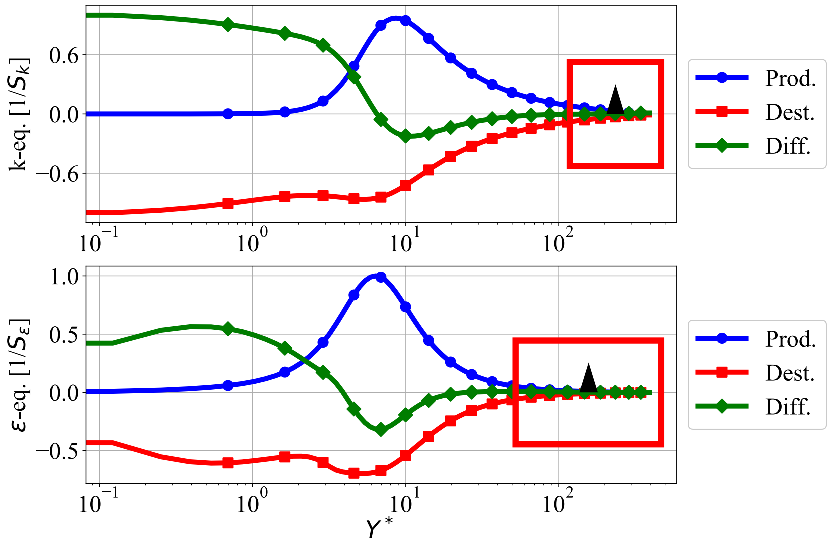


Figure 4.3: Turbulence budgets obtained for the MK model [7] during the study of the DNS case $JFM.CRe_{\tau}^{*}$.

Since the previous analysis revealed that adding explicit $\boldsymbol{\delta}$ corrections to a RANS model creates an unstable solver, it was decided to study which type of $\boldsymbol{\beta}$ multipliers created the best solution environment. Based on the trends observed in Figure 4.3, it was decided to build $\boldsymbol{\beta}$ corrections based on the destruction terms of both equations present in the MK turbulence model. These terms were chosen, since it can be noted that both the production and the diffusion terms present in Figure 4.3 are only active in specific locations, and thus they do not constitute an appropriate basis to build global corrections. The choice of a specific term to build $\boldsymbol{\beta}$ corrections does not have further consequences, since it was proven in Section 3.1 that adding $\boldsymbol{\beta}$ multipliers to a system is equivalent to incorporating explicit $\boldsymbol{\delta}$ source term corrections [5, 79]. From this point of view, the present discussion only affects the numerical stability of the models created. The final equations employed by the Field Inversion optimizer take the following form:

$$P_k - \beta_k \; \rho \; \epsilon + \nabla \cdot \left(\left(\mu + \frac{\mu_t}{\sigma_k} \right) \nabla k \right) = 0, \tag{4.6}$$

$$C_{\epsilon 1} \; P_k \; \frac{\epsilon}{k} - \beta_\epsilon \; C_{\epsilon 2} \; f_\epsilon \; \rho \; \frac{\epsilon^2}{k} + \nabla \cdot \left(\left(\mu + \frac{\mu_t}{\sigma_\epsilon} \right) \nabla \epsilon \right) = 0, \tag{4.7}$$

$$\mu_t = C_\mu \; f_\mu \; \rho \; \frac{k^2}{\epsilon}. \tag{4.8}$$

In eqs. (4.6-4.7), the variables β_k and β_ϵ represent the Field Inversion multipliers added to each respective equation. Any missing terms in the MK turbulence model, such as P_k or f_ϵ, can be computed using eqs. (2.68-2.71) from Section 2.1.7 as usual. It can be further noted that eqs. (4.6-4.7) were already simplified employing the identities (4.1-4.2) listed in Section 4.1, which cancel any terms related to the material derivatives of the mean flow.

One important innovation made during the present analysis was to consider the following definition for the cost function ($\mathcal{J}$):

$$\mathcal{J} = \sum_{i=1}^{N} I_U \left(\frac{u_i - u_i^*}{S_U} \right)^2 + I_k \left(\frac{\delta_k}{S_k} \right)^2 + I_\epsilon \left(\frac{\delta_\epsilon}{S_\epsilon} \right)^2 . \tag{4.9}$$

In eq. (4.9), the variables δ_k and δ_ϵ represent the effective source terms added to the MK turbulence model, which are given by the following equations:

$$\delta_k = \rho \, \epsilon \, (1 - \beta_k) , \tag{4.10}$$

$$\delta_\epsilon = C_{\epsilon 2} \, f_\epsilon \, \rho \, \frac{\epsilon^2}{k} \, (1 - \beta_\epsilon) . \tag{4.11}$$

While it can be noted that the previous relations take a complex form, the use of eqs. (4.10-4.11) allows the optimizer to seek for explicit ($\boldsymbol{\delta}$) source terms while remaining numerically stable, since only $\boldsymbol{\beta}$ multipliers are added to the governing equations. The variables I_U, I_k and I_ϵ present in eq. (4.9) refer to the importance assigned to minimizing each target objective, whereas the terms S_U, S_k and S_ϵ refer to the scales employed in each case:

$$S_U = max(|u^*|), \tag{4.12}$$

$$S_k = max\left(|P_k|, \; |\rho\epsilon|, \; \left| \nabla \cdot \left(\left(\mu + \frac{\mu_t}{\sigma_k} \right) \nabla k \right) \right| \right), \tag{4.13}$$

$$S_\epsilon = max\left(\left| C_{\epsilon 1} P_k \frac{\epsilon}{k} \right|, \; \left| C_{\epsilon 2} f_\epsilon \rho \frac{\epsilon^2}{k} \right|, \; \left| \nabla \cdot \left(\left(\mu + \frac{\mu_t}{\sigma_\epsilon} \right) \nabla \epsilon \right) \right| \right). \tag{4.14}$$

In eqs. (4.12-4.14), it can be seen that each scale corresponds to the maximum preliminary value for its respective category. The values of S_k and S_ϵ are intended to be computed only once based on the initial results for the MK turbulence model, such that they become constants. In Section 4.4, it will be further proven that these scales help the Elbow Method to obtain inflection points at predictable locations. The importance factors I_U, I_k and I_ϵ were determined through an hyper-parameter optimization study.

Based on the previous methodology, it was possible to obtain a well-balanced framework to optimize the MK turbulence model, which proved to be exceptionally stable. The Field Inversion optimization process was further converted into an automatic routine, which was successfully run over 400 times. Before concluding this section, it is important to note that the present methodology can be trivially adapted, such that only $\boldsymbol{\beta}_\mathbf{k}$ corrections are employed, or vice-versa. The previous possibilities also constituted part of the hyper-parameter optimization study performed.

4.3.2. Field Inversion Optimization for the Spalart-Allmaras Turbulence Model

This Section will describe the Field Inversion formulation created for the Spalart-Allmaras (SA) turbulence model [8], which is based on the methodology described in Section 4.3.1. A representative example of the turbulence budgets for the SA model can be found in Figure 4.4, which corresponds to the trends obtained for the DNS case $JFM.CRe_\tau^*$. The small spikes observed in this Figure at the channel center are a feature inherent to this model. The production term is free from these issues however, and it can be further noted that its action spawns almost the entire channel. As a result, it was chosen to build $\boldsymbol{\beta}$ multipliers based on the production term of the SA model. The governing equation employed was the following:

$$0 = \beta_{SA} \, c_{b1} \hat{S} \hat{\nu} - c_{w1} f_w \left(\frac{\hat{\nu}}{y} \right)^2 + \frac{c_{b2}}{c_{b3}} \left(\nabla \hat{\nu} \right)^2 + \frac{1}{c_{b3}} \nabla \cdot \left(\left(\nu + \hat{\nu} \right) \nabla \hat{\nu} \right) . \tag{4.15}$$

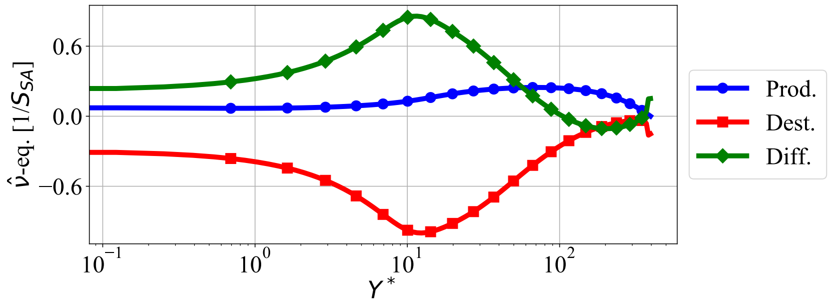


Figure 4.4: Initial turbulence budgets for the Spalart-Allmaras model [8] during the study of the DNS case $JFM.CRe_\tau^*$.

Based on eq. (4.15), it can be inferred that the source term (δ_{SA}) added to the SA turbulence model takes the following form:

$$\delta_{SA} = c_{b1}\hat{S}\hat{\nu}\left(\beta_{SA} - 1\right). \tag{4.16}$$

The cost function for this problem was thus defined as:

$$\mathcal{J} = \sum_{i=1}^{N} I_U \left(\frac{u_i - u_i^*}{S_U}\right)^2 + I_{SA}\left(\frac{\delta_{SA}}{S_{SA}}\right)^2. \tag{4.17}$$

Where the scaling terms S_U and S_{SA} are given by:

$$S_U = max(|u^*|), \tag{4.18}$$

$$S_{SA} = max\left(\left|c_{b1}\hat{S}\hat{\nu}\right|, \left|c_{w1}f_W\left(\frac{\hat{\nu}}{y}\right)^2\right|, \left|\frac{c_{b2}}{c_{b3}}\left(\nabla\hat{\nu}\right)^2\right|, \left|\frac{1}{c_{b3}}\nabla\cdot\left(\left(\nu + \hat{\nu}\right)\nabla\hat{\nu}\right)\right|\right). \tag{4.19}$$

The previous set of equations (4.15-4.19) define a numerically stable scheme which can be automatized employing the Bold Drive Method, as it occurred with the MK turbulence model. In this case however, the only trade-off present in the cost function is given by the relation I_U/I_{SA}, since the individual values of I_U and I_{SA} only scale the learning rates employed by the optimizer. As a result, it was decided to choose $I_{SA} = 1$ during the hyper-parameter optimization study for I_U. The results found after analyzing the present problem will be discussed in Section 4.4.2.

4.3.3. Field Inversion Study for the Turbulent Prandtl Number

The present Section will describe the Field Inversion optimization study created for the turbulent Prandtl number (Pr_T), which constitutes an independent parameter in the solution of the general RANS equations. While the values of Pr_T may present a secondary importance compared to the RANS turbulence model chosen, a wrong temperature profile can severely affect the molecular properties employed in the rest of the calculations. However, it was found that $Pr_T \approx 1$ constituted a good approximation for every DNS case present in the dataset assembled. Thanks to this simple approximation, the remaining RANS Field Inversion optimizers were able to develop highly accurate solutions, even though the DNS analysis performed by Patel (2017) [9] revealed that the values of Pr_T ranged from 0.4 to 1.15 in reality. Based on these insights, it was concluded that the approximation $Pr_T \approx 1$ managed to produce good results since it accurately modelled the most active turbulent flow region, which is the buffer layer.

Beyond the previous considerations, it was decided to study whether a Field Inversion optimizer could manage to replicate the DNS values for the turbulent Prandtl number ($Pr_{T,DNS}$) or not. Even if the same trends were not obtained, the presence of consistent patterns in the Field Inversion results could still create improved Machine Learning models. The formulation created for this problem was the following:

$$\frac{\partial}{\partial y}\left(\left(\lambda + \frac{c_P\mu_t}{Pr_t}\right)\frac{\partial T}{\partial y}\right) + \tau(y)\frac{\partial u_{opt}}{\partial y} + S_i = 0. \tag{4.20}$$

In eq. (4.20), the variable u_{opt} corresponds to the optimized velocity profiles which were found during the Field Inversion study for the MK turbulence model. It was decided to prefer these velocity profiles over the raw DNS data, since their gradients do not contain noise. The Field Inversion variable considered in eq. (4.20) corresponds directly to Pr_T, since this is the ultimate target of the study. The following cost function was employed to drive the optimization process:

$$\mathcal{J} = \sum_{i=1}^{N} I_T \left(T_i - T_i^*\right)^2 + \left(Pr_{T,i} - 1\right)^2 . \tag{4.21}$$

In eq. (4.21), only the importance factor I_T was considered. This decision was taken, since it was discussed in Section 4.3.2 that only the relative trade-offs between the target objectives influenced the final solution. The values of the turbulent Prandtl number were further regularized starting from $Pr_T = 1$, since this corresponds to the most accurate baseline approximation known. The results obtained after performing the present analysis will be discussed in Section 4.4.3.

4.4. Field Inversion Optimization Results

4.4.1. MK Turbulence Model

This Section will outline the results obtained during the hyper-parameter optimization study for the MK turbulence model. These results are extensive in nature, since the cost function for this model (4.9) includes cross-interactions between three hyper-parameters, namely I_U, I_k and I_ϵ. The results obtained after testing the different possible combinations will be outlined in the next subsections.

Hyper-parameter Optimization Study for I_U ($I_k = I_\epsilon = 1$)

One of the first hyper-parameter combinations tested during the optimization of the MK turbulence model consisted in forcing the importance factors I_k and I_ϵ to take unitary values, such that the optimizer assigned equal importance to building corrections for each governing equation (k and ϵ). Based on these considerations, it was possible to perform a direct hyper-parameter optimization study for the values of I_U, which corresponds to the importance factor assigned to the velocity residuals. The results found after applying the Elbow Method to the DNS database assembled can be found in Figure 4.5. The trends presented in this Figure correspond to the final outcome of 160 independent Field Inversion studies. In these results, it can be seen that the DNS cases studied reached an inflection point near $I_U = 100$, and that the velocity residuals obtained were fairly low otherwise. These insights were employed to choose a constant value of $I_U = 100$ to analyze all the DNS flow cases considered. Regarding this point, it is important to note that a different importance factor I_U can also be chosen for every DNS case studied. However, the choice of a constant value for I_U might improve the reproducibility of the $\boldsymbol{\delta}$ corrections obtained.

The source term corrections $[\boldsymbol{\delta}_\mathbf{k}, \boldsymbol{\delta}_\epsilon]$ found during the previous hyper-parameter optimization study for $I_U = 100$ can be found in Figure 4.6. Here, it can be observed that clear trends appear in the results for $\boldsymbol{\delta}_\mathbf{k}$ and $\boldsymbol{\delta}_\epsilon$ across a wide variety of flow cases. Furthermore, it can be noted that the $\boldsymbol{\delta}$ solution curves obtained are roughly anti-symmetric, which indicates that the solver attempted to minimize their magnitude by synchronizing their effect. The presence of anti-symmetric trends in Figure 4.6 can be explained based on the definition of the eddy viscosity for the MK turbulence model: $\mu_t = C_\mu \, f_\mu \, \rho \, k^2/\epsilon$. In this formula, it can be noted that increasing the values of k would produce a direct increment on the values of μ_t, whereas increasing the values of the turbulent dissipation rate (ϵ) would create the opposite effect. The previous trends are further enhanced by the nature of the governing equations for the MK turbulence model, since it can be noted back in eq. (4.6) that increasing the values of the turbulent dissipation rate (ϵ) would also help to reduce the values obtained for the turbulent kinetic energy (k). As a result, it can be noted that the governing equations of the MK turbulence model provide two different mechanisms which help to create the anti-symmetric trends observed in Figure 4.6. Regarding the effect of every individual set of (δ) corrections, the numerical analysis performed revealed that any net positive contribution added to the budget of a governing equation corresponded to a source term, which increased the variable modelled therein (k or ϵ).

Beyond the previous analysis, it can be noted that the peak of the corrections obtained in Figure 4.6 further focus on the buffer layer, which is a zone that traditional RANS turbulence models rarely model accurately. Regarding the magnitude of the corrections found, it can be noted that the case $JFM.CRe^{*}_{\tau}$ presented the largest overall corrections. These results are natural, since it could be noted back in Figure 4.2 that this case presented the largest deviations with respect to the DNS data employed for comparison.

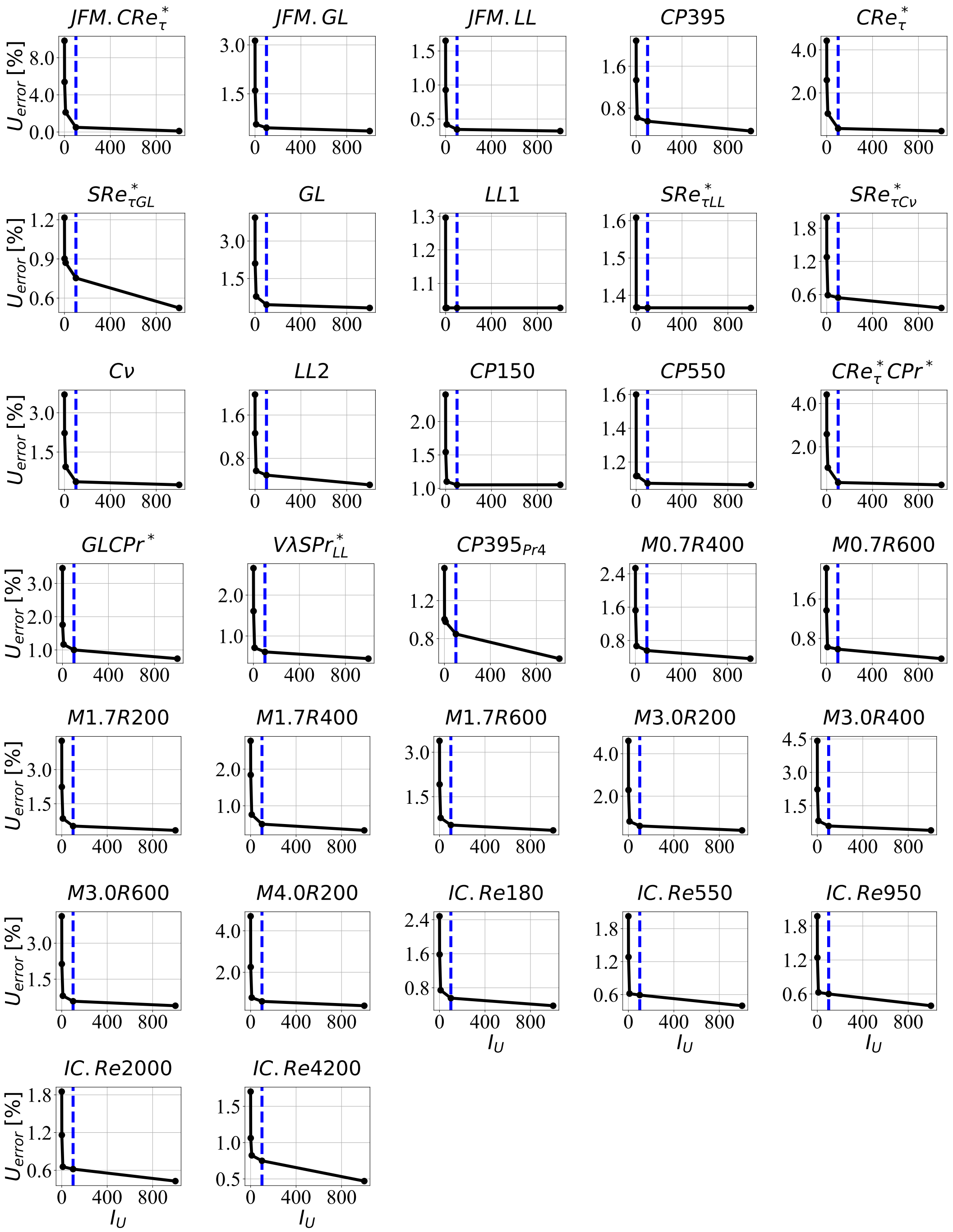


Figure 4.5: Application of the Elbow method to determine the magnitude of I_U during the initial Field Inversion study for the MK turbulence model ($I_k = I_\epsilon = 1$). The blue dashed line marks the position where $I_U = 100$.

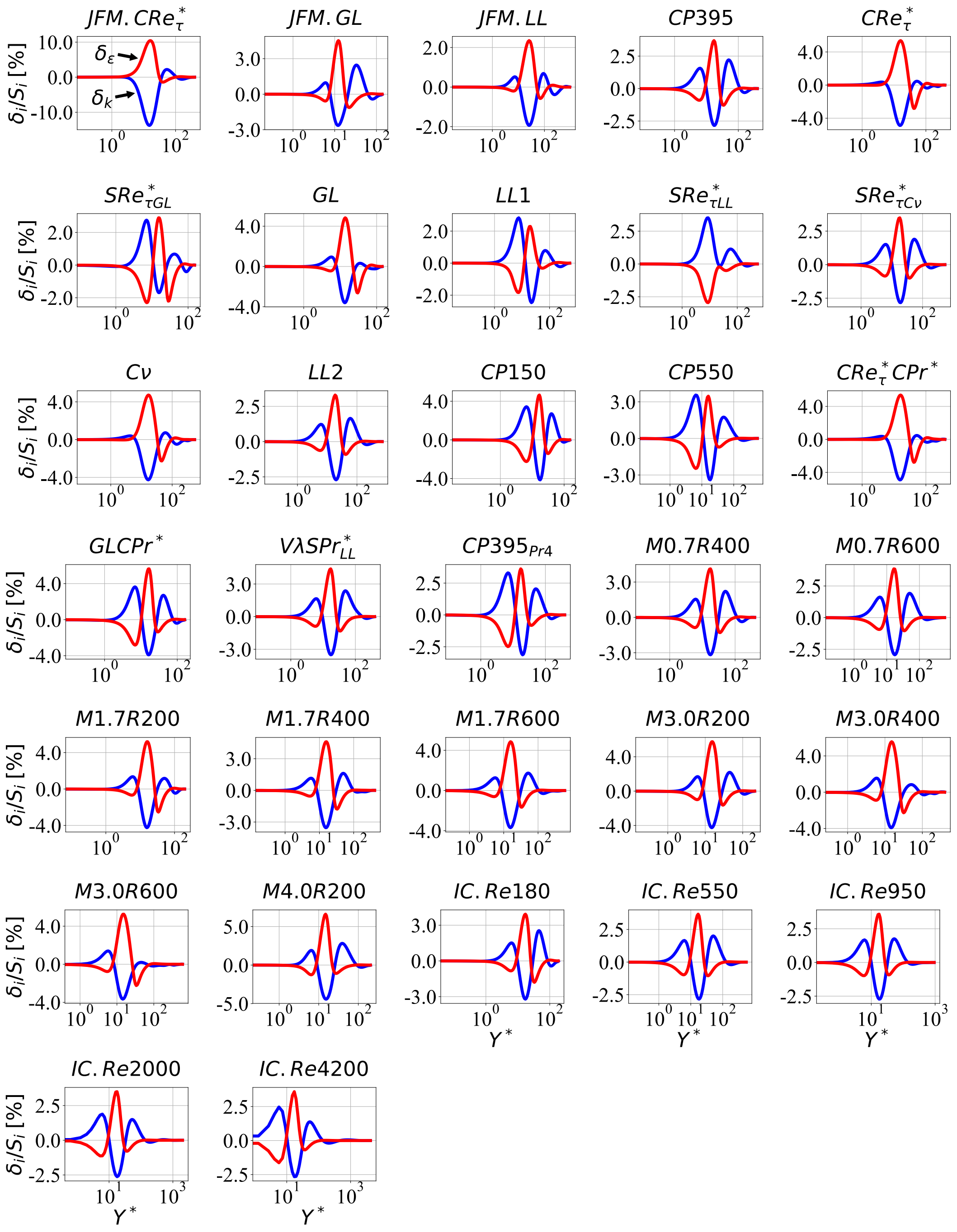


Figure 4.6: Field Inversion corrections ($\boldsymbol{\delta}$) for the MK turbulence model obtained while employing $I_U = 100$ and $I_k = I_\epsilon = 1$. The blue solid lines represent the δ_k corrections, whereas the red lines denote the δ_ϵ corrections.

Interactions between I_k and I_ϵ

During the previous Section, it could be noted that $I_U = 100$ yielded satisfactory results for a wide variety of DNS flow cases. However, it was not possible to draw insights regarding the individual effect of the importance factors I_k and I_ϵ, since they were forced to take unitary values. Due to this reason, the present Section will investigate the effect of these hyper-parameters while imposing $I_U = 100$. The results presented in Figure 4.7 show the evolution of the $[\boldsymbol{\delta}_k, \boldsymbol{\delta}_\epsilon]$ correction curves as I_k and I_ϵ vary from 0.5 to 1.5. The plots are based on the DNS case $JFM.CRe_\tau^*$, which presents the largest corrections found in the dataset. The trends observed in Figure 4.7 indicate that only the magnitude of the $\boldsymbol{\delta}$ corrections obtained varies in each case, since their overall shape is largely preserved. This holds important implications within a Machine Learning context, since it was detected during the present thesis that Neural Networks generally struggle to predict the shape of the solution curves required for a dataset. As a result, it was concluded that the interactions between I_k and I_ϵ had a minor importance during the present study, and their effects were not further studied.

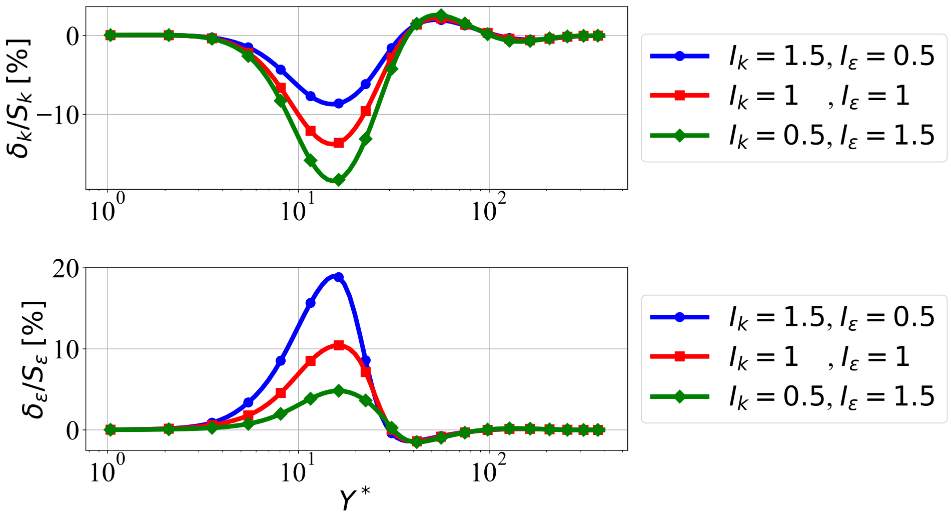


Figure 4.7: Effect of the cross-interactions between I_k and I_ϵ for $I_U = 100$ during the Field Inversion study of the MK turbulence model for the DNS case $JFM.CRe_\tau^*$.

Independent set of δ_k corrections ($I_\epsilon = 0$)

The analysis presented in Section 4.4.1 revealed that the individual values of I_k and I_ϵ only produced secondary effects in the Field Inversion optimizer. Therefore, it was decided to assess the effect of obtaining independent sets of corrections for δ_k and δ_ϵ. The results presented in this Section correspond to the first experiment performed, where only δ_k corrections were obtained for $I_k = 1$ and $I_\epsilon = 0$. Fixing the value of I_k to one corresponded to a good strategy, since it was previously mentioned that only the ratio between the importance factors present in the cost function influences the final distributions (I_U/I_k). The application of the Elbow Method to this problem yielded the results presented in Figure 4.8. Here, it can be seen that $I_U = 100$ again corresponds to a reasonable approximation for the inflection points present in this problem. Regarding the convergence of the system, it is important to note that the velocity residuals (U_{error}) presented in Figure 4.8 for $I_U = 100$ managed to remain below 1%, which is consistent with the trends previously observed in Figure 4.5 while employing two sets of corrections ($\boldsymbol{\delta_k}$,$\boldsymbol{\delta_\epsilon}$). A detailed comparison between the velocity residuals obtained in each case would result irrelevant, since both Field Inversion optimizers managed to replicate the DNS velocity profiles with high accuracy. As a result, only the feasibility of replicating the corrections obtained in each case results important. The sets of δ_k corrections obtained for $I_U = 100$ can be found in Figure 4.9. In this Figure, it can be first observed that the magnitude of the δ_k corrections grew with respect to Figure 4.6 from Section 4.4.1. This effect is natural, since only one set of corrections was employed to drive the simulations.

Beyond the previous considerations, it can be noted that the independent set of δ_k corrections obtained contains smooth transitions and clear trends across a wide variety of DNS cases. As a result, it was determined that employing this type of corrections constituted a good basis for a Machine Learning study, especially since only one parameter needs to be predicted. Employing pairs of corrections ($\boldsymbol{\delta_k}$, $\boldsymbol{\delta_\epsilon}$) can result complex, since the Neural Network system is then forced to produce multichannel outputs. This increases the size of the Neural Network required and the difficulty of assessing the quality of the predictions. Another advantage of employing an independent set of $\boldsymbol{\delta_k}$ corrections is that the system can converge significantly faster, as the optimizer can work with less degrees of freedom.

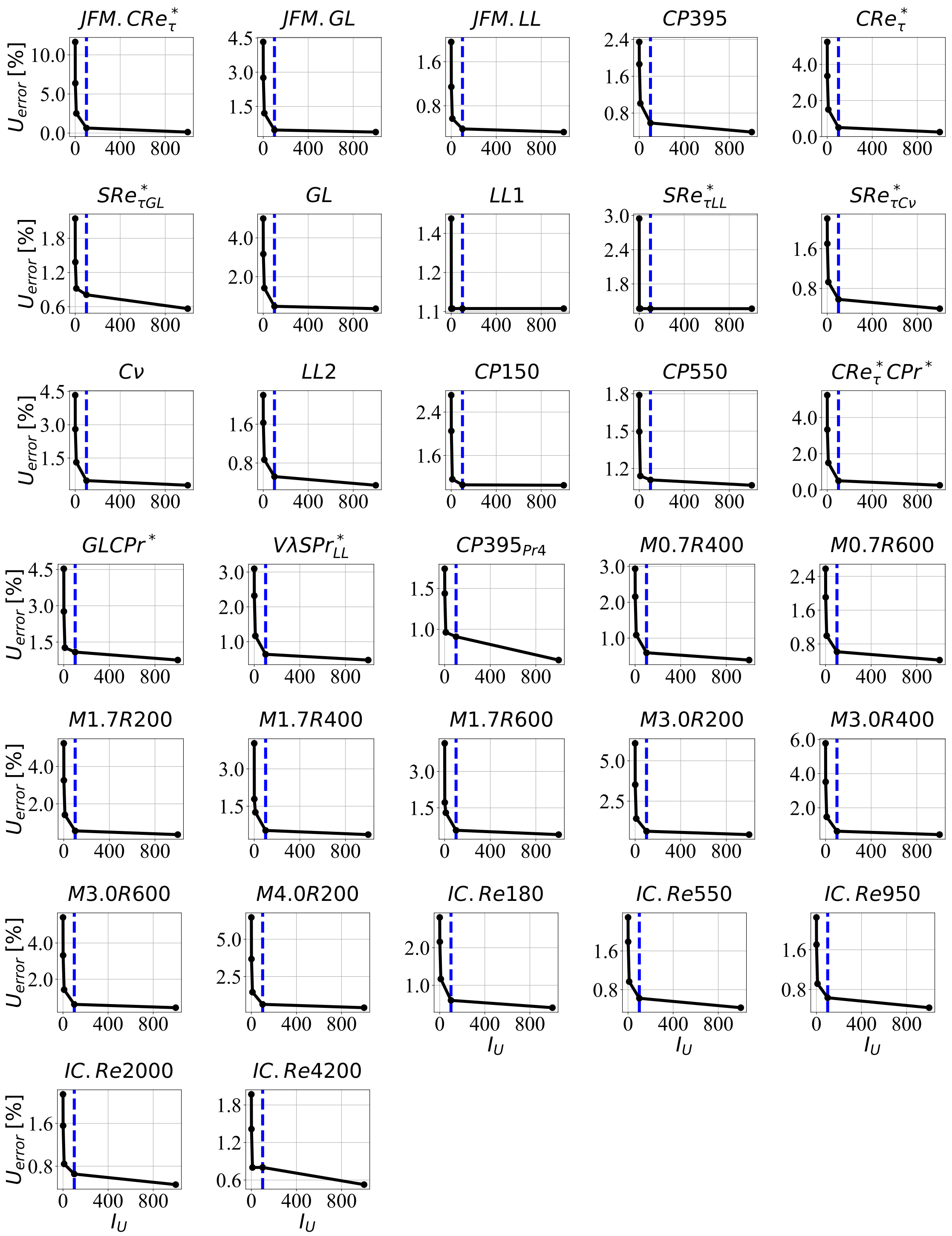


Figure 4.8: Application of the Elbow method to determine the magnitude of I_U during the Field Inversion study of the MK turbulence model while employing only $\boldsymbol{\delta}_\mathbf{k}$ corrections ($I_\epsilon = 0$). The blue dashed line marks the position where $I_U = 100$.

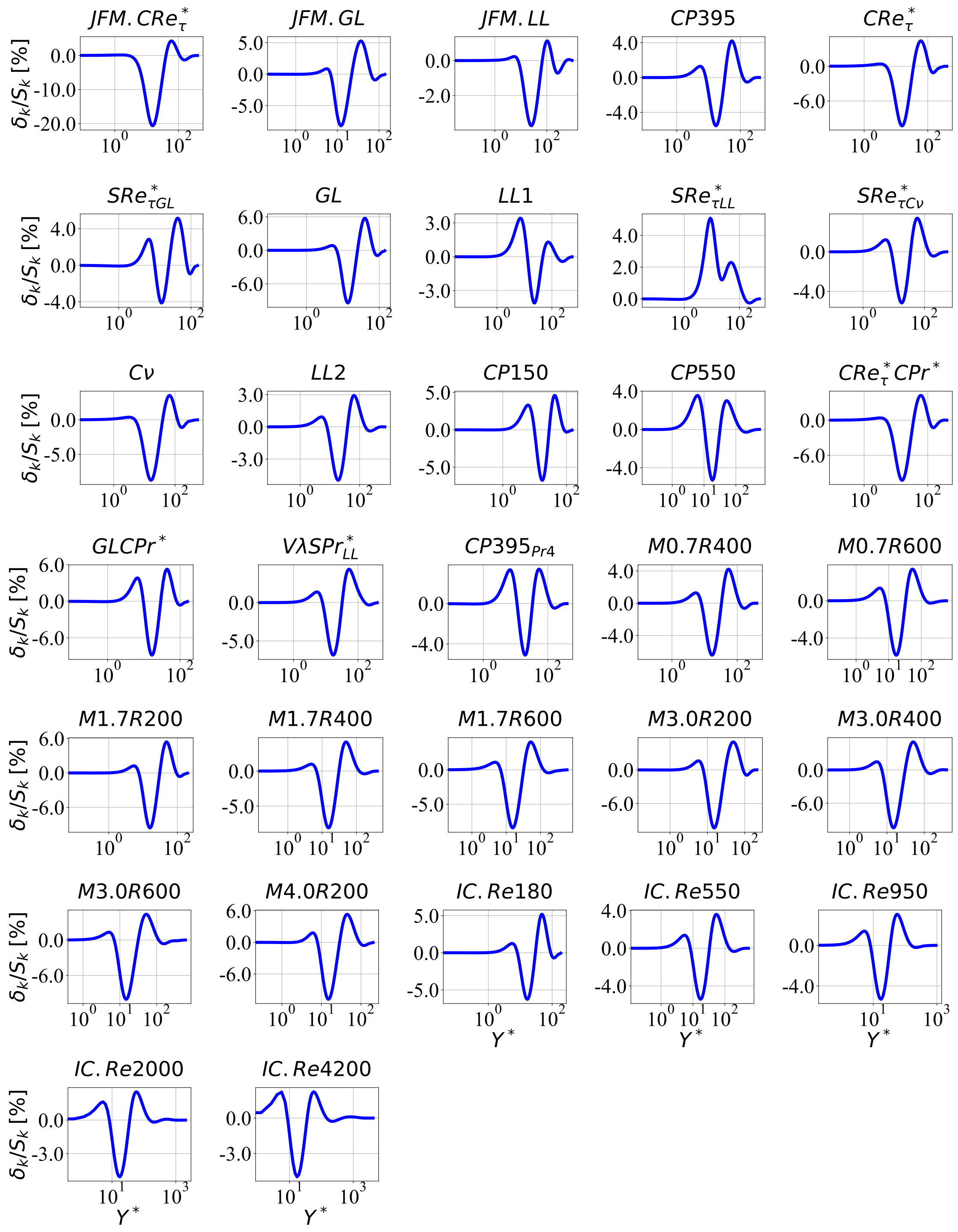


Figure 4.9: Independent set of $\boldsymbol{\delta}_{\mathbf{k}}$ Field Inversion corrections obtained for the MK turbulence model while employing $I_U = 100$, $I_k = 1$ and $I_\epsilon = 0$.

Independent set of δ_ϵ corrections ($I_k = 0$)

The last experiment performed during the Field Inversion study for the MK turbulence model consisted in obtaining an independent set of $\boldsymbol{\delta}_\epsilon$ corrections for $I_\epsilon = 1$ and $I_k = 0$. The results obtained after applying the Elbow Method to this problem can be found in Figure 4.10. Here, it can again be noted that $I_U = 100$ corresponds to a good approximation for the level of corrections required. The Field Inversion corrections obtained for $I_U = 100$ can be found in Figure 4.11. Here, it can be seen the system again obtained sets of smooth corrections for a wide variety of DNS flow cases. However, the set of $\boldsymbol{\delta}_\epsilon$ corrections obtained in this case seem to contain sharper transitions than the previous set of $\boldsymbol{\delta}_\mathbf{k}$ corrections. Since the effect of such transitions in a Machine Learning system is largely unknown, it was decided to build Neural Network architectures based on both set of corrections in order to establish a clear comparison.

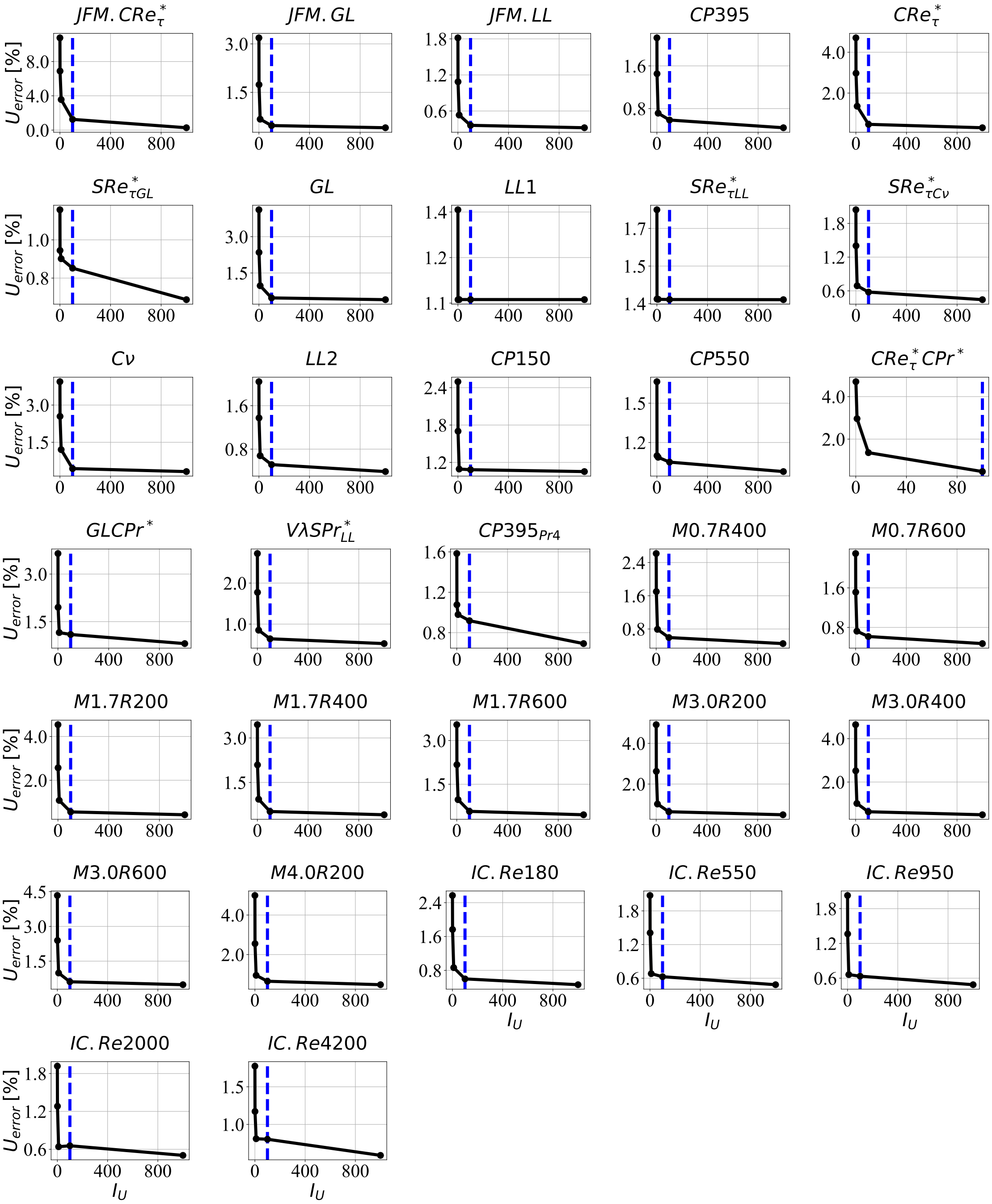


Figure 4.10: Application of the Elbow method to determine the magnitude of I_U during the Field Inversion study of the MK turbulence model while employing only $\boldsymbol{\delta}_\epsilon$ corrections ($I_k = 0$). The blue dashed line marks the position where $I_U = 100$.

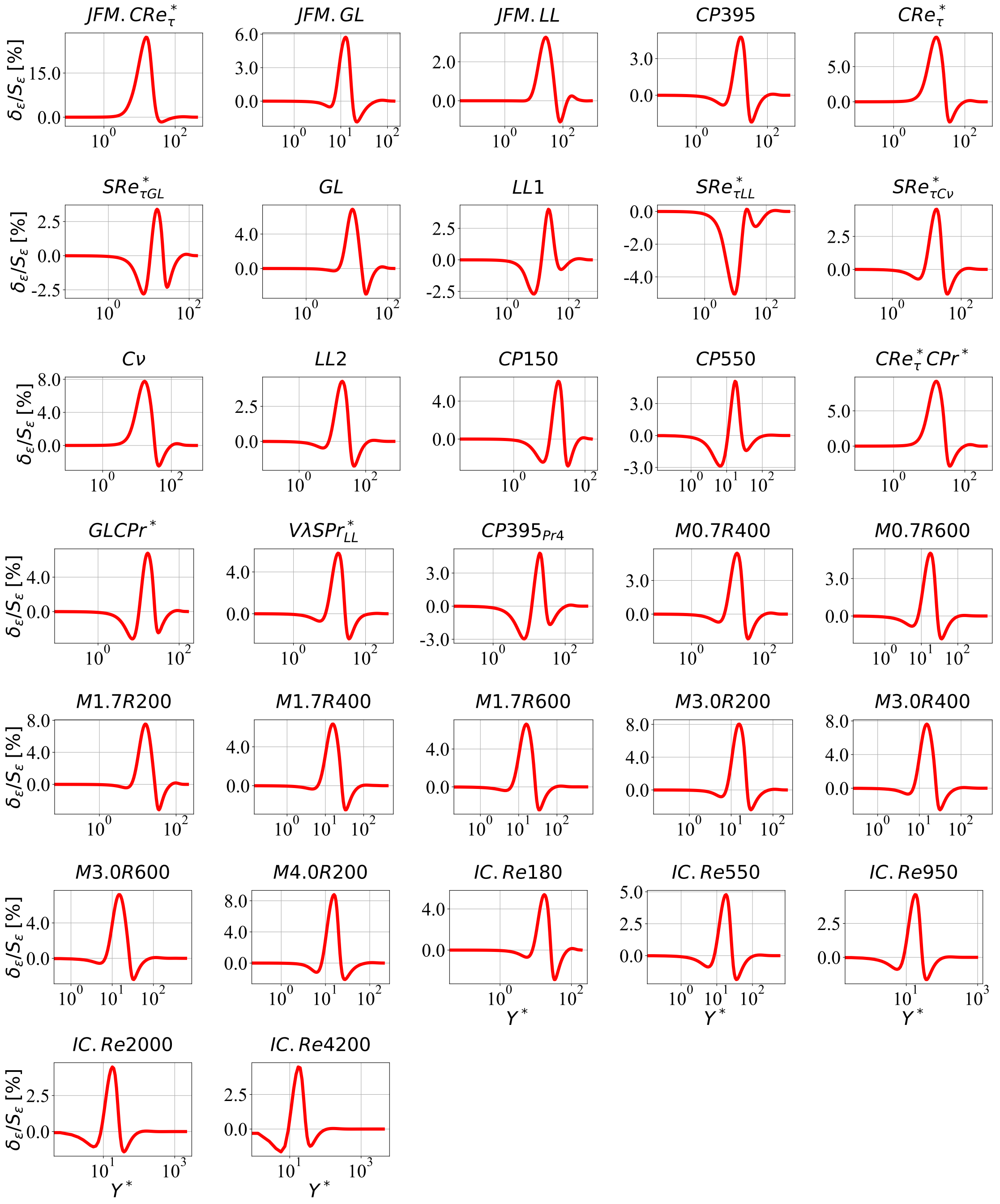


Figure 4.11: Independent set of δ_ϵ Field Inversion corrections obtained for the MK turbulence model while employing $I_U = 100$, $I_\epsilon = 1$ and $I_k = 0$.

4.4.2. Field Inversion Results for the SA turbulence model

The present Section will describe the Field Inversion results obtained for the SA turbulence model. As it was previously mentioned, the Field Inversion model created is only affected by the hyper-parameter ratio I_U/I_{SA}, since individual values of I_U and I_{SA} only scale the learning rates employed. Based on this insight, it was possible to normalize the value of I_{SA} to one without compromising the results. The application of the Elbow method to determine the best overall values of I_U can be found in Figure 4.12 as before. Unfortunately, here it can be noted that the error curves presented do not show a clear inflection point in many cases. However, it can still be noted that $100 < I_U < 500$ corresponds to a reasonable approximation of the inflection point present in various flow cases. The results obtained for $I_U = \{100, 500\}$ can be found in Figure 4.13. In this chart, it can be noted that every DNS case requires drastically different corrections, and that no clear patterns appear. Furthermore, it can be noted that the corrections obtained for $I_U = 500$ are drastically larger, whereas the velocity residuals of the Elbow Method show that little improvements are achieved. For example, the corrections for the case $IC.Re4200$ grew 12.6% after changing I_U from 100 to 500, yet the velocity residuals perceived variations lower than 0.1%. As a result, it was concluded that the Spalart-Allmaras model cannot be further tuned within the context of Machine Learning for variable-property flows without creating highly over-fitted corrections. The elaboration of Neural Network architectures for these Field Inversion corrections was thus discarded.

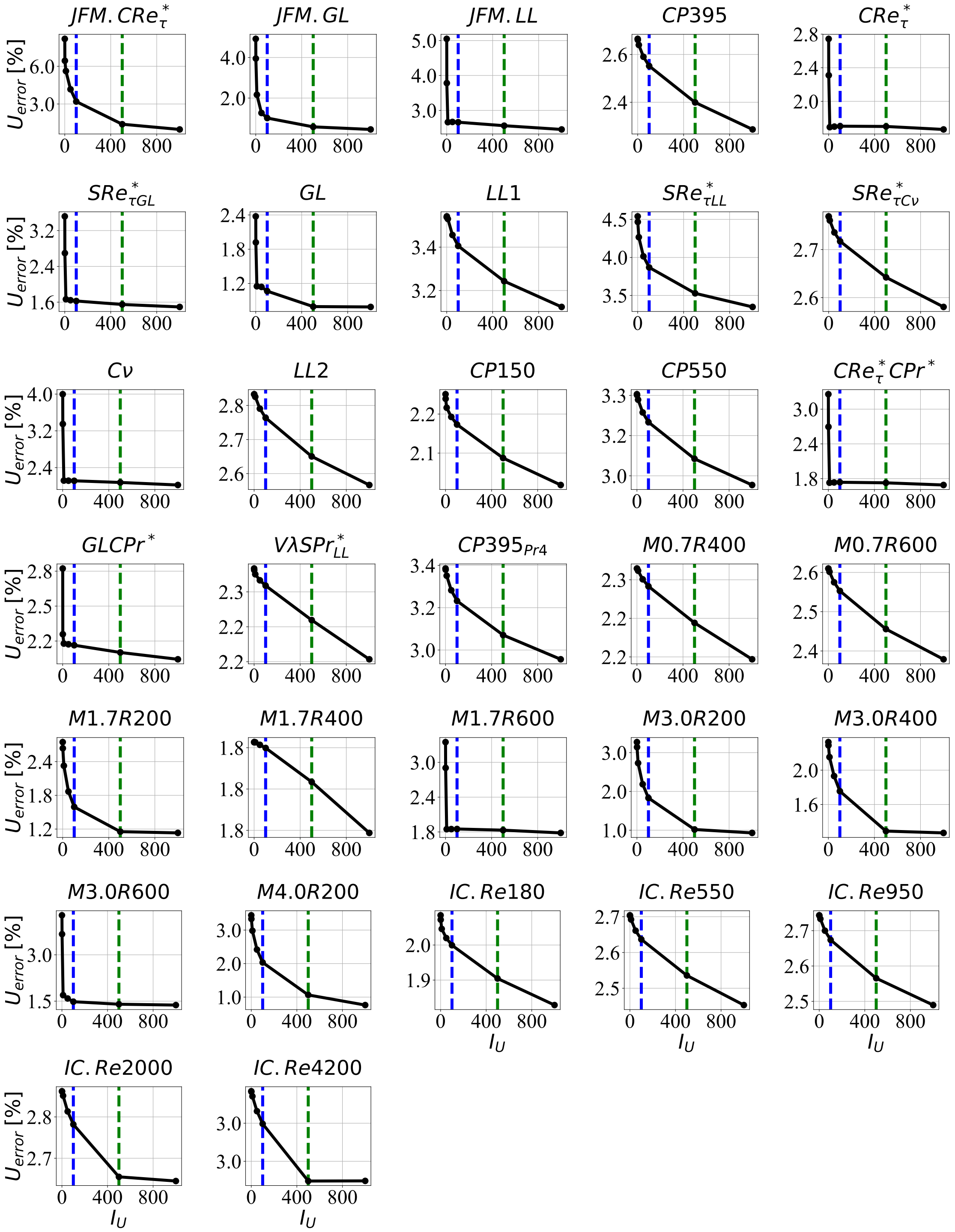


Figure 4.12: Application of the Elbow Method to determine the magnitude of I_U during the Field Inversion study of the SA turbulence model [8]. The blue dashed line marks the position where $I_U = 100$, whereas the green dashed line represents $I_U = 500$.

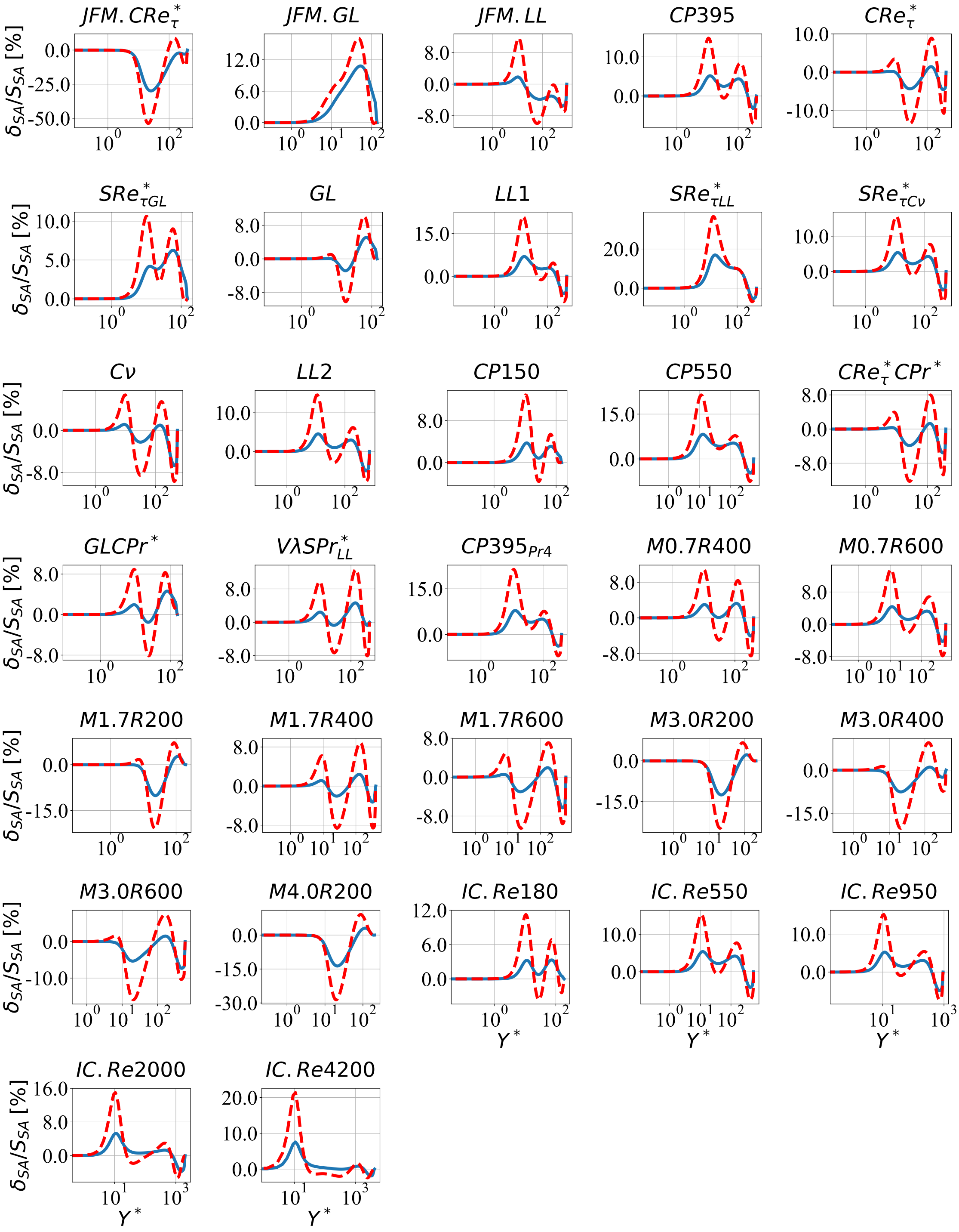


Figure 4.13: Field Inversion corrections (δ_{SA}) obtained for the SA turbulence model while employing $I_U = 100$ (blue solid lines) and $I_U = 500$ (red dashed lines).

4.4.3. Field Inversion Results for the Turbulent Prandtl Number

This Section will present the results obtained during the Field Inversion study of the turbulent Prandtl number (Pr_T). The velocity curves employed to perform this analysis (u_{opt}) came from the Field Inversion results obtained in Section 4.4.1 for the MK turbulence model while employing $I_U = 100$ and $I_k = I_\epsilon = 1$. Thanks to these velocity profiles, it was possible to optimize the energy equation as an stand-alone entity and to accelerate the convergence process. The Elbow method was applied to this problem in order to assess the influence of the importance factor I_T present in eq. (4.21). The results obtained during the hyper-parameter optimization study can be found in Figure 4.14. Here, it can be noted that $I_T = 100$ again corresponds to a reasonable approximation for the present dataset. The profiles obtained for Pr_T therein can be found in Figure 4.15. One of the first conclusions which can be drawn from these curves is that the optimizer managed to obtain accurate results after performing minimal alterations to the values of Pr_T. As a result, $Pr_T \approx 1$ effectively constitutes a good approximation for this dataset. However, it can be noted that the trends obtained by the Field optimizer do not contain clear patterns, which complicates the creation of advanced Machine Learning systems to improve the predictions of Pr_T.

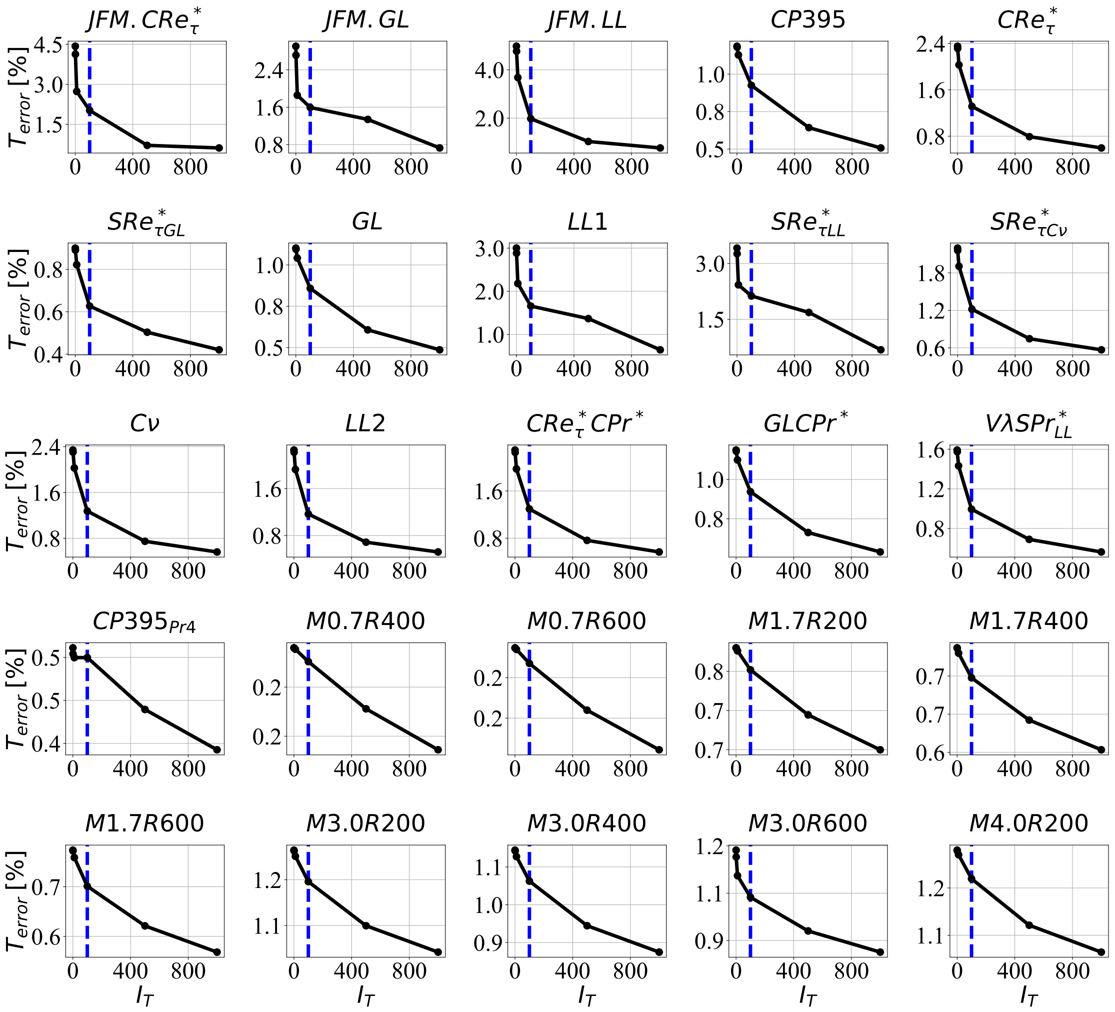


Figure 4.14: Application of the Elbow Method to determine the overall magnitude of I_T during the Field Inversion study of the Turbulent Prandtl Number (Pr_T). The blue dashed line marks the position where $I_U = 100$.

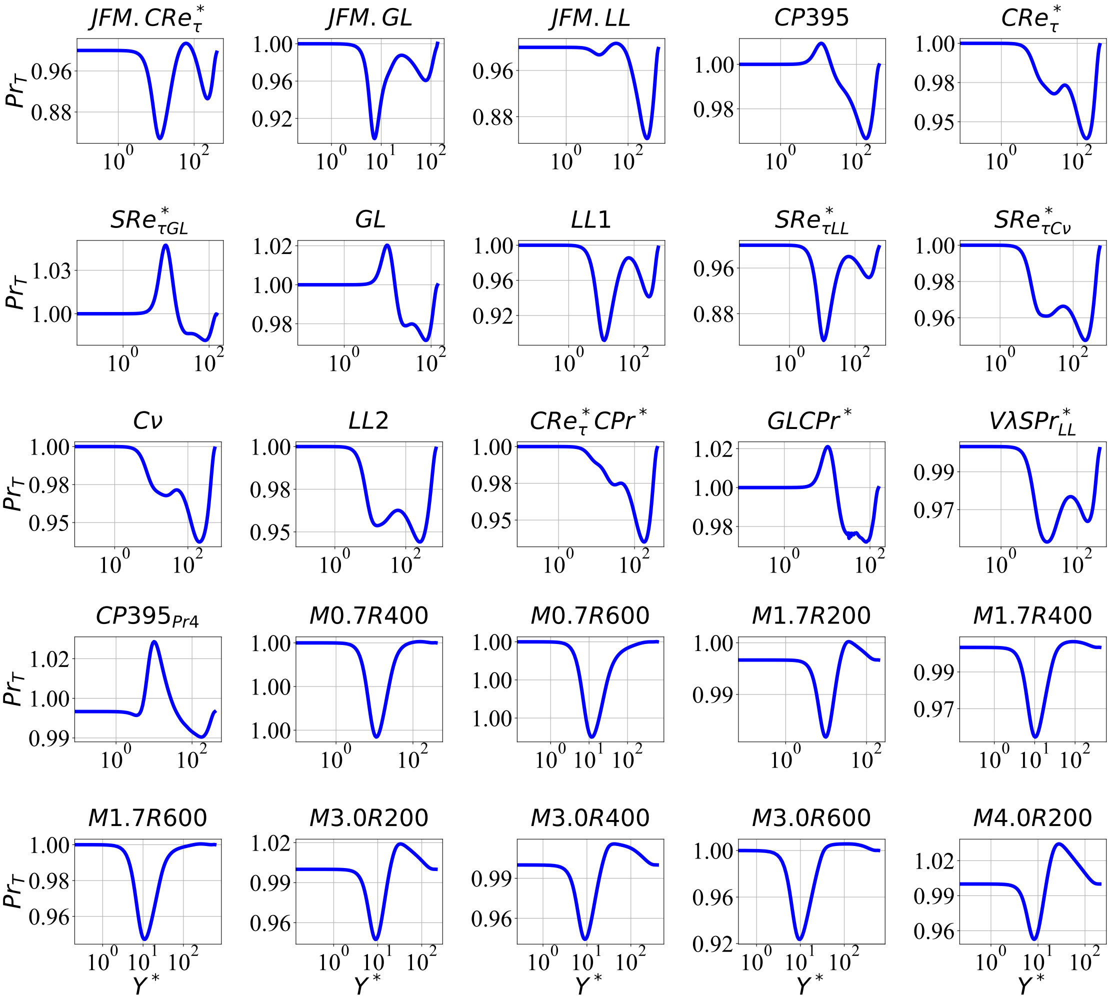


Figure 4.15: Field Inversion results obtained for the Turbulent Prandtl Number (Pr_T) while employing $I_U = 100$.

Regarding the nature of the trends presented in Figure 4.15, it must be noted that the results obtained by the Field Inversion optimizer did not approach the true DNS distributions obtained by Patel (2017) [9]. A direct comparison between both solution curves can be found in Figure 4.16. Here, it can be seen that the DNS values for Pr_T start from values close to one, yet they fall towards the end of the log-layer to values as low as 0.4. The Field Inversion optimizer had little incentive to recover such changes however, since the temperature profiles obtained presented errors below 1% while employing turbulent Prandtl number (Pr_T) corrections up to 8%. As a result, the Field Inversion optimizer would have regarded the DNS distribution for Pr_T at the channel center as a highly over-fitted solution. Despite the previous fact, it can be noted that the Field Inversion optimizer matched the DNS data for Pr_T at the buffer layer in several cases. This information was recovered by the Field Inversion optimizer, since the buffer layer corresponds to the zone of largest turbulent activity. Beyond these considerations, it was decided not to build a Machine Learning system for the Field Inversion results obtained for the turbulent Prandtl number (Pr_T), since the baseline approximation established yielded highly accurate results and the trends obtained by the optimizer resulted unclear.

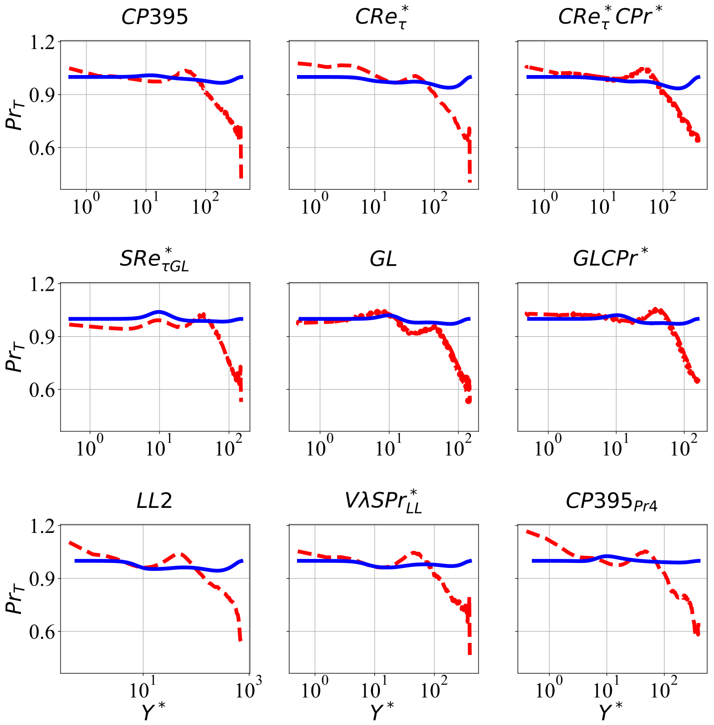


Figure 4.16: Comparison between the DNS data analyzed by Patel (2017) [9] (red dashed lines) and the results obtained by the Field Inversion optimizer (blue solid lines) for the turbulent Prandtl number Pr_T in various flow cases.

4.5. Machine Learning Systems

This Section will describe the Machine Learning systems built to predict the best sets of Field Inversion corrections obtained for the MK turbulence model in Section 4.4.1. Several sets of Field Inversion corrections were discarded, since no clear patterns were observed in the results obtained for the Spalart-Allmaras turbulence model nor the turbulent Prandtl number (Pr_T).

However, before addressing the final Neural Network systems created, the next Section will introduce the formula of an intelligent relaxation factor, which was derived during the present thesis work. Thanks to this factor, it was possible to mitigate any spurious sets of Machine Learning corrections predicted by the Neural Networks created. In Section 4.5.1, it will be further proven that the formula created is able to recover valuable information from Machine Learning predictions that would have resulted unstable otherwise.

4.5.1. Derivation of an Intelligent Relaxation Factor

This Section will present the mathematical derivation of an intelligent relaxation factor created to improve the predictions made by Neural Network architectures. The introduction of this feature into the FIML methodology constitutes a substantial innovation in this field. The presentation of the current methodology will be split into two sub-sections. In the first Section, the problem encountered will be presented, along with the results obtained thanks to the solution developed. Then, in the second Section, the mathematical derivation of the intelligent relaxation factor will be explained in detail.

Solution Overview

In order to understand the need for an intelligent relaxation factor, Figure 4.17 presents an example of the $\boldsymbol{\delta}_{\mathbf{ML}}$ predictions made by a (fictitious) Neural Network vs. the ground-truth Field Inversion corrections ($\boldsymbol{\delta}^{*}$) obtained for the DNS case $JFM.CRe^{*}_{\tau}$. Here it can be seen that the corrections proposed by the Machine Learning system ($\boldsymbol{\delta}_{\mathbf{ML}}$) at the channel center fell into the red zone defined back in

Figure 4.3, which constitutes a region where RANS turbulence models cannot balance any external source terms. As a result, the oscillations present in the term δ_{ML} would make the RANS equations unstable, despite the accurate predictions obtained for $Y^* < 100$.

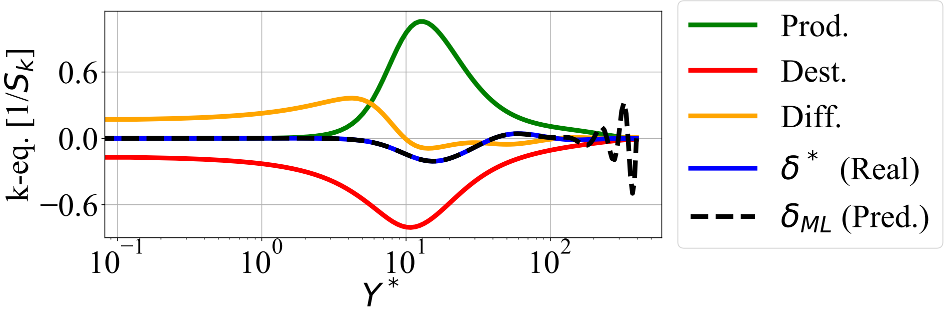


Figure 4.17: Turbulence budget for the optimized k-equation of the MK turbulence model after applying the independent set of $\boldsymbol{\delta_k}$ corrections obtained in Section 4.4.1 for the DNS case $JFM.CRe_\tau^*$. The fictitious Machine Learning predictions contain the perturbation term: $\Delta = 0.6\ y^3\ sin\,(8\pi y)$.

The previous problem was solved by introducing an intelligent relaxation factor (α), which is defined as the fraction of the original Machine Learning corrections (δ_{ML}) to keep. The remaining part of the corrections is deleted based on their perceived incompatibility with the RANS equations. The results of the application of this principle to the exercise proposed can be found in Figure 4.18. Here, it can be seen that the intelligent relaxation factor (α) deleted the 40% worst part of the initial corrections ($\boldsymbol{\delta_{ML}}$) based on the budget for the destruction term of the k-equation. The final distribution obtained effectively resembles the ground-truth labels ($\boldsymbol{\delta}^*$), which were hidden from the system. The code presented in Appendix B.1 was able to perform an intelligent selection of the best $\boldsymbol{\delta_{ML}}$ corrections to keep in less than 1 millisecond for the present example.

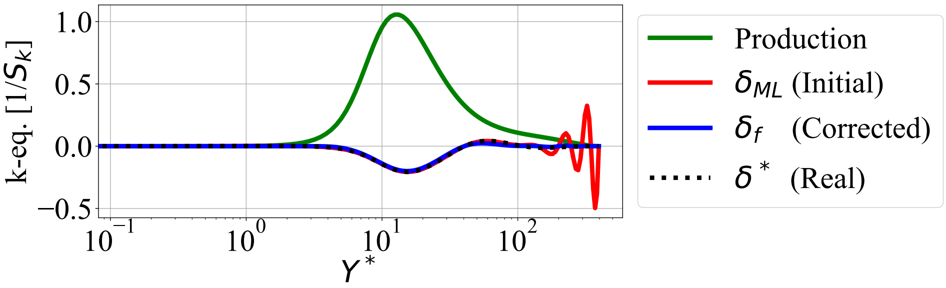


Figure 4.18: Results obtained after applying the intelligent relaxation factor methodology to the exercise proposed in Figure 4.17 using $\alpha = 0.6$.

Mathematical Derivation

The mathematical derivation behind the intelligent relaxation factor (α) methodology starts by noting that the magnitude of the final corrections that will be applied to the RANS model ($\boldsymbol{\delta_f}$) only correspond to an (α) fraction of the original Machine Learning corrections ($\boldsymbol{\delta_{ML}}$). This relation can be stated as:

$$\|\delta_f\| = \alpha\ \|\delta_{ML}\|. \tag{4.22}$$

Or alternatively,

$$\mathcal{J}_\delta = \sum_{i=1}^{N} \delta_{f,i}^2 = \sum_{i=1}^{N} \left(\alpha\ \delta_{ML,i}\right)^2\ \ (= constant). \tag{4.23}$$

In order to assess the true compatibility of the $\boldsymbol{\delta_f}$ corrections with a RANS turbulence model, it was decided to express these corrections as $\boldsymbol{\beta}$ multipliers from the production terms present in the respective governing equations ($\mathbf{P}$):

$$\boldsymbol{\delta}_{\mathbf{f}} = \boldsymbol{\beta}\ \mathbf{P}. \tag{4.24}$$

Employing the production terms ($\mathbf{P}$) as the baseline reference for the current methodology was deemed appropriate, since it was detected during a preliminary study that the Field Inversion corrections obtained for the MK turbulence model were well-aligned with these terms. Introducing eq. (4.24) into eq. (4.23) further yields that:

$$\mathcal{J}_\delta = \sum_{i=1}^{N} \left(P_i\beta_i\right)^2 = \sum_{i=1}^{N} \left(\alpha\ \delta_{ML,i}\right)^2\ \ (= constant). \tag{4.25}$$

At this point, it must be noted that the spurious Machine Learning corrections ($\boldsymbol{\delta}_{\mathbf{ML}}$) present in Figure 4.17 would create large spikes in the $\boldsymbol{\beta}$ multipliers defined by eq. (4.24). As a result, the introduction of a regularization hyper-parameter λ for the $\boldsymbol{\beta}$ multipliers would immediately mitigate their presence. The cost function associated to this problem is:

$$\begin{aligned} \mathcal{J}_\beta &= \frac{1}{2}\sum_{i=1}^{N} \left(\delta_{f,i} - \delta_{ML,i}\right)^2 + \lambda\beta_i^2 \\ &= \frac{1}{2}\sum_{i=1}^{N} \left(P_i\beta_i - \delta_{ML,i}\right)^2 + \lambda\beta_i^2. \end{aligned} \tag{4.26}$$

Eq. (4.26) states that the final $\boldsymbol{\beta}$ multipliers must produce the greatest degree of similarity between $\boldsymbol{\delta}_{\mathbf{f}}$ and $\boldsymbol{\delta}_{\mathbf{ML}}$, while minimizing the magnitude of $||\boldsymbol{\beta}||^2$ according to a regularization hyper-parameter λ. In order to minimize the cost function defined in eq. (4.26), its Jacobian can be forced to form a null vector:

$$\nabla_\beta \mathcal{J}_\beta = \mathbf{0}. \tag{4.27}$$

Replacing eq. (4.26) into the previous condition yields the following vector equation:

$$\mathbf{P}\left(\mathbf{P}\boldsymbol{\beta} - \boldsymbol{\delta}_{\mathbf{ML}}\right) + \lambda\boldsymbol{\beta} = 0. \tag{4.28}$$

Re-arranging the terms of eq. (4.28) further reveals that:

$$\boldsymbol{\beta} = \frac{\boldsymbol{\delta}_{\mathbf{ML}}\ \mathbf{P}}{\lambda + \mathbf{P}^2}. \tag{4.29}$$

Eq. (4.29) is intended to be evaluated element-wise, since vectors cannot be divided. Replacing eq. (4.29) back into eq. (4.25) yields a direct residual equation for λ:

$$\mathcal{R}_\lambda = \sum_{i=1}^{N} \left(\delta_{ML,i}\ \frac{P_i^2}{\lambda + P_i^2}\right)^2 - \sum_{i=1}^{N} \left(\alpha\ \delta_{ML,i}\right)^2 = 0. \tag{4.30}$$

Since eq. (4.30) only contains one unknown (λ), a simple root-finding library can be used to solve the previous optimization problem. In the code presented in Appendix B.1, a Newton-Raphson solver was employed to solve eq. (4.30). While the convergence of this method is not guaranteed, the solver created has proven to be stable in every case tested. For future reference, the gradient of the previous residual equation ($\mathcal{R}_\lambda$) is given by the following formula:

$$\nabla_\lambda \mathcal{R}_\lambda = \sum_{i=1}^{N} -2\frac{\left(\delta_{ML,i}P_i^2\right)^2}{\left(\lambda + P_i^2\right)^3}. \tag{4.31}$$

After obtaining the regularization hyper-parameter λ, the final Machine Learning corrections ($\boldsymbol{\delta}_{\mathbf{f}}$) are given by:

$$\boldsymbol{\delta}_{\mathbf{f}} = \frac{\boldsymbol{\delta}_{\mathbf{ML}}\ \mathbf{P}^2}{\lambda + \mathbf{P}^2}. \tag{4.32}$$

Eqs. (4.30-4.32) constitute the only required components to implement the present methodology in a computer environment. As it can be noted, the method presented was greatly simplified by the synthesis of a direct scalar equation for λ. No references are provided in this Section, since the equations presented were derived based on the application of the principles described in Chapter 2.

4.5.2. Neural Network Architectures

The present Section will describe the final Neural Network architectures created for the sets of Field Inversion corrections chosen for the MK turbulence model. First, in Section 4.5.2, a Neural Network system will be built for the independent set of $\boldsymbol{\delta}_{\mathbf{k}}$ corrections obtained back in Section 4.4.1. Then, based on this analysis, the methodology created will be modified in Section 4.5.2 such as to build a Machine Learning system for the second set of independent $\boldsymbol{\delta}_{\epsilon}$ corrections obtained for the MK turbulence model. The global results obtained will be finally discussed in Section 4.6, along with future perspectives.

MK Turbulence Model - Independent Set of $\boldsymbol{\delta}_{\mathbf{k}}$ Corrections

The Machine Learning system created to predict the independent set of $\boldsymbol{\delta}_{\mathbf{k}}$ corrections for the MK turbulence model was inspired by the logarithmic Neural Network architecture employed in Chapter 3 to infer the existence of the Reynolds number using the Drag Coefficient of a cylinder. This architecture was chosen, since it is capable of performing an automatic study of the best parameters groups to drive a Neural Network. The overall framework developed for the FIML (Field Inversion Machine Learning) system can be found depicted in Figure 4.19. Here, it can be seen that the final Neural Network predictions are completely independent from the DNS data employed to train the model. A triangular feedback loop was established to create the required features given to the model, since the Machine Learning system was trained using the true DNS molecular properties (μ, ρ, λ). While this loop increases the complexity of the system, employing the molecular properties obtained from the uncorrected RANS model would have created an undesired correlation with respect to the temperature-dependency laws listed in Tables 4.1 and 4.2. As a result, the introduction of a feedback loop allowed the Machine Learning system to focus on modelling the flow physics exclusively.

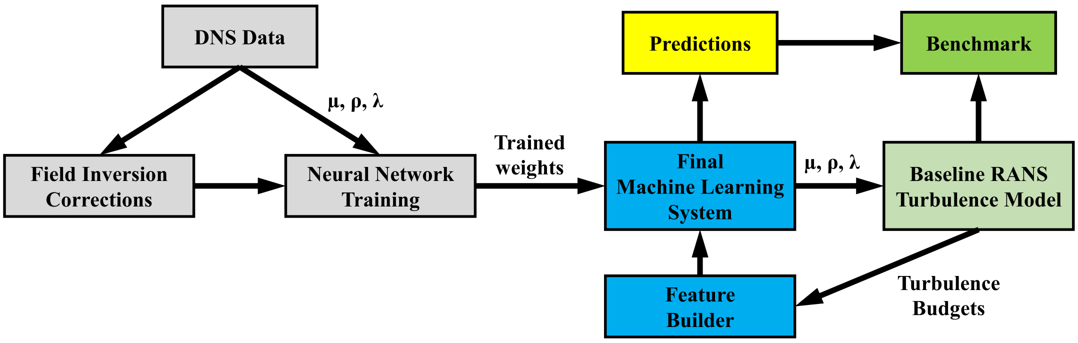


Figure 4.19: Final framework established for the Field Inversion Machine Learning methodology (FIML).

Due to the small nature of the present dataset, the K-fold validation method was employed to perform a scientific study of the variations encountered in the Machine Learning system. The datasets employed for testing during the implementation of the K-fold validation method can be found listed in Table 4.4, which contains a total of 10 different scenarios. The first trial run (K-1) corresponds to the top-5 cases with the highest velocity residuals identified during the initial RANS turbulence modelling benchmark presented in Figure 4.2. The 9 remaining test sets were generated by randomly selecting cases from the different data sources available.

K-1	$JFM.CRe_{\tau}^{*}$	Cv	$CRe_{\tau}^{*}CPr^{*}$	$M3.0R200$	$M4.0R200$	
K-2	$JFM.CRe_{\tau}^{*}$	$SRe_{\tau LL}^{*}$	$GLCPr^{*}$	$M1.7R200$	$M4.0R200$	$IC.Re950$
K-3	$JFM.CRe_{\tau}^{*}$	$JFM.GL$	$JFM.LL$	$M3.0R600$	$M4.0R200$	$IC.Re550$
K-4	$JFM.CRe_{\tau}^{*}$	$LL2$	$CP150$	$V\lambda SPr_{LL}^{*}$	$M3.0R200$	$IC.Re950$
K-5	$JFM.LL$	$LL1$	Cv	$CP395_{Pr4}$	$M0.7R400$	$IC.Re550$
K-6	$JFM.GL$	Cv	$CRe_{\tau}^{*}CPr^{*}$	$M0.7R400$	$M4.0R200$	$IC.Re2000$
K-7	$JFM.CRe_{\tau}^{*}$	GL	$CRe_{\tau}^{*}CPr^{*}$	$GLCPr^{*}$	$M1.7R200$	$IC.Re550$
K-8	$JFM.GL$	$SRe_{\tau GL}^{*}$	$SRe_{\tau LL}^{*}$	$V\lambda SPr_{LL}^{*}$	$M1.7R400$	$IC.Re4200$
K-9	$JFM.LL$	$CP150$	$GLCPr^{*}$	$CP395_{Pr4}$	$M1.7R600$	$IC.Re180$
K-10	$JFM.CRe_{\tau}^{*}$	$SRe_{\tau Cv}^{*}$	$CRe_{\tau}^{*}CPr^{*}$	$CP395_{Pr4}$	$M0.7R400$	$IC.Re950$

Table 4.4: Test cases considered during the implementation of the K-fold validation method for the study of Neural Network architectures.

One of the greatest difficulties encountered during the present context was the lack of clear cross-validation (CV) sets which could be defined for every K-fold combination. Since all the DNS cases available correspond to different scenarios, it was confirmed that the conclusions drawn from a given CV set may not yield relevant remarks regarding the final test set. Due to this reason, the study was performed by employing only one of the K-fold validation sets to select a suitable Neural Network architecture, and subsequently training the chosen Neural Network system under different K-fold combinations in order to assess its robustness. As it will be seen in Section 4.5.2, the robustness of a Neural Network architecture is not guaranteed, since the system must actively guess whether a flow case presents large ground-truth corrections ($\boldsymbol{\delta}_{\mathbf{k}}$) or not. As a result, the system can eventually grow unstable depending on the training set.

The first K-fold validation case was finally employed to select a suitable Neural Network architecture. The stopping criterion employed to choose the model was based on the final test error perceived. The iterations performed quickly revealed that small-sized networks presented the lowest risk of over-fitting, and that employing more than two hidden layers of Hyperbolic Tangent neurons provided little benefits. At a certain point however, reducing the size of the Neural Network created large spikes in the test-set errors, which can be interpreted as the emergence of under-fitted models without enough parameters to reasonably approximate the functional relation $\mathbf{Y} = f(\mathbf{X})$ between the stack of input features ($\mathbf{X}$) and the output labels ($\mathbf{Y}$). The final Neural Network architecture chosen can be found depicted in Figure 4.20. Here, it can be seen that 14 hidden features were supplied to the system, and that only 3 parameter groups were required to yield accurate predictions. The subsequent hidden layers of Hyperbolic Tangent Neurons contained only 5 and 2 units respectively. The turbulence budget terms listed in the feature stack from Figure 4.20 were obtained by pre-computing the uncorrected results of the RANS equations. The variables M_k and M_ϵ present in Figure 4.20 correspond to the sub-scales synthesized for the values of k and ϵ respectively. From a dimensional analysis, it can be proven that these factors are given by the following relations:

$$M_\epsilon = \frac{S_k}{\rho_w}, \tag{4.33}$$

$$M_k = \frac{\rho_w M_\epsilon^2}{S_\epsilon}. \tag{4.34}$$

It can be noted that the magnitude scales given by eqs. (4.33-4.34) can be ultimately changed by the Neural Network architecture created, since the input features contain all the required information to build transformations in log-space. A similar argument can be made regarding the flow variables Y^* and Re_τ^*. Therefore, the pre-processed input features given to the system are only intended to facilitate the convergence process. The overall similarity in the number of neurons between the assembled Neural Network and the Deep Learning system presented in Section 3.4 for the drag coefficient of a cylinder can be regarded as a coincidence, since the ML system created in this Section was obtained through a rigorous process of model evaluation. The final results obtained after performing the K-fold validation process can be found reported in Table 4.5. Here, a side-by-side comparison is offered between the training and test errors obtained for the system. A selection of the ML results obtained for the K-fold validation trials with the highest and the lowest test errors can be found in Figure 4.21. The results presented in this Figure employ an intelligent relaxation factor of $\alpha = 0.5$, and correspond to a sub-set of the extended results presented in Appendix B. While the relaxation factor $\alpha = 0.5$ was prescribed based on general experience, it can be noted that the intelligent relaxation factor methodology managed to eliminate significant spurious corrections present in the DNS cases $JFM.CRe_\tau^*$ and $JFM.LL$ from the K-fold run (K-3), or the spike present at the channel center in the DNS case $JFM.GL$ from the K-fold run (K-6). As a result, employing a more aggressive relaxation factor ($\alpha \to 1$) is not recommended. Beyond the previous analysis, the case $JFM.LL$ corresponds to the only case where the Neural Network did not manage to identify the right sign (or direction) for the corrections to apply. Despite these difficulties, it can be noted that the predictions for the velocity profile remained stable, and that relatively low errors were introduced into the system. Finally, it must be mentioned that the corrections found by the Neural Network architecture for the incompressible DNS flow cases achieved high levels of accuracy for every test set defined. This can be partially seen in the results presented in Figure 4.21 for the DNS cases $IC.Re950$ and $IC.Re2000$. As a result, the present methodology can also be considered as a valid alternative during the Machine Learning study of incompressible DNS flow cases.

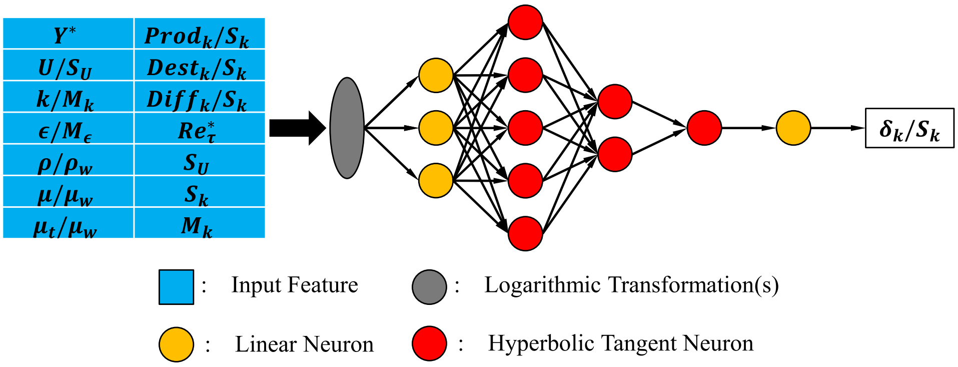


Figure 4.20: Neural Network architecture created to predict the Field Inversion corrections ($\boldsymbol{\delta}_\mathbf{k}$) required by the MK turbulence model.

K-fold Run	$\mathcal{J}_{\mathbf{train}}$	$\mathcal{J}_{\mathbf{test}}$	Error Ranking
K-1	1.60	47.3	2
K-2	1.00	11.1	7
K-3	1.59	182.1	1
K-4	1.60	38.8	4
K-5	1.29	5.8	9
K-6	2.04	2.0	10
K-7	1.16	35.6	5
K-8	1.55	12.7	6
K-9	1.22	9.5	8
K-10	1.58	39.0	3

Table 4.5: Normalized cost function values obtained for the training and test sets defined in every K-fold validation run during the prediction of $\boldsymbol{\delta}_\mathbf{k}$ for the MK turbulence model. Reference scale: Minimum value encountered.

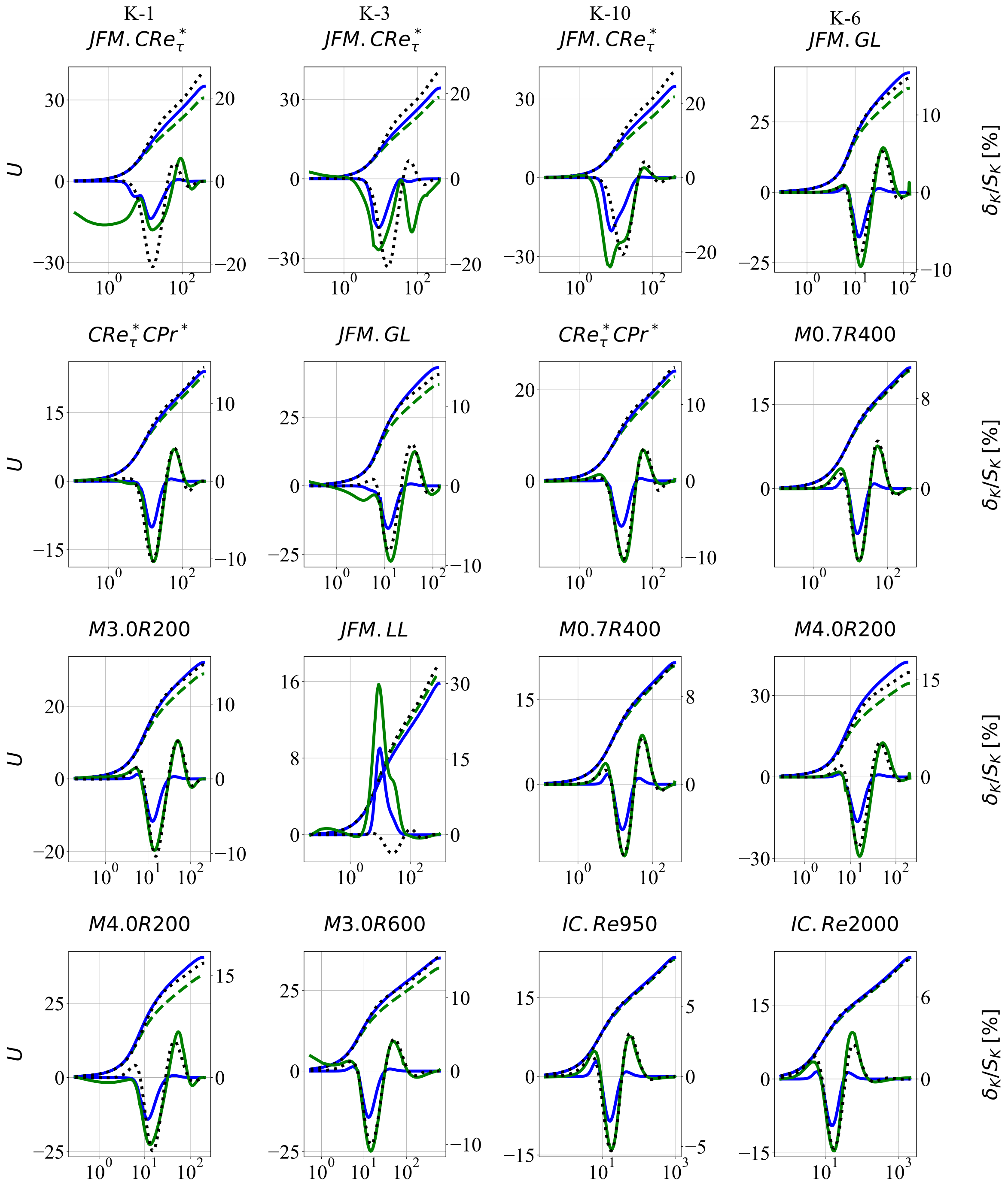


Figure 4.21: Selection of the results obtained for the K-fold validation trials at the extreme ends of the test error ranking presented in Table 4.5 during the prediction of the independent sets of $\boldsymbol{\delta_k}$ corrections for the MK turbulence model. The upper curves represent the initial RANS velocity profiles (green dashed lines), the data-augmented velocity predictions (blue solid lines) and the reference DNS data (red dotted lines). The lower curves present the initial predictions made by the Neural Network system (green solid lines), the results obtained after applying an intelligent relaxation factor of 0.5 (blue solid lines) and the ground-truth labels for the Field Inversion values (red dotted lines).

Regarding the uncertainty quantification in the results, Figure 4.22 presents a synthesis of the results obtained for the DNS case $JFM.CRe_\tau^*$ in the K-fold trials {1, 2, 3, 4, 7, 10}. The colored area shown in this Figure corresponds to the standard deviation obtained for the error. While it can be seen that the standard deviation obtained is fairly large, the Machine Learning model managed to produce positive contributions to the system nevertheless, and to reduce the maximum error by 9.4% in average.

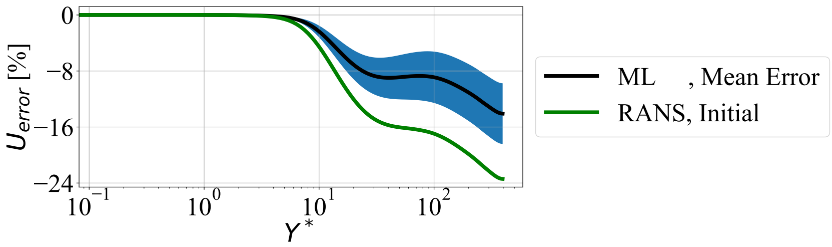


Figure 4.22: Results of the uncertainty quantification process followed for the DNS case $JFM.CRe^*_\tau$ while employing the results of the K-fold validation runs {1, 2, 3, 4, 7, 10} after the prediction of the independent set of $\boldsymbol{\delta_k}$ corrections for the MK turbulence model.

A deeper analysis of the results obtained by every K-fold validation run revealed that the logarithmic neurons present in the system created a different vector basis in each case. The only clear trend present in the K-fold trial runs was the cancellation of the k-equation diffusivity ($Diff_k$) as an input feature. The behavior of the Neural Network regarding this feature is similar to the trends observed in Section 3.4 once the density (ρ) was added as a feature to predict the drag coefficient of a cylinder (C_D). In such case, it could be noted that the Logarithmic neurons cancelled this feature, since its values only corresponded to randomly generated noise. Based on this experience, it can be concluded that the present Machine Learning system perceived the k-equation diffusivity ($Diff_k$) as being completely uncorrelated with the final results ($\boldsymbol{\delta_k}$).

MK Turbulence Model - Independent Set of $\boldsymbol{\delta_\epsilon}$ Corrections

The present Section will describe the results obtained after training a Machine Learning system to predict the independent set of $\boldsymbol{\delta_\epsilon}$ corrections for the MK turbulence model obtained in Section 4.4.1. The methodology employed to conduct this study was based on the K-fold validation trials described back in Table 4.4, which serve as a clear basis of comparison. The Neural Network architecture chosen was based on the lowest test errors obtained for the first K-fold validation trial (K-1) as well. The final Machine Learning architecture chosen can be found depicted in Figure 4.23. Here, it can be seen that the analysis produced a system containing 4 logarithmic neurons instead of 3 as before. However, these changes were compensated by employing a lower number of hyperbolic Tangent neurons in the subsequent hidden layer, thereby reducing the overall size of the system. The global training and test errors obtained for this architecture can be found listed in Table 4.6. Here, it can be seen that the system achieved comparable errors across a wide variety of test sets, except for the trial run K-8. A selection of the ML results obtained in the extreme cases of the test error ranking can be found in Figure 4.24. Here, it can be seen that the system achieved good results in the majority of the test scenarios, except for the DNS cases $M4.0R200$ and $JFM.GL$ in the trial runs K-1 and K-8. The issues observed in these cases result interesting, since it can be noted that the solver remained stable for both flow cases in the K-fold validation run (K-6). Based on this experience, it can concluded that the final predictions made by a Machine Learning system should be based on a committee of Neural Networks trained under different flow conditions, such as to create a reference to discard any spurious corrections.

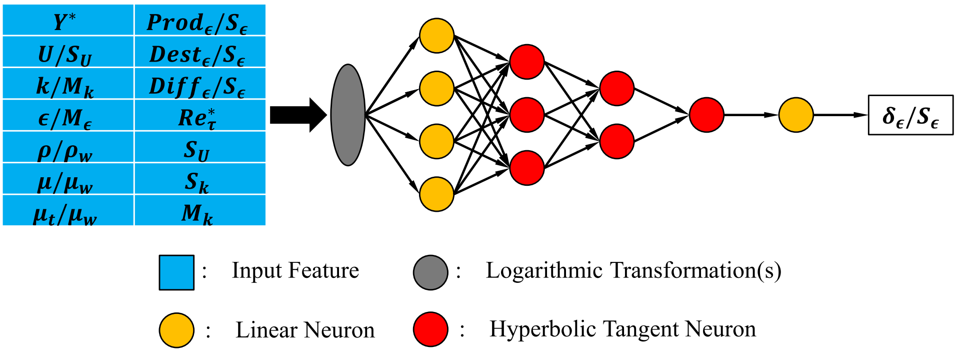

Y^*
$Prod_\epsilon/S_\epsilon$
U/S_U
$Dest_\epsilon/S_\epsilon$
k/M_k
$Diff_\epsilon/S_\epsilon$
ϵ/M_ϵ
Re_τ^*
ρ/ρ_w
S_U
μ/μ_w
S_k
μ_t/μ_w
M_k
$\delta_\epsilon/S_\epsilon$
: Input Feature
: Logarithmic Transformation(s)
: Linear Neuron
: Hyperbolic Tangent Neuron

Figure 4.23: Neural Network architecture created to predict the Field Inversion corrections (δ_ϵ) required by the MK turbulence model.

K-fold Run	$\mathcal{J}_{\text{train}}$	$\mathcal{J}_{\text{test}}$	Error Ranking
K-1	2.34	26.9	5
K-2	1.06	69.5	3
K-3	2.07	30.8	4
K-4	1.85	21.8	6
K-5	2.05	8.0	9
K-6	2.35	3.7	10
K-7	1.49	75.5	2
K-8	3.25	1758.2	1
K-9	1.00	20.4	7
K-10	1.99	15.0	8

Table 4.6: Normalized cost function values obtained for the training and test sets defined in every K-fold validation run during the prediction of δ_ϵ for the MK turbulence model. Reference scale: Minimum value encountered.

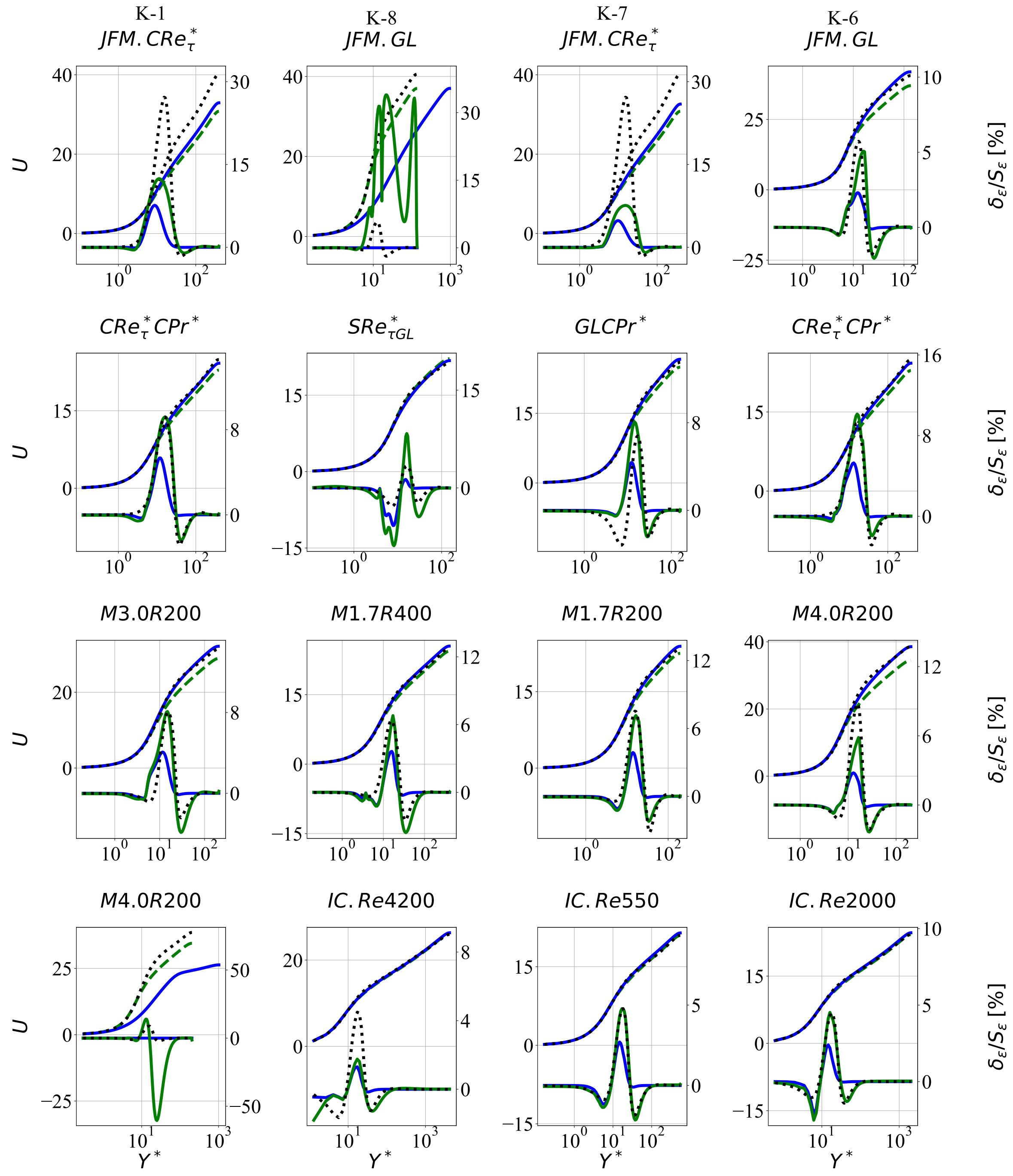


Figure 4.24: Selection of the results obtained for the K-fold validation trials at the extreme ends of the test error ranking presented in Table 4.5 during the prediction of the independent sets of $\boldsymbol{\delta_\epsilon}$ corrections for the MK turbulence model. The upper curves represent the initial RANS velocity profiles (green dashed lines), the data-augmented velocity predictions (blue solid lines) and the reference DNS data (red dotted lines). The lower curves present the initial predictions made by the Neural Network system (green solid lines), the results obtained after applying an intelligent relaxation factor of 0.5 (blue solid lines) and the ground-truth labels for the Field Inversion values (red dotted lines).

The results of the uncertainty quantification process followed for the DNS case $JFM.CRe_\tau^*$ can be found in Figure 4.25. Here, it can be noted that the ML system obtained a relatively low standard deviation error compared to the results presented back in Figure 4.22 for the set of $\boldsymbol{\delta_k}$ corrections. These trends can (again) be explained by analyzing the formula employed by the MK turbulence model for the eddy viscosity: $\mu_t = C_\mu \; f_\mu \; \rho \; k^2/\epsilon$. Here, it can be noted that μ_t is proportional to k^2, therefore, a 20% increase in k is immediately translated into a 44% change in the eddy viscosity μ_t. However, a similar increment of 20% in ϵ only produces a -17% change in μ_t. As a result, it can be noted that the predictions for the eddy viscosity are far less sensitive to any changes in the ϵ-equation, since this quantity produces more subtle changes. Beyond the previous discussion, the maximum overall error only presented an improvement of 7.9% on average, which is lower than the previous improvement margin of 9.4% obtained while employing δ_k corrections. This trend is associated to the higher quality of the predictions obtained in the former case (δ_k).

A detailed analysis of the weights employed by the Neural Network architecture revealed that the system categorically discarded the use of the feature $Diff_\epsilon$, which resembles the situation encountered in Section 4.5.2 for the variable $Diff_k$. As a result, it can be concluded that the diffusivity terms present in the MK turbulence model are not a good basis to build corrections. Beyond these results, no clear patterns were identified in the vector basis formed by the logarithmic neurons as before. This can be due to the presence of hidden correlations among the features or the lack of more training data. A detailed discussion regarding the global results will be provided in the next Section.

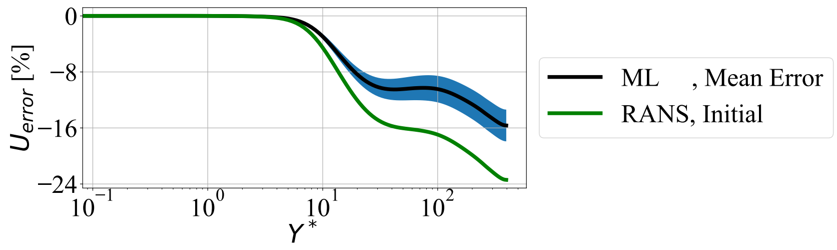


Figure 4.25: Results of the uncertainty quantification process followed for the DNS case $JFM.CRe_\tau^*$ while employing the results of the K-fold validation runs {1, 2, 3, 4, 7, 10} after the prediction of the independent set of $\boldsymbol{\delta}_\epsilon$ corrections for the MK turbulence model.

4.6. Final Remarks

Regarding the overall results obtained during the present Chapter, it can first be noted that the Field Inversion study of the MK turbulence model was successful. This can be evidenced both in the presence of clear inflection points for the Elbow method and in the stable patterns observed in the results. The overall nature of the corrections obtained hints at the absence of an appropriate modelling term to improve the behavior of the standard MK model in the buffer layer, which is the zone of highest turbulent activity.

During the Field Inversion study of the Spalart-Allmaras turbulence model [8], it was observed that the model did not show clear patterns nor a well-defined inflection point in the Elbow Method. These results can be attributed to the high initial accuracy of this model. As a rule-of-thumb, it was noticed during the present thesis that Field Inversion optimizers tend to yield over-fitted solutions when the baseline models employed presented low error margins. As a result, improving the Spalart-Allmaras turbulence model through the use of Field Inversion corrections does not appear feasible for variable-property flows. However, this model has been successfully corrected for general CFD studies in 2-D geometries [4]. The Field Inversion results obtained for the turbulent Prandtl number presented a similar problem to the corrections found for the SA turbulence model, since the accurate initial predictions available implied that no clear directions of advancement existed. However, the results still managed to approach the patterns observed in the DNS data studied by Patel (2017) [9] in the buffer layer.

Regarding the numerical properties of the Field Inversion optimizer created, it can be concluded that the use of $\boldsymbol{\beta}$ multipliers to drive the RANS turbulence model while employing a cost function seeking $\boldsymbol{\delta}$ corrections was successful, since the optimizer proved to be stable and the final results were not tied to any particular term in the RANS equations. As a result, the future use of this method is recommended, as well as the use of the Bold Drive Method with Added Momentum introduced back in Section 3.1.

Based on the analysis performed in Section 4.5.2, it can be concluded that the introduction of an intelligent relaxation factor to the system constituted a substantial contribution to the project, which should be included in any future study. Thanks to this method, valuable information could be recovered from predictions which were initially unstable.

Regarding the Machine Learning systems built to correct the MK turbulence model, it can be concluded that the system created to predict the independent set of $\boldsymbol{\delta}_{\mathbf{k}}$ corrections presented the best numerical properties. The Neural Network architectures trained under this context presented both the best improvement margins and the greatest numerical stability. Furthermore, it was noticed during the calculations performed that both the Field Inversion optimizer and the final Machine Learning architectures converged faster. The high standard deviation obtained during the prediction of the $\boldsymbol{\delta}_{\mathbf{k}}$ corrections can also be interpreted as a future opportunity to create a system which consistently deliver the best performance offered by the system.

Beyond the previous discussion, the Machine Learning systems created for the MK turbulence model revealed a clear need to incorporate more training data in the future. This could be evidenced in the high variability of the test errors obtained for each K-fold trial run, and in the unclear vector basis formed by the logarithmic neurons present in the system. Adding more training data would help the system to obtain a clear optimization basin for the local minimum. One of the greatest requirements in this context is a future database which not only contains a large variety of flow cases, but also DNS cases with a uniform complexity. In the present dataset, the flow cases considered ranged from subtle variations from neighbouring samples to unique DNS cases subject to large heat source terms, such as the cases denoted by the prefix JFM. As a result, it resulted complex to choose clear Cross-Validation and Test sets, since interpolating similar flow cases does not constitute a challenge.

The next Chapter will present an innovative analysis of turbulent channel flows, where a direct Machine Learning system will be built for the prediction of the velocity profiles. The formulation will be based on a re-defined turbulence parameter, which operates between clear bounds and can be easily visualized. Thanks to this parameter, it will also be possible to assess the true complexity of the turbulent channel flows studied, and to gain new insights regarding the properties of Machine Learning systems.

5 Alternative Machine Learning Formulation for Turbulent Channel Flows

This Chapter will present the derivation of a new turbulence parameter to describe the behavior of channel flows subject to strong property variations. The new parameter has clear operational bounds, which are suitable for both Machine Learning and general analysis purposes. In Section 5.1, the mathematical derivation of the new turbulence parameter (Φ) will be first presented, along with a brief description of its behavior in the dataset studied. Then, in Section 5.2, a Machine Learning system will be built based on the new formulation established, and the results obtained will be presented. A global discussion of the analysis performed and the new insights gained regarding Machine Learning will be offered in Section 5.3.

5.1. Mathematical Derivation of a New Turbulence Parameter (Φ)

In Chapter 4, it could be noted that correcting an existing RANS turbulence model could result highly complex and computationally expensive. Furthermore, the prediction of variable-property flows seemed to be guided by unclear constraints. The Cess algebraic model employs intricate non-linear terms, whereas general-purpose RANS turbulence models iterate until an equilibrium is reached in the full channel geometry. As a result, the behavior of turbulent channel flows remains hard to visualize, and the parameters employed in the literature do not provide the clear operational bounds required to build a classical Neural Network architecture. Modelling the eddy viscosity μ_t could result problematic in this context, since this quantity can grow indefinitely. The transformed $U^*(Y^*)$ velocity profiles defined by Patel (2017) [9] still require the definition of a system to predict the Y^* location of the centerline velocity for the flow. As a result, it is convenient to redefine the analysis of turbulent channel flows.

In order to derive a new turbulence parameter, the shear stress profile equation for channel flows was first considered:

$$(\mu + \mu_t)\ \nabla u = \tau_w \left(1 - \frac{Y}{H}\right). \tag{5.1}$$

However, since $\mu_t \geq 0$, the previous problem can also be written as:

$$K\ \ (\mu\ \nabla u) = \tau_w \left(1 - \frac{Y}{H}\right). \tag{5.2}$$

In eq. (5.2), K corresponds to the viscosity amplification ratio $K = (1 + \mu_t/\mu) \geq 1$. By further employing the arbitrary scaling conventions defined in Section 2.1.3, eq. (5.2) can be rewritten as:

$$\mu \, \nabla u = \frac{1}{K} \, (1 - y) \leq 1. \tag{5.3}$$

Eq. (5.3) corresponds to an important result for turbulent channel flows, since it imposes a strong limit upon the values of the velocity gradients for flows subject to constant pressure gradients. The inequality presented is also valid for flows subject to variable viscosity profiles. Furthermore, it can be noted that the values of $\mu\nabla u$ have clear operational bounds, since this quantity ranges from $\mu\nabla u = 1$ at $y = 0$, to $\mu\nabla u = 0$ at $y = 1$. The profiles obtained for $\mu\nabla u$ across the different DNS cases studied by Pecnik et al. (2017) [6] can be found in Figure 5.1 as a function of both y and Y^*. Here, it can be observed that predicting the values of $\mu\nabla u$ can create small contradictions, since the end-points of these curves are governed by $y = Y/H$, whereas their shape is better scaled by Y^*. During the present thesis work, the previous issues were avoided by creating an alternative formulation which combined both effects.

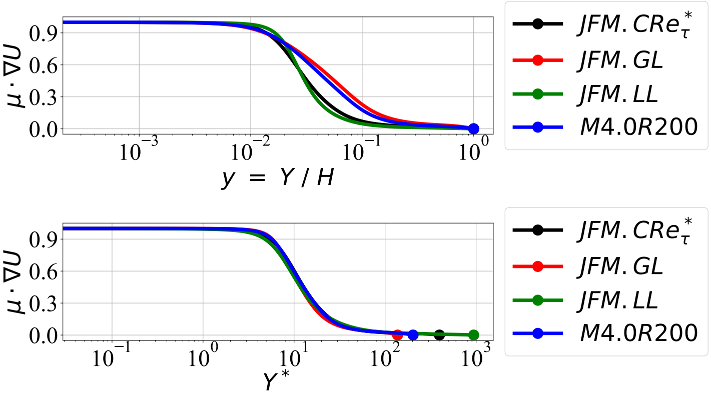


Figure 5.1: Profiles obtained for the product $(\mu\nabla u)$ during the analysis of the DNS cases defined by Pecnik et al. (2017) [6].

In order to build an improved diagnostic tool, it was first decided to create a solution space which was free from the boundary conditions imposed by the channel flow geometry. This was achieved by creating a function (Ω) which automatically fulfilled the boundary condition requirements, regardless of the values obtained for a shape parameter (φ). The requirements for the new function Ω can be expressed as:

$$\Omega = \mu \, \nabla u, \tag{5.4}$$

$$\Omega \, (y = 0) = 1, \tag{5.5}$$

$$\Omega \, (y = 1) = 0, \tag{5.6}$$

$$0 \leq \Omega \leq 1. \tag{5.7}$$

Based on the previous set of constraints, it can be proven that the following function is one of the simplest solutions which admits a preliminary shape parameter (φ):

$$\Omega = (1 - y) \, (1 - y \, \varphi)^2 \, . \tag{5.8}$$

In eq. (5.8), it can be noted that the shape parameter introduced (φ) does not affect the fulfillment of the end-point boundary conditions. The second term present in eq. (5.8) was squared in order to prevent the system from developing reversed flow conditions, especially within a Machine Learning context. Since the shape parameter φ is directly related to turbulence, it was detected that this quantity was directly scaled by the flow Reynolds number (Re^*_τ). As a result, the velocity gradients ∇u were finally expressed as:

$$\nabla u = \frac{1}{\mu} \, (1 - y) \, (1 - Y^* \, \Phi)^2 \, . \tag{5.9}$$

In eq. (5.9), the variable Φ corresponds to an exceptionally well-bounded shape parameter related to turbulence. The values of Φ obtained for a wide variety of DNS cases can be found in Figure 5.2. Here, it can be noted that all the flows studied follow similar trajectories as a function of Y^*, although small differences naturally exist. A detailed numerical analysis further revealed that every $\Phi(Y^*)$ curve present in the dataset could be fitted by a Gaussian function with 5 or 6 terms:

$$\Phi\left(Y^*\right)=\sum_{i=1}^{6} A_i\ e^{\left(Y^*-b_i\right)^2/c_i^2}. \tag{5.10}$$

The curve fitting errors obtained using the previous approximations remained below 0.78% for all the cases tested, except for the flow case $IC.Re4200$, which presented a maximum error of 1.3 %. Based on the previous analysis, it can be concluded that a DNS velocity profile can be described by approximately 15 or 18 DOF's (Degrees-of-Freedom), since every Gaussian term contains 3 parameters (a_i, b_i and c_i). However, a lower number of DOF's may exist in practice, since more advanced approximations could also be established for turbulent flows. Furthermore, the peak values for $\Phi(Y^*)$ observed in the buffer layer clearly indicate that this zone controls the emergence of turbulence, and that $\Phi=0$ corresponds to the recovery of a laminar flow regime.

Regarding the scope of the new turbulence parameter (Φ) proposed, it can be noted that its use could be potentially extended to general flat plate turbulent boundary layers. However, a preliminary study of the supersonic boundary layers modelled by Pirozzoli & Bernardini (2011) [29] revealed that the scaling parameters Y^* and Re_τ^* do not bring a universal collapse of the turbulence statistics encountered therein, and that further research is required regarding the parameters governing these flows.

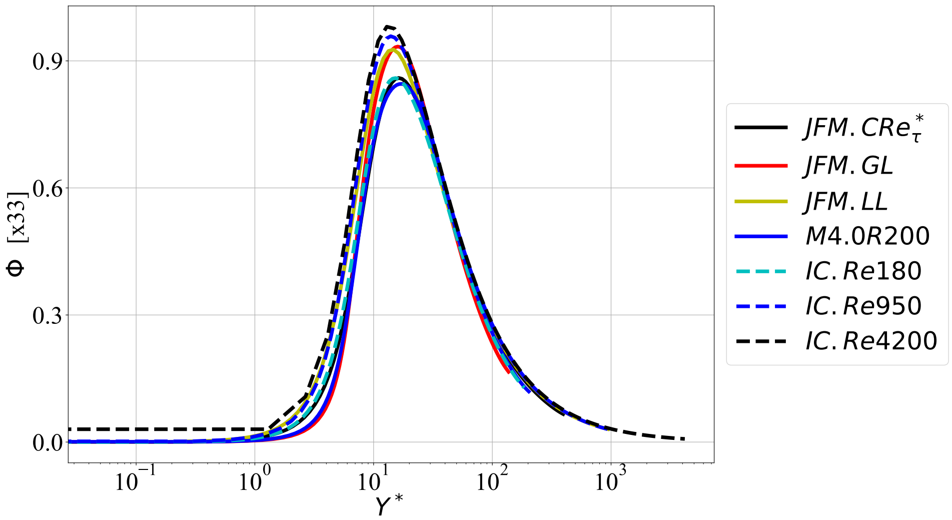


Figure 5.2: Trajectory curves obtained for the flow parameter $\Phi(Y^*)$ in the DNS cases defined by Pecnik et al. (2017) [6] and the incompressible channel flows studied by Jimenez et al. (2008) [10].

In summary, an exceptionally well-bounded turbulence parameter has been derived to model the velocity profiles of turbulent channel flows, and to propose an alternative Machine Learning formulation. The parameter created operates in log-space and presents an exponential decaying rate surrounding the peak values obtained in the buffer layer. Few other turbulence statistics present the regularity exhibited by the function (Φ) plotted in Figure 5.2.

5.2. Machine Learning Predictive System

This Section will describe the Machine Learning system built to predict the different $\Phi(Y^*)$ trajectory curves presented in Figure 5.2 for the whole dataset assembled. The K-fold validation trials defined back in Table 4.4 were again employed to perform the analysis, since this created a better analogy with the scenarios previously encountered. The Neural Network architecture selected for this case was chosen by minimizing the test errors obtained for the first K-fold validation trial established (K-1) as before. The resulting configuration can be found depicted in Figure 5.3. Here, it can be noted that logarithmic neurons were again employed for the first layer of the system, since their presence allows the optimizer to identify adequate input parameter groups. The use of Hyperbolic Tangent neurons in the subsequent hidden layers also constituted a good alternative, since the exponential curves obtained for the flow parameter (Φ) present smooth transitions which are compatible with these neurons. The variable α^* present in the stack of input features from Figure 5.3 corresponds to the scaling transformation defined by Patel (2017) [9] in order to establish the U^* velocity profiles:

$$\frac{\nabla U^*}{\nabla U} \approx \alpha^* = \left(1 + \frac{y}{Re_\tau^*}\nabla Re_\tau^*\right)\sqrt{\frac{\rho}{\rho_w}}. \tag{5.11}$$

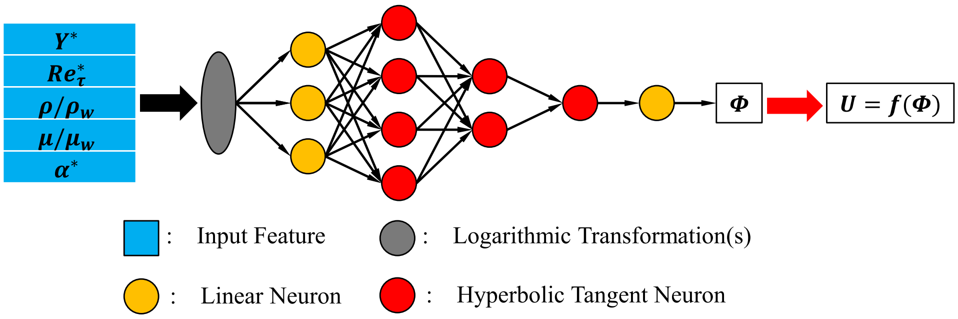


Figure 5.3: Neural Network architecture chosen to predict the values of Φ in the channel flows studied.

Since a good collapse of the U^* velocity profiles was observed in the work of Patel (2017) [9], it was decided to incorporate the previous transformation (α^*) into the Machine Learning system. The final Neural Networks trained assigned a substantial importance to this feature in fact. Beyond the previous considerations, it can be noted that the output of the Neural Network architecture presented in Figure 5.3 are explicit $\mathbf{U_{ML}}$ velocity profiles, which are employed to drive the cost function:

$$\mathcal{J}_{ML} = \sum_{i=1}^{N} \left(U_{ML,i} - U_i^*\right)^2 . \tag{5.12}$$

In order to avoid coupling a CFD solver to the Machine Learning optimizer, the velocity profiles $\mathbf{U_{ML}}$ were generated by employing an integration matrix ($\mathbf{A}$), such that $\mathbf{U} = \mathbf{A} \cdot \nabla U$. Based on the previous formula, it can be noted that the matrix ($\mathbf{A}$) contains the weights given by a trapezoidal integration scheme. The predictions made by the Neural Network were finally transformed into velocity profiles according to the following formulas:

$$\mathbf{U_{ML}} = \mathbf{A} \cdot \nabla \mathbf{U_{ML}}, \tag{5.13}$$

$$\nabla \mathbf{U_{ML}} = \frac{1}{\mu}\,(1-y)\,(1 - Y^*\Phi)^2 . \tag{5.14}$$

The previous formulas were readily incorporated into Tensorflow [42], since its built-in functions allow the user to multiply high-order tensors. Regarding this point, it is important to mention that the matrix $\mathbf{A}$ was implemented as a third-order rank tensor, where every slice corresponded to an independent 2-D integration matrix. Therefore, it was possible to model every flow case independently. The different tensors required by the Machine Learning system were saved as pre-defined Python dictionaries, which were generated by extensive functions. Thanks to this feature, the use of in-line commands to manipulate third-rank order tensors was completely avoided. The final training and test errors obtained by every K-fold combination can be found listed in Table 5.1. Here, it can be noted that the system reached a maximum test error of only 13 % in the K-fold validation run (K-3). The results obtained for the K-fold validation cases at the extreme ends of the test error ranking presented in Table 5.1 can be found in Figure 5.4. Here, it can be noted that the greatest test errors tend to occur near the channel center. However, it can be noted that the Machine Learning system sometimes manages to recover from the emerging deviations, as it occurred in the case $IC.Re2000$ from the K-fold trial run (K-6). In general, the predictions obtained for incompressible flow cases at high Reynolds Numbers ($IC.Re2000$ and $IC.Re4200$) tended to deviate from the DNS data, because the training database assembled contained scarce information regarding such cases. Most of the DNS flow cases studied by Patel (2017) [9] and Pecnik et al. (2017) [6] focus on the range $Re^*_\tau = \{150 - 395\}$, whereas the supersonic flow cases shared by Trettel & Larson (2016) [11] cannot be fully collapsed in the trends observed for the incompressible channel flows studied by Jimenez et al. (2008) [10]. Based on this context, it can be concluded that the Machine Learning system achieved a consistent behavior since all the $\Phi(Y^*)$ curves followed similar trajectories by default.

K-fold Run	$\mathbf{U}_{\text{train, error}}$	$\mathbf{U}_{\text{test, error}}$	Error Ranking
K-1	2.86 %	3.61 %	7
K-2	2.54 %	5.4 %	5
K-3	2.64 %	12.96 %	1
K-4	2.87 %	3.59 %	8
K-5	5.08 %	3.31 %	9
K-6	7.51 %	10.11 %	2
K-7	2.48 %	7.31 %	4
K-8	1.2 %	8.61 %	3
K-9	4.17 %	4.72 %	6
K-10	2.02 %	3.24 %	10

Table 5.1: Maximum velocity error values encountered during the prediction of the DNS flow cases considered in the training set and the test set of every K-fold validation run.

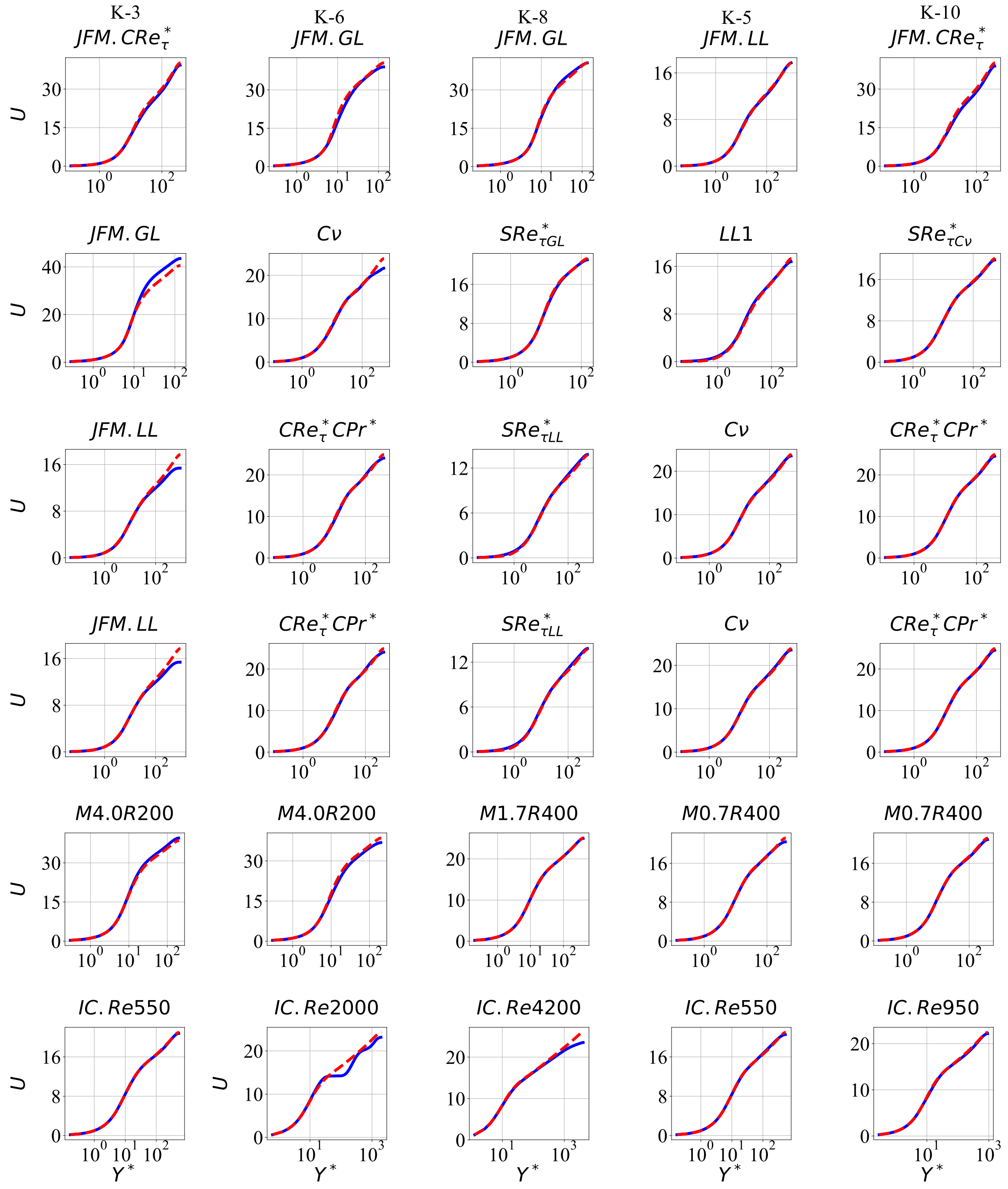


Figure 5.4: Results obtained for the velocity profiles considered in the test sets of the K-fold trial runs located at the extreme ends of the test error ranking presented in Table 5.1.

Beyond the previous discussion, it can be noted that the Machine Learning system managed to obtain excellent predictions for complex DNS flows such as the case $JFM.CRe_\tau^*$ and the different supersonic flow cases studied ($M3$, $M4$). The different DNS flow cases studied by Patel (2017) [9] were also predicted with exceptional accuracy. The only exceptions detected in this context were the predictions obtained for flow cases $JFM.GL$ and $JFM.LL$ in the K-fold validation run (K-3). A detailed analysis revealed that the issues observed in these flow cases could be attributed to their atypical friction Reynolds numbers (Re_τ) within the context of the database assembled, since they operated at $Re_\tau = 950$ and $Re_\tau = 150$ respectively. However, it can still be observed that the Machine Learning system reached high levels of performance for such flow cases in the remaining K-fold validation runs.

Regarding the stability of the Machine Learning system created, it is important to mention that those velocity predictions which presented imperceptible changes with respect to the DNS data according to Figure 5.4 remained identical after coupling the energy equation to the system. However, the flow cases presenting substantial deviations with respect to the DNS data were prone to diverge, such as the cases $JFM.LL$ and $JFM.GL$ from the K-fold trial run (K-3). The numerical issues found in such cases were linked to the erroneous μ_t profiles obtained, which contained sharp variations that invalidated the turbulent Prandtl number analogy employed by the energy equation. One direct solution to the previous problem would be to introduce a secondary Field Inversion optimizer, such that a moderate μ_t distribution could be recovered for every velocity profile. However, the running times associated to Field Inversion optimizers are orders of magnitude greater than the solution times of CFD solvers. As a result, the previous procedure is not feasible in practice. Another alternative would be to generate an estimate of the real μ_t profiles by employing a secondary RANS turbulence model, such that the intelligent relaxation factor methodology could be employed to mitigate the values of μ_t. While this alternative is far more feasible than introducing a secondary Field Inversion optimizer, it can be noted that the present Machine Learning architecture would become dependent of an additional support model to correct the energy equation. A final alternative would be to create a large database, such that only similar flow cases are modelled. This solution is feasible in principle, since it could be observed in Figure 5.4 that the present Machine Learning architecture can make accurate predictions for flows cases which are well-represented in the training dataset.

The results of the uncertainty analysis performed for the flow case $JFM.CRe_\tau^*$ can be found in Figure 5.5, which presents the mean error and the standard deviation obtained after coupling the energy equation to the system. Here, it can be noted that the errors obtained are substantially lower than for the Machine Learning systems created in Chapter 4, since the average error manages to remain below 1.9%. These results prove that the issues encountered in the flow case $JFM.CRe_\tau^*$ while applying the MK turbulence model are caused by the numerical properties of this model, and not by the complexity of the DNS case studied. One interesting trend observed in Figure 5.5 is the reduction of both the mean error and its standard deviation as the flow enters the log-layer. These results also mark a contrast with the uncertainty curves presented in Chapter 4, which presented monotonically increasing errors. Based on these insights, it can be concluded that the present Machine Learning system learned to compensate the errors obtained in certain regions by applying posterior increments. An exacerbated example of this behavior could be observed in the test case $IC.Re2000$ for the K-fold validation run (K-6). A detailed review of the input coefficients employed by the model can be found listed in Appendix C.1. Among the results obtained, it results interesting the note that the K-fold validation run which achieved the lowest error margins (K-10) assigned a large importance to the scaling transformation (α^*) defined by Patel (2017) [9]. Therefore, the use of such feature is important to model variable-property flows, and its inclusion into traditional RANS turbulence models should be further explored.

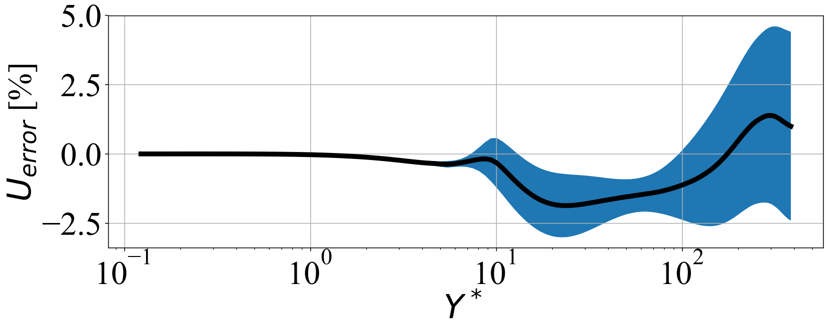


Figure 5.5: Results of the uncertainty quantification process followed for the case $JFM.CRe_\tau^*$ after employing the results of the K-fold trial runs {1, 2, 3, 4, 7, 10}, and coupling the thermal energy equation to the system.

5.3. Final Remarks

The results obtained during the present Chapter indicate that the new flow parameter derived (Φ) corresponds to an adequate representation of turbulence in channel flows. Few other turbulence statistics described in the literature manage to reach the levels of regularity observed in Figure 5.2 for the $\Phi(Y^*)$ parameter across various DNS flow cases. Due to this reason, the use of the Φ parameter can be recommended as a general diagnostic tool for the future, since it allows a quick assessment of the deviations obtained between different flow cases. The high levels of regularity observed in the trajectories of the Φ parameter curves also allowed the Machine Learning systems to make accurate flow predictions, despite the lack of a baseline model to yield a solution estimate. The maximum error obtained during the K-fold validation trials only reached 13%, and the DNS flow case $JFM.CRe_\tau^*$ was modelled with a maximum average deviation of 1.9%. This proves that the difficulties encountered by the previous Machine Learning systems created for the MK turbulence model during the prediction of the DNS case $JFM.CRe_\tau^*$ were tied to the numerical properties of such model.

Based on the previous context, it can be concluded that the problem of building corrections for an existing RANS turbulence model corresponds to a ”double inference” problem, where the Machine Learning system must guess both the flow physics of the respective case, as well as the mathematical corrections required by the PDE’s of the RANS turbulence model. This scenario can substantially increase the Machine Learning training data required by the Neural Network, since the mathematical behavior of the RANS turbulence model must be additionally assessed. Based on these insights, it can be noted that the tools provided by Machine Learning should be used to explore the possibility of building new refined physical models, which only depend on physical parameters subject to clear operational bounds.

Beyond the previous discussion, it is important to note that the Machine Learning system developed during the present Chapter could result unstable for those cases where the Neural Networks developed inaccurate predictions. The issues were caused by the difficulties encountered during the prediction of μ_t to drive the energy equation though the use of the turbulent Prandtl number analogy (Pr_T). Among the different solutions explored, it was theorized that one of the best alternatives consisted in employing a secondary RANS turbulence model, whose results could be used to apply the intelligent relaxation factor methodology to mitigate the values of μ_t encountered.

The abrupt changes occasionally encountered in the velocity profiles plotted in Figure 5.4 are tied to the numerical properties of traditional Neural Networks, which are prone to produce output curves presenting staircase steps in the case of sigmoidal functions, or sharp breaks in the case of RELU functions. As a result, it can be concluded that the direct prediction of the velocity profiles for a system is only feasible once large databases are assembled, or once a mechanism is introduced to restrict the values of μ_t inferred.

Due to the issues previously discussed, it can finally be concluded that the building δ_k corrections for the MK turbulence model emerged as the most reliable Machine Learning system alternative. While the improvement margins obtained were often narrow, the model was exceptionally stable and independent. Furthermore, the use of the intelligent relaxation factor (α) methodology derived in Section 4.5.1 greatly improved the reliability of the corrections obtained, since any spurious corrections could be automatically discarded. The incorporation of the scaling transformation factor (α^*) defined by Patel (2017) [9] during the definition of the U^* velocity profiles could bring greater improvements into traditional RANS turbulence modelling, since the most successful Neural Network architectures built assigned a large importance to this factor.

6 Conclusions and Recommendations

6.1. Summary and Conclusions

The present thesis project explored the different possibilities available to build improved data-driven RANS turbulence models for variable-property flows. The analysis was based on both the direct application of Neural Network architectures to the DNS database assembled, as well as in the application of a technique known as FIML (Field Inversion Machine Learning), which was proposed by Parish & Duraisamy (2016) [5]. Two of the original examples proposed by these authors were replicated in Chapter 3, where additional introductory problems were also defined. Based on the preliminary examples studied, it was possible to analyze the numerical properties of Field Inversion optimizers, and to synthesize a fast and reliable Hessian-free optimization algorithm, which formally corresponds to the Bold Drive Method with Added Momentum. Thanks to this optimizer, the overall process of Field Inversion was accelerated by orders of magnitude, while also providing exceptional numerical stability and a reasonable proof of local convergence. The use of a Sympy script to generate the matrices required by the Discrete Adjoint Method further improved the efficiency of the system by a factor of x1000 approximately, since the use of Automatic Differentiation (AD) libraries was completely avoided.

While the choice of the previous numerical schemes can seem like a natural alternative from a Machine Learning perspective, it is important to mention that the original Bayesian Field Inversion methodology developed by Parish & Duraisamy (2016) [5] was substantially more complex and only employed an approximate probabilistic approach based on Gaussian assumptions, whose consistency cannot be verified in large-scale systems. The issues found in this formulation were caused by the PDF's (Probability Density Functions) supplied to the Bayesian Field Inversion optimizer, since Parish et al. (2016) [5] discussed that Gaussian distributions must be employed in order to avoid using a more rigorous technique known as MCMC (Markov chain Monte Carlo) sampling, which can result prohibitive even during the optimization of 1-D RANS turbulence models. However, the Jacobian optimization schemes adopted during the present thesis can offer fast and reliable convergence properties, since only the optimal parameter distributions calculated are required in the FIML methodology. Therefore, it can be concluded that the use of Bayesian Field Inversion optimizers should be approached with caution; at least during the optimization of RANS turbulence models.

The analysis presented in Chapter 4 described the application of the FIML methodology to the DNS database for variable-property flows assembled. However, before starting such analysis, the energy equation of fluids was adapted in order to model the supersonic DNS flow cases shared by Trettel & Larson (2016) [11], and the existing CFD solvers of the Process & Energy research group were thus extended. The initial RANS turbulence modelling benchmark performed revealed that the Spalart-Allmaras model [8] and the MK model [7] corresponded to the most interesting optimization targets, since they presented the best overall performance and the highest modelling errors respectively. The innovations introduced into the Field Inversion optimization procedure allowed the optimizer to build explicit source terms corrections ($\boldsymbol{\delta}$) while only employing $\boldsymbol{\beta}$ multipliers for one of the terms present in the original RANS governing equations. As a result, the Field Inversion optimizers created were able to achieve good results without compromising their general scope nor their numerical stability.

The results obtained during the previous optimization process for the MK turbulence model further revealed that it was possible to obtain independent sets of Field Inversion corrections for every governing equation ($\boldsymbol{\delta}_{\mathbf{k}}$ and $\boldsymbol{\delta}_{\boldsymbol{\epsilon}}$), which presented good numerical properties. As a result, it was decided to create separate Machine Learning systems for every set of independent corrections obtained. The Neural Networks architectures developed in both cases reached similar test errors, although the predictions made for the $\boldsymbol{\delta}_{\boldsymbol{\epsilon}}$ corrections resulted less stable in practice. As a result, it was concluded that the Neural Network architecture created to predict $\boldsymbol{\delta}_{\mathbf{k}}$ corrections corresponded to the best overall alternative. The introduction of the intelligent relaxation factor (α) methodology into the analysis performed also proved to be a valuable asset, since the formula invented was capable of deleting any $\boldsymbol{\delta}$ corrections which may have compromised the stability of the RANS turbulence model. The unstable behavior occasionally obtained during the prediction of $\boldsymbol{\delta}_{\boldsymbol{\epsilon}}$ corrections was caused by an irremediable discrepancy between the energy equation and the predictions made by the Neural Network, and not by an issue in the turbulence budgets of the RANS equations.

Regarding the Field Inversion corrections obtained for the Spalart-Allmaras turbulence model ($\boldsymbol{\delta}_{\mathbf{SA}}$) and the turbulent Prandtl number ($\mathbf{Pr_T}$), it was detected that inconsistent patterns were obtained by the Field Inversion optimizer. Based on the similitude between both sets of corrections, it was theorized that Field Inversion optimizers were unable to obtain clear directions of advancement once a baseline model presented small errors. The previous issue can be alternatively interpreted as stating that an accurate baseline model can only be improved through direct over-fitting. Due to the previous issues, it was decided not to build Neural Network predictive systems based on sets of corrections $\boldsymbol{\delta}_{\mathbf{SA}}$ and $\mathbf{Pr_T}$.

The last stage of the analysis consisted in building a direct Machine Learning predictive system for the velocity profiles obtained in every DNS flow case considered. The alternative formulation presented in Chapter 5 was based on an innovative turbulence flow parameter (Φ), which showed a remarkable collapse between the trajectories followed by each DNS flow case studied. The visualization of this parameter corresponded to one of the clearest representations of turbulence in channel flows available. The Neural Network architectures trained were able to obtain accurate predictions for the DNS flows cases studied by Patel (2017) [9]. The high quality of the predictions was attributed to the properties of the new turbulence parameter (Φ) formulated, which collapsed the different DNS flow cases into similar trajectory curves. As a result, the Machine Learning system had abundant training data to make accurate predictions. However, it was detected that the Machine Learning system presented a high sensitivity with respect to unseen Re_τ^* distributions, since this parameter created substantial changes in the $\Phi(Y^*)$ trajectories obtained. Beyond such considerations, the Machine Learning architecture could be successfully coupled with the energy equation of fluids in order to make predictions for any DNS flow case modelled with accuracy. However, the system quickly turned unstable if the velocity profiles predicted contained sharp variations, which were associated to large spikes in the eddy viscosity (μ_t) profiles. Any abrupt changes in the μ_t profiles immediately destabilized the energy equation solver, since the turbulent Prandtl analogy became invalid. As a result, it was concluded that the alternative Machine Learning formulation should be accompanied by a secondary RANS turbulence model, whose predictions can be used to discard any abrupt changes in the eddy viscosity profiles given to the energy equation (μ_t). The values of μ_t obtained are otherwise inconsequential, since the Machine Learning system makes direct predictions for the velocity gradients.

One global conclusion obtained during the present thesis project was the need to incorporate more training data into the analysis. The need for more training data could be clearly detected once several K-fold validation trials were performed, since every Neural Network trained reached substantially different weights. As a result, the training data was insufficient to define a common optimization basin with a clear local minimum, despite the small nature of the Neural Networks employed. While the predictions for the $\boldsymbol{\delta}_{\mathbf{k}}$ corrections required by the MK turbulence model corresponded to the most stable Machine Learning system developed, it is important to note that the corrections obtained in each case presented highly different shapes, as it can be evidenced in Chapter 4. Since the trajectory curves obtained in Chapter 5 for the turbulence parameter $\boldsymbol{\Phi}$ resulted far more homogeneous, it can be concluded that building direct corrections for an existing RANS turbulence model can increase the mathematical complexity of the corrections required. Due to these reasons, the present analysis suggested that future research should explore new alternatives for incorporating Machine Learning corrections into RANS turbulence modelling. The possibility of building a non-linear space where different flows are inherently similar should constitute the first priority in data-driven RANS turbulence modelling.

Beyond the previous discussion, the intelligent relaxation factor methodology derived, accompanied by predictions for $\boldsymbol{\delta}_{\mathbf{k}}$, was able to produce positive changes in the MK turbulence model almost without exception. Therefore, the analysis performed during the present thesis was deemed successful.

6.2. Recommendations

One of the first recommendations which can be derived from the present study is to avoid using the Bayesian Field Inversion methodology proposed by Parish et al. (2016) [5] during the optimization of RANS turbulence models. As it was discussed during the previous Section, the methodology proposed by these authors has a mathematical formulation which is unnecessarily complex while not providing any clear advantages. The use of a simple Jacobian optimizer based on the Bold Drive Method with Added Momentum should be thus preferred, especially since this method actively seeks the best optimization hyper-parameters to use and can be fully automatized. The Sympy script presented in Appendix A.3 can be further adapted in order to derive the matrices required by the Discrete Adjoint Method during the optimization of any RANS turbulence model. Thanks to this script, the use of Automatic Differentiation (AD) libraries can be avoided, and the efficiency of an optimizer can be increased by orders of magnitude according to the tests performed.

While building a Neural Network predictive system, it is recommended to employ a limited number of logarithmic neurons in order to determine the best parameter groups to use, such as the Reynolds number, the Prandtl number or any intermediate combinations (e.g. $Re^a Pr^b$). Employing Hyperbolic Tangent neurons in deeper hidden layers can also bring benefits in fluid mechanics, since these neurons produce smooth output distributions free of the sharp discontinuities found in RELU activation functions. In order to analyze the behavior of any Neural Network architectures created, it is recommended to employ the K-fold validation procedure described in Section 2.3.5 to check if a unique distribution of neural weights can be obtained. If every K-fold validation trial converges to the same Neural Network parameters, then it is reasonable to conclude that enough training data was given to the optimizer and that the system created is stable.

If the Machine Learning corrections created contains non-physical spikes or otherwise produces divergence, it is recommended to employ the intelligent relaxation factor methodology derived in Section 4.5.1. The methodology proposed therein can perform an intelligent selection of the best corrections to keep in less than a millisecond according to the tests performed. The use of this methodology is recommended for any future studies.

Regarding the Machine Learning analysis performed using the existing DNS database of variable property flows assembled, it is recommended to enhance this database before attempting to build direct Machine Learning corrections for a RANS turbulence model ($\boldsymbol{\delta}$ or $\boldsymbol{\beta}$), since the results obtained during the present study showed that significant changes occurred between different DNS cases. The original results obtained by Parish & Duraisamy (2016) [5] for incompressible channel flows also showed similar trends. Alternatively, it can also be recommended to build innovative parameters for RANS turbulence modelling, such that smooth correction curves can be obtained for different DNS datasets by default. The results obtained in Chapter 5 seem to indicate that simpler turbulence trajectory curves may exist in practice.

A

Source Code for the Examples Presented in Chapter 3

A.1. Field Inversion Optimization of a 1-D Radiative Heat Transfer Profile

A.1.1. Version A: Build β multipliers for P_k (Inefficient)

```
import numpy as np
import matplotlib
from copy import deepcopy
plt=matplotlib.pyplot
pi=np.pi
sin=np.sin
exp=np.exp

# ----------------------------------------------------
# # 1-D Field Inversion Radiative Heat Transfer Problem
# ----------------------------------------------------

# ----------------------------------------------------
# # # # # Problem Replicated from:
# Eric J. Parish, Karthik Duraisamy,
# A paradigm for data-driven predictive modeling using field inversion and
 ↪ machine learning,
# Journal of Computational Physics,
# Volume 305,
# 2016,
# Pages 758-774,
# ISSN 0021-9991,
# https://doi.org/10.1016/j.jcp.2015.11.012.
# (http://www.sciencedirect.com/science/article/pii/S0021999115007524)
# Keywords: Data-driven modeling; Machine learning; Closure modeling
# ----------------------------------------------------
zT_Parish=np.array([[3.12523747e-02, 1.63138286e+01], [6.25036317e-02,
 ↪ 2.90723605e+01],
       [9.37515381e-02, 3.76641118e+01], [1.24996094e-01, 4.29329280e+01],
       [1.56253307e-01, 4.60092889e+01], [1.87500842e-01, 4.77622723e+01],
       [2.18752843e-01, 4.87502426e+01], [2.49997027e-01, 4.93031644e+01],
       [2.81252379e-01, 4.96118198e+01], [3.12499913e-01, 4.97837545e+01],
       [3.43744469e-01, 4.98791455e+01], [3.74999821e-01, 4.99326106e+01],
       [4.06249589e-01, 4.99612664e+01], [4.37500845e-01, 4.99772290e+01],
       [4.68745401e-01, 4.99849411e+01], [5.00000753e-01, 4.99864797e+01],
       [5.31248660e-01, 4.99849411e+01], [5.62496194e-01, 4.99772290e+01],
       [5.93748195e-01, 4.99612664e+01], [6.24996102e-01, 4.99326106e+01],
       [6.56253315e-01, 4.98791455e+01], [6.87497499e-01, 4.97837545e+01],
       [7.18749128e-01, 4.96122237e+01], [7.49989589e-01, 4.93031644e+01],
       [7.81250525e-01, 4.87506272e+01], [8.12498432e-01, 4.77626569e+01],
       [8.43744477e-01, 4.60092889e+01], [8.75001318e-01, 4.29333126e+01],
       [9.06245874e-01, 3.76648811e+01], [9.37497503e-01, 2.90716104e+01],
       [9.68741686e-01, 1.63126555e+01]])
#
zBeta_Parish=np.array([[0.03124516, 1.17785807], [0.06250632, 1.54141139],
       [0.09374702, 1.6069432 ], [0.12500110, 1.57388281],
       [0.15624851, 1.53103891], [0.18750261, 1.49959031],
       [0.21874702, 1.47980724], [0.24999851, 1.46810165],
       [0.28125633, 1.46138052], [0.31250261, 1.45759421],
       [0.34374703, 1.45545334], [0.37499852, 1.45426899],
       [0.40625261, 1.45361606], [0.43749666, 1.4532657 ],
       [0.4687541 , 1.45309742], [0.49999517, 1.45304357],
       [0.53125262, 1.45309742], [0.56249852, 1.4532657 ],
       [0.59375262, 1.45361606], [0.62500262, 1.45426899],
       [0.6562489 , 1.45545334], [0.68750262, 1.45759421],
       [0.7187448 , 1.46138052], [0.75000262, 1.46810131],
       [0.78124853, 1.47980724], [0.81250263, 1.49959031],
       [0.84374518, 1.53103891], [0.8749989 , 1.57388281],
       [0.90625263, 1.6069432 ], [0.93749481, 1.54139086],
       [0.96875635, 1.17781769]])
#
class Class_Mesh(): pass
class Class_Model(): pass
# Create Mesh
theMesh                        = Class_Mesh()
theMesh.N_points               = N_points = 30
theMesh.z_coords               = z_coords = np.linspace(0.,0.5,N_points)
theMesh.inds_z_cond            = [0]
__ones=np.ones(z_coords.shape)
theMesh.G2_matrix              = -2*np.diag(__ones) +
 ↪ np.diag(__ones[1:],1)+ np.diag(__ones[1:],-1)
theMesh.G2_matrix              /= np.mean(np.diff(z_coords))**2
theMesh.G2_matrix [ 0,:] *=0.
# theMesh.G2_matrix [-1,:] *=0.
theMesh.G2_matrix [-1,-2]*=2.
# Create Solver
def f_solve_Model(selfModel):
    #
    # Generic Equation
    # G2*T = beta(z) * e(T) * (T^4-T_inf^4) + h * (T-T_inf)
    # (beta and h_conv may be omitted)

    # beta(z) * e(T) * (T_inf^4) + h * (T_inf) =  beta(z) * e(T) * (T^4) +
     ↪ h * (T) - G2*T
    # A =  beta(z) * e(T) * (T^3) + h - G2
    # b = beta(z) * e(T) * (T_inf^4) + h * (T_inf)
    h_conv = selfModel.h_conv
    T_inf  = selfModel.T_inf
    BetaZ=selfModel.BetaZ
    i_iter=0
    vkey=True
    N_iter_max=1000
    N_iter_min=20
    alpha=0.5
    while vkey:
        i_iter+=1
        eT=selfModel.epsilon_T(selfModel.T)
        A= np.diag(BetaZ*eT*(selfModel.T**3)+h_conv) -
          ↪ selfModel.Mesh.G2_matrix
        b = BetaZ*eT*(T_inf**4) + h_conv*T_inf

        A=A[1:,:]; A=A[:,1:]
        b= b[1:]
        T_new= np.linalg.solve(A,b)
        T_old=selfModel.T[1:]
        if max(np.fabs(T_old-T_new))<1e-11: vkey=False
        if (i_iter>N_iter_max) : vkey=False ; print('Warning: Maximum
          ↪ Numbers of Iterations Reached')
        if (i_iter<N_iter_min) : vkey=True
        selfModel.T[1:] = (1-alpha)*T_old + alpha*T_new
    # print(i_iter)
    return selfModel
#

# Create True Model
TrueModel                   = Class_Model()
TrueModel.Mesh              = theMesh
TrueModel.h_conv            = 0.5
TrueModel.T_inf     = T_inf = 50.
TrueModel.epsilon_T         = lambda T:
  ↪ 1e-4*(1.+5*sin(3*pi/200.*T)+exp(0.02*T) )
TrueModel.BetaZ             = np.ones(z_coords.shape)
TrueModel.T                 = np.ones(z_coords.shape) ;
TrueModel.T[0]=0. ;
#
TrueModel=f_solve_Model(TrueModel)

#

# Create Rough Model
RoughModel           = Class_Model()
RoughModel.Mesh      = theMesh
RoughModel.h_conv    = 0.
RoughModel.T_inf     = T_inf
RoughModel.epsilon_T = lambda T: 5e-4
RoughModel.BetaZ     = np.ones(z_coords.shape)
RoughModel.T         = np.ones(z_coords.shape) ;
RoughModel.T[0]=0. ;
RoughModel=f_solve_Model(RoughModel)
#
BoolPlotIni=False
if BoolPlotIni:
    matplotlib.rcParams.update({'font.size': 18})
    matplotlib.rcParams.update({'font.family': ”Times New Roman”})
    plt.rcParams.update({'font.size': 18})
    plt.rcParams.update({'font.family': ”Times New Roman”})

    plt.plot(TrueModel.Mesh.z_coords, TrueModel.T, 'b', linewidth=3,
      ↪ label='True Model')
    plt.plot(RoughModel.Mesh.z_coords, RoughModel.T, 'r', linewidth=3,
      ↪ label='Rough Model')
    plt.plot(zT_Parish[:,0],zT_Parish[:,1], 'go', markersize=6,
      ↪ label='Parish et al. (2016)')
    plt.legend(loc='best')
    plt.xlabel('z')
    plt.ylabel('T')
    plt.grid()
    plt.show()
#

RoughModel.Tstar = TrueModel.T
RoughModel.DJDB  = np.zeros(RoughModel.T.shape)
def f_get_Jacobian(selfModel):
    # Rough Model Equation
    # G2*T = beta(z) * e_0 * (T^4-T_inf^4)
    # RR = beta(z) * e_0 * (T^4-T_inf^4) - G2*T
    # dRRdT = 4*beta(z) * e_0 * (T^3) - G2
    # dRRdB = e_0 * (T^4-T_inf^4)

    # JJ= sum((T - T*)**2)
    # dJJdT = 2*(T - T*)
    # dJJdB = 0.
    selfModel=f_solve_Model(selfModel)

    T=selfModel.T
    e0 = selfModel.epsilon_T(T)
    dRRdT = np.diag(4*selfModel.BetaZ*e0* (T**3)) -
      ↪ selfModel.Mesh.G2_matrix
    dRRdB = np.diag(e0* (T**4 - T_inf**4))

    dJJdT = 2*(T- selfModel.Tstar)
    dJJdB = np.zeros(dJJdT.shape)

    dRRdT=dRRdT[1:,:] ; dRRdT=dRRdT[:,1:]
    dRRdB=dRRdB[1:,:] ; dRRdB=dRRdB[:,1:]

    dJJdT=dJJdT[1:]
    dJJdB=dJJdB[1:]

    Psi=np.linalg.solve(dRRdT.transpose(),-dJJdT)
    DJDB = dRRdB @ Psi + dJJdB
    # Discrete Adjoint Method
    selfModel.DJDB[1:] = DJDB
    selfModel.Jcost=np.sum((T- selfModel.Tstar)**2)
    return selfModel
#
RoughModel=f_get_Jacobian(RoughModel)

# Bold Drive Method with Added Momentum
kPlus  = 1.2
```

```
kMinus = 0.5
c_1    = 0.9
m_n    = np.zeros(RoughModel.DJDB.shape)
alpha  = 0.1 / np.fabs(RoughModel.DJDB).max()


def f_get_beta_real(selfModel,TrueModel=TrueModel):
    T     = selfModel.T
    T_inf = selfModel.T_inf
    e0=RoughModel.epsilon_T(T)
    selfModel.BetaZ_real = TrueModel.epsilon_T(T)/e0 +
      ↪ TrueModel.h_conv*(T-T_inf) / (e0*(T**4-T_inf**4))
    return selfModel
#
N_iters_print = 200
iiter   =0
n_count =0
vkey=True
while vkey:
    iiter  +=1
    n_count+=1
    BackupModel = deepcopy(RoughModel)
    m_n = (c_1*m_n + (1-c_1)*RoughModel.DJDB)#/(1-c_1**n_count)
    RoughModel.BetaZ -= alpha*m_n
    #
    RoughModel=f_get_Jacobian(RoughModel)
    if RoughModel.Jcost<BackupModel.Jcost:
        alpha*=kPlus
    else:
        alpha *=kMinus
        RoughModel = deepcopy(BackupModel)
        m_n     = np.copy(RoughModel.DJDB)
        # n_count=0
    if alpha<1e-14: vkey=False
    if iiter%N_iters_print==0: print('Iteration %8d: Jcost=%12.4e (alpha =
      ↪ %12.4e)' % (iiter,RoughModel.Jcost,alpha))
print('Iteration END: Jcost=%12.4e (alpha = %12.4e)' %
  ↪ (RoughModel.Jcost,alpha))
#
def f_reflect(selfModel):
    try:
        if selfModel.has_reflected:
            print('Model already reflected, operation skipped!')
        else:
            raise Exception('Error: selfModel.has_reflected should be True
              ↪ or undefined')
    except:
        selfModel.Mesh.z_coords= np.append( selfModel.Mesh.z_coords,
          ↪ 1.0-selfModel.Mesh.z_coords[::-1][1:])
        f_complete = lambda x:  np.append( x, x[::-1][1:])
        selfModel.T      = f_complete(selfModel.T)
        selfModel.BetaZ= f_complete(selfModel.BetaZ)
        selfModel.has_reflected=True
    return selfModel
#
matplotlib.rcParams.update({'font.size': 18})
matplotlib.rcParams.update({'font.family': ”Times New Roman”})
plt.rcParams.update({'font.size': 18})
plt.rcParams.update({'font.family': ”Times New Roman”})

RoughModel=f_reflect        (RoughModel)
fig=plt.figure()
fig, axarr = plt.subplots(2,1)
ax1,ax2=axarr
ax1.plot(RoughModel.Mesh.z_coords, RoughModel.T, 'b', linewidth=5,
  ↪ label='Current Model')
ax1.plot(zT_Parish[:,0], zT_Parish[:,1], 'go', markersize=10,
  ↪ label='Parish et al. (2016)')
ax1.legend(loc='best')
# ax1.set_xlabel('z')
ax1.set_ylabel('T')
ax1.grid()
#
ax2.plot(RoughModel.Mesh.z_coords[1:-1], RoughModel.BetaZ[1:-1], 'b',
  ↪ linewidth=5, label='Current Model')
ax2.plot(zBeta_Parish[:,0], zBeta_Parish[:,1], 'go', markersize=10,
  ↪ label='Parish et al. (2016)')
ax2.legend(loc='best')
ax2.set_xlabel('z')
ax2.set_ylabel(r'$\beta$ $(z)$')
ax2.grid()
fig.set_size_inches(fig.get_size_inches()*1.1*np.array([1.6,2.0]))
plt.savefig(”1D_Radiative_HT_Parish_Beta_Corrs.pdf”, bbox_inches =
  ↪ 'tight')
plt.show()
```

A.1.2. Version B: Build Explicit δ Source Term Corrections (Efficient)

```
import numpy as np
import matplotlib
from copy import deepcopy
plt=matplotlib.pyplot
pi=np.pi
sin=np.sin
exp=np.exp

# -------------------------------------------------------
# # 1-D Field Inversion Radiative Heat Transfer Problem
# -------------------------------------------------------

# -------------------------------------------------------
# # # # # Problem Replicated from:
# Eric J. Parish, Karthik Duraisamy,
# A paradigm for data-driven predictive modeling using field inversion and
  ↪ machine learning,
# Journal of Computational Physics,
# Volume 305,
# 2016,
# Pages 758-774,
# ISSN 0021-9991,
# https://doi.org/10.1016/j.jcp.2015.11.012.
# (http://www.sciencedirect.com/science/article/pii/S0021999115007524)
# Keywords: Data-driven modeling; Machine learning; Closure modeling
# -----------------------------------------------------

zT_Parish=np.array([[3.12523747e-02, 1.63138286e+01], [6.25036317e-02,
  ↪ 2.90723605e+01],
       [9.37515381e-02, 3.76641118e+01], [1.24996094e-01, 4.29329280e+01],
       [1.56253307e-01, 4.60092889e+01], [1.87500842e-01, 4.77622723e+01],
       [2.18752843e-01, 4.87502426e+01], [2.49997027e-01, 4.93031644e+01],
       [2.81252379e-01, 4.96118198e+01], [3.12499913e-01, 4.97837545e+01],
       [3.43744469e-01, 4.98791455e+01], [3.74999821e-01, 4.99326106e+01],
       [4.06249589e-01, 4.99612664e+01], [4.37500845e-01, 4.99772290e+01],
       [4.68745401e-01, 4.99849411e+01], [5.00000753e-01, 4.99864797e+01],
       [5.31248660e-01, 4.99849411e+01], [5.62496194e-01, 4.99772290e+01],
       [5.93748195e-01, 4.99612664e+01], [6.24996102e-01, 4.99326106e+01],
       [6.56253315e-01, 4.98791455e+01], [6.87497499e-01, 4.97837545e+01],
       [7.18749128e-01, 4.96122237e+01], [7.49989589e-01, 4.93031644e+01],
       [7.81250525e-01, 4.87506272e+01], [8.12498432e-01, 4.77626569e+01],
       [8.43744477e-01, 4.60092889e+01], [8.75001318e-01, 4.29333126e+01],
       [9.06245874e-01, 3.76648811e+01], [9.37497503e-01, 2.90716104e+01],
       [9.68741686e-01, 1.63126555e+01]])
#
zBeta_Parish=np.array([[0.03124516, 1.17785807], [0.06250632, 1.54141139],
       [0.09374702, 1.6069432 ], [0.12500110, 1.57388281],
       [0.15624851, 1.53103891], [0.18750261, 1.49959031],
       [0.21874702, 1.47980724], [0.24999851, 1.46810165],
       [0.28125633, 1.46138052], [0.31250261, 1.45759421],
       [0.34374703, 1.45545334], [0.37499852, 1.45426899],
       [0.40625261, 1.45361606], [0.43749666, 1.4532657 ],
       [0.4687541 , 1.45309742], [0.49999517, 1.45304357],
       [0.53125262, 1.45309742], [0.56249852, 1.4532657 ],
       [0.59375262, 1.45361606], [0.62500262, 1.45426899],
       [0.6562489 , 1.45545334], [0.68750262, 1.45759421],
       [0.7187448 , 1.46138052], [0.75000262, 1.46810131],
       [0.78124853, 1.47980724], [0.81250263, 1.49959031],
       [0.84374518, 1.53103891], [0.8749989 , 1.57388281],
       [0.90625263, 1.6069432 ], [0.93749481, 1.54139086],
       [0.96875635, 1.17781769]])
#
class Class_Mesh(): pass
class Class_Model(): pass
# Create Mesh
theMesh             = Class_Mesh()
theMesh.N_points    = N_points = 30
theMesh.z_coords    = z_coords = np.linspace(0.,0.5,N_points)
theMesh.inds_z_cond            = [0]
__ones=np.ones(z_coords.shape)
theMesh.G2_matrix              = -2*np.diag(__ones) +
  ↪ np.diag(__ones[1:],1)+ np.diag(__ones[1:],-1)
theMesh.G2_matrix              /= np.mean(np.diff(z_coords))**2
theMesh.G2_matrix [ 0,:] *=0.
# theMesh.G2_matrix [-1,:] *=0.
theMesh.G2_matrix [-1,-2]*=2.
# Create Solver
def f_solve_Model(selfModel):
    #
    # Generic Equation
    # G2*T =  e(T) * (T^4-T_inf^4) + h * (T-T_inf) + DeltaZ
    # (beta and h_conv may be omitted)

    #  e(T)  *  (T_inf^4)  + h  *  (T_inf)  - DeltaZ =  e(T)  *  (T^4)  + h  *  (T)  -
      ↪  G2*T
    # A =   e(T)  *  (T^3)  + h - G2
    # b =  e(T)  *  (T_inf^4)  + h  *  (T_inf)  - DeltaZ
    h_conv = selfModel.h_conv
    T_inf  = selfModel.T_inf
    DeltaZ=selfModel.DeltaZ

    i_iter=0
    vkey=True
    N_iter_max=1000
    N_iter_min=20
    alpha=0.5
    while vkey:
        i_iter+=1
        eT=selfModel.epsilon_T(selfModel.T)
        A= np.diag(eT*(selfModel.T**3)+h_conv) - selfModel.Mesh.G2_matrix
        b = eT*(T_inf**4) + h_conv*T_inf - DeltaZ

        A=A[1:,:]; A=A[:,1:]
        b= b[1:]
        T_new= np.linalg.solve(A,b)
        T_old=selfModel.T[1:]
        if max(np.fabs(T_old-T_new))<1e-11: vkey=False
        if (i_iter>N_iter_max) : vkey=False ; print('Warning: Maximum
          ↪  Numbers of Iterations Reached')
        if (i_iter<N_iter_min) : vkey=True
        selfModel.T[1:] = (1-alpha)*T_old + alpha*T_new
    # print(i_iter)
    return selfModel
#

# Create True Model
TrueModel                   = Class_Model()
TrueModel.Mesh              = theMesh
TrueModel.h_conv            = 0.5
TrueModel.T_inf     = T_inf = 50.
TrueModel.epsilon_T         = lambda T:
  ↪ 1e-4*(1.+5*sin(3*pi/200.*T)+exp(0.02*T) )
TrueModel.DeltaZ             = np.zeros(z_coords.shape)
TrueModel.T                 = np.ones(z_coords.shape) ;
TrueModel.T[0]=0. ;
#
TrueModel=f_solve_Model(TrueModel)

#

# Create Rough Model
RoughModel           = Class_Model()
RoughModel.Mesh      = theMesh
RoughModel.h_conv    = 0.
RoughModel.T_inf     = T_inf
RoughModel.epsilon_T = lambda T: 5e-4
RoughModel.DeltaZ     = np.zeros(z_coords.shape)
RoughModel.T         = np.ones(z_coords.shape) ;
RoughModel.T[0]=0. ;
RoughModel=f_solve_Model(RoughModel)
#
BoolPlotIni=False
if BoolPlotIni:
    matplotlib.rcParams.update({'font.size': 18})
    matplotlib.rcParams.update({'font.family': ”Times New Roman”})
    plt.rcParams.update({'font.size': 18})
    plt.rcParams.update({'font.family': ”Times New Roman”})

    plt.plot(TrueModel.Mesh.z_coords, TrueModel.T, 'b', linewidth=3,
      ↪ label='True Model')
    plt.plot(RoughModel.Mesh.z_coords, RoughModel.T, 'r', linewidth=3,
      ↪ label='Rough Model')
    plt.plot(zT_Parish[:,0],zT_Parish[:,1],'go',markersize=6,label='Parish
      ↪ et al. (2016)')
    plt.legend(loc='best')
    plt.xlabel('z')
    plt.ylabel('T')
    plt.grid()
    plt.show()
#

RoughModel.Tstar = TrueModel.T
RoughModel.DJDB  = np.zeros(RoughModel.T.shape)
def f_get_Jacobian(selfModel):
    # Rough Model Equation
    # G2*T = e_0 * (T^4-T_inf^4) +deltaZ
    # RR =  e_0 * (T^4-T_inf^4) - G2*T +deltaZ
    # dRRdT = 4 * e_0 * (T^3) - G2
    # dRRdB = 1

    # JJ= sum((T - T*)**2)
    # dJJdT = 2*(T - T*)
    # dJJdB = 0.
    selfModel=f_solve_Model(selfModel)

    T=selfModel.T
    e0 = selfModel.epsilon_T(T)
    dRRdT = np.diag(4*e0* (T**3)) - selfModel.Mesh.G2_matrix
    dRRdB = 1.

    dJJdT = 2*(T- selfModel.Tstar)
    dJJdB = 0.

    dRRdT=dRRdT[1:,:] ; dRRdT=dRRdT[:,1:]

    dJJdT=dJJdT[1:]

    Psi=np.linalg.solve(dRRdT.transpose(),-dJJdT)
    DJDB = dRRdB * Psi + dJJdB
    # Discrete Adjoint Method
    selfModel.DJDB[1:] = DJDB
    selfModel.Jcost=np.sum((T- selfModel.Tstar)**2)
    return selfModel
```

```
#
RoughModel=f_get_Jacobian(RoughModel)

# Bold Drive Method with Added Momentum
kPlus  = 1.2
kMinus = 0.5
c_1    = 0.9
m_n    = np.zeros(RoughModel.DJDB.shape)
alpha  = 0.1 / np.fabs(RoughModel.DJDB).max()

N_iters_print = 200
iiter   =0
n_count =0
vkey=True
while vkey:
    iiter  +=1
    n_count+=1
    BackupModel = deepcopy(RoughModel)
    m_n = (c_1*m_n + (1-c_1)*RoughModel.DJDB)#/(1-c_1**n_count)
    RoughModel.DeltaZ -= alpha*m_n
    #
    RoughModel=f_get_Jacobian(RoughModel)
    if RoughModel.Jcost<BackupModel.Jcost:
        alpha*=kPlus
    else:
        alpha *=kMinus
        RoughModel = deepcopy(BackupModel)
        m_n     = np.copy(RoughModel.DJDB)
        # n_count=0
    if alpha<1e-14: vkey=False
    if iiter%N_iters_print==0: print('Iteration %8d: Jcost=%12.4e (alpha =
      ↪ %12.4e)' % (iiter,RoughModel.Jcost,alpha))
print('Iteration END: Jcost=%12.4e (alpha = %12.4e)' %
  ↪ (RoughModel.Jcost,alpha))
#
def f_get_BetaZ(selfModel):
    # beta*e0(DT4) = e0*DT4 + delta
    # beta = (e0*DT4 + delta)/(e0*DT4)
    DT4= selfModel.T**4 - selfModel.T_inf**4
    e0 = selfModel.epsilon_T(selfModel.T)
    selfModel.BetaZ = 1. + selfModel.DeltaZ/(e0*DT4)
    return selfModel
#
def f_get_delta_real(selfModel,TrueModel=TrueModel):
    T     = selfModel.T
    T_inf = selfModel.T_inf
    e0=RoughModel.epsilon_T(T)
    selfModel.DeltaZ_real = (TrueModel.epsilon_T(T)-e0)*(T**4-T_inf**4) +
      ↪ TrueModel.h_conv*(T-T_inf)
    #
    selfModel.BetaZ_real = TrueModel.epsilon_T(T)/e0 +
      ↪ TrueModel.h_conv*(T-T_inf) / (e0*(T**4-T_inf**4))
    return selfModel
#
def f_reflect(selfModel):
    try:
        if selfModel.has_reflected:
            print('Model already reflected, operation skipped!')
        else:
            raise Exception('Error: selfModel.has_reflected should be True
             ↪  or undefined')
    except:
        selfModel.Mesh.z_coords= np.append( selfModel.Mesh.z_coords,
          ↪  1.0-selfModel.Mesh.z_coords[::-1][1:])
        f_complete = lambda x:  np.append( x, x[::-1][1:])
        selfModel.T     = f_complete(selfModel.T)
        selfModel.DeltaZ= f_complete(selfModel.DeltaZ)
        selfModel.has_reflected=True
    return selfModel
#
RoughModel=f_reflect        (RoughModel)
RoughModel=f_get_BetaZ      (RoughModel)
RoughModel=f_get_delta_real (RoughModel)
#

matplotlib.rcParams.update({'font.size': 18})
matplotlib.rcParams.update({'font.family': ”Times New Roman”})
plt.rcParams.update({'font.size': 18})
plt.rcParams.update({'font.family': ”Times New Roman”})


fig=plt.figure()
fig, axarr = plt.subplots(2,1)
ax1,ax2=axarr
ax1.plot(RoughModel.Mesh.z_coords, RoughModel.T, 'b', linewidth=5,
  ↪  label='Current Model')
ax1.plot(zT_Parish[:,0], zT_Parish[:,1], 'go', markersize=10,
  ↪  label='Parish et al. (2016)')
ax1.legend(loc='best')
# ax1.set_xlabel('z')
ax1.set_ylabel('T')
ax1.grid()
#
ax2.plot(RoughModel.Mesh.z_coords[1:-1], RoughModel.BetaZ[1:-1], 'b',
  ↪  linewidth=5,  label='Current Model')
ax2.plot(zBeta_Parish[:,0],zBeta_Parish[:,1],'go',markersize=10,label=
  ↪  'Parish et al. (2016)')
ax2.legend(loc='best')
ax2.set_xlabel('z')
ax2.set_ylabel(r'$\beta$ $(z)$')
ax2.grid()
fig.set_size_inches(fig.get_size_inches()*1.1*np.array([1.6,2.0]))
plt.savefig(”1D_Radiative_HT_Parish_Delta_Corrs.pdf”, bbox_inches =
  ↪  'tight')
plt.show()
```

A.2. Recovery of a Dynamic Viscosity Profile

```
import numpy as np
import matplotlib
from copy import deepcopy
plt=matplotlib.pyplot
```

```
pi=np.pi
sin=np.sin
cos=np.cos
exp=np.exp
#
class Class_Mesh(): pass
class Class_Model(): pass
# Create Mesh
theMesh             = Class_Mesh()
theMesh.N_points    = N_points = 30
theMesh.y_coords    = y_coords = np.linspace(0.,0.5,N_points)
theMesh.inds_z_cond            = [0]
__ones=np.ones(y_coords.shape)
theMesh.G2_matrix              = -2*np.diag(__ones) +
  ↪ np.diag(__ones[1:],1)+ np.diag(__ones[1:],-1)
theMesh.G2_matrix              /= np.mean(np.diff(y_coords))**2
theMesh.G2_matrix [ 0,:] *=0.
theMesh.G2_matrix [-1,-2]*=2.

theMesh.G1_matrix_O1              = np.diag(__ones)+
  ↪ -np.diag(__ones[1:],-1)
theMesh.G1_matrix_O1             /= np.mean(np.diff(y_coords))
theMesh.G1_matrix_O1 [ 0,:] *=0.
theMesh.G1_matrix_O1 [-1,:]*=0.

theMesh.G1_matrix_O2        = np.diag(__ones[1:],1)+
  ↪ -np.diag(__ones[1:],-1)
theMesh.G1_matrix_O2       /= np.mean(np.diff(y_coords))
theMesh.G1_matrix_O2 [ 0,:] *=0.
theMesh.G1_matrix_O2 [-1,:]*=0.

theMesh.G1_matrix=theMesh.G1_matrix_O1
#
f_broadcast = lambda x: np.array(np.matrix(x).transpose())
# Create Solver
def f_solve_Model(selfModel):
    #
    # # # # # # # Generic Equation
    # # # # mu*(G2*u) + (G1*mu)*(G1*u) = -1
    # A= mu*G2 + (G1*mu)*G1
    # b= -1
    mu = selfModel.mu
    G1_matrix = selfModel.Mesh.G1_matrix
    G2_matrix = selfModel.Mesh.G2_matrix

    A =f_broadcast(mu)*G2_matrix + f_broadcast(G1_matrix @ mu)*G1_matrix
    b =-np.ones(mu.shape)

    selfModel.A_solveU=np.copy(A)
    A=A[1:,:]; A=A[:,1:]
    b= b[1:]

    selfModel.u[1:]=np.linalg.solve(A,b)
    return selfModel
#
# Create True Model
```

```
f_get_real_mu  = lambda x: 3-cos(2*pi*x)**2-cos(pi*x)**2
if np.fabs(f_get_real_mu(0)-1)>1e-12: raise Exception('Error: mu_w = 1 was
  ↪ expected!')
#
TrueModel                   = Class_Model()
TrueModel.Mesh              = theMesh
TrueModel.mu                = f_get_real_mu(TrueModel.Mesh.y_coords)
TrueModel.u                 = np.zeros(y_coords.shape) ;
TrueModel.u[0]=0. ;
TrueModel=f_solve_Model(TrueModel)
plt.plot(TrueModel.u );plt.show()
plt.plot(TrueModel.mu);plt.show()
#
# Create Rough Model
RoughModel                   = Class_Model()
RoughModel.Mesh              = theMesh
RoughModel.mu                = 1.3*np.ones(y_coords.shape) ;
RoughModel.u                 = np.zeros(y_coords.shape) ;
RoughModel.u[0]=0. ;
RoughModel.mu[0]=1. ;
RoughModel=f_solve_Model(RoughModel)
#
#

RoughModel.uStar = TrueModel.u
RoughModel.DJDB  = np.zeros(RoughModel.u.shape)
def f_get_Jacobian(selfModel):
    # Rough Model Equation
    # RR =   (mu*G2)*u  +  ((mu*G1)*G1)*u  - 1
    # dRRdW   = (mu*G2)+((mu*G1)*G1)
    # dRRdB   = G2*u  + ((u*G1)*G1)

    # JJ= sum((u - u*)**2)
    # dJJdW = 2*(u - u*)
    # dJJdB = 0.
    selfModel=f_solve_Model(selfModel)

    dRRdW = selfModel.A_solveU

    u  = selfModel.u
    mu = selfModel.mu
    G1_matrix = selfModel.Mesh.G1_matrix
    G2_matrix = selfModel.Mesh.G2_matrix

    dRRdB  =np.diag(G2_matrix @ u) + f_broadcast(G1_matrix @ u)*G1_matrix

    dJJdW  = 2*(u- selfModel.uStar)
    dJJdB  = 0.

    dRRdB=dRRdB[1:,:] ; dRRdB=dRRdB[:,1:]
    dRRdW=dRRdW[1:,:] ; dRRdW=dRRdW[:,1:]

    dJJdW=dJJdW[1:]

    # Discrete Adjoint Method
    Psi=np.linalg.solve(dRRdW.transpose(),-dJJdW)
    DJDB = Psi @ dRRdB   + dJJdB
    selfModel.DJDB[1:] = DJDB
    selfModel.Jcost=np.sum((u- selfModel.uStar)**2)
    return selfModel
#
RoughModel=f_get_Jacobian(RoughModel)

# Bold Drive Method with Added Momentum
kPlus  = 1.2
kMinus = 0.5
c_1    = 0.9
m_n    = np.zeros(RoughModel.DJDB.shape)
alpha  = 0.1 / np.fabs(RoughModel.DJDB).max()

N_iters_print = 1000
iiter   =0
vkey=True
while vkey:
    iiter  +=1
    BackupModel = deepcopy(RoughModel)
    m_n = (c_1*m_n + (1-c_1)*RoughModel.DJDB)
    RoughModel.mu -= alpha*m_n
    #
    RoughModel=f_get_Jacobian(RoughModel)
    if RoughModel.Jcost<BackupModel.Jcost:
        alpha*=kPlus
    else:
        alpha *=kMinus
        RoughModel = deepcopy(BackupModel)
        m_n    = np.copy(RoughModel.DJDB)
    if alpha<1e-14: vkey=False
    if iiter%N_iters_print==0: print('Iteration %8d: Jcost=%12.4e (alpha =
      ↪ %12.4e)' % (iiter,RoughModel.Jcost,alpha))
print('Iteration END: Jcost=%12.4e (alpha = %12.4e)' %
  ↪ (RoughModel.Jcost,alpha))
#

matplotlib.rcParams.update({'font.size': 18})
matplotlib.rcParams.update({'font.family': ”Times New Roman”})
plt.rcParams.update({'font.size': 18})
plt.rcParams.update({'font.family': ”Times New Roman”})

f_reflect_coords = lambda x,ind_first=1: np.append( x,
  ↪ 1.0-x[::-1][ind_first:])+0.
f_reflect_prop   = lambda x,ind_first=1:  np.append( x,
  ↪ x[::-1][ind_first:])+0.

rough_y_coords = f_reflect_coords(RoughModel.Mesh.y_coords)
rough_mu       = f_reflect_prop  (RoughModel.mu)
rough_u        = f_reflect_prop  (RoughModel.u )

true_y_coords = f_reflect_coords(TrueModel.Mesh.y_coords[::2]
  ↪ ,ind_first=0)
true_mu       = f_reflect_prop  (TrueModel.mu           [::2]
  ↪ ,ind_first=0)
```

```
true_u          = f_reflect_prop   (TrueModel.u            [::2]
  ↪  ,ind_first=0)

fig=plt.figure()
fig, axarr = plt.subplots(2,1)
ax1,ax2=axarr

ax1.plot(rough_y_coords,rough_u,'b',linewidth=5,label='Field Inversion')
ax1.plot(true_y_coords ,true_u ,'go',markersize=10,label='True Model')
ax1.legend(loc='best')
ax1.set_ylabel('u')
ax1.grid()
#
ax2.plot(rough_y_coords,rough_mu,'b',linewidth=5,label='Field Inversion')
ax2.plot(true_y_coords ,true_mu ,'go',markersize=10,label='True Model')
ax2.legend(loc='best')
ax2.set_xlabel('y')
ax2.set_ylabel(r'$\mu$ $(y)$')
ax2.grid()
fig.set_size_inches(fig.get_size_inches()*1.1*np.array([1.6,2.0]))
plt.savefig("Laminar_Channel_mu_recovered.pdf", bbox_inches = 'tight')
plt.show()
```

A.3. Field Inversion Optimization for the $k-\omega$ Turbulence Model

A.3.1. Sympy Code to Derive the Matrices Required by the Discrete Adjoint Method

```
%reset -f
from sympy import init_printing
init_printing()
from IPython.display import display
from sympy import
  ↪  Function,diff,Eq,dsolve,symbols,exp,solve,sin,Matrix,Array,
  ↪  sympify,sqrt,integrate,pi,cos,sin,ln,limit,oo,nsolve
from sympy.tensor import MutableDenseNDimArray
# ****************************************************
# # # Initialize variables employed in the code (mandatory step)
# ****************************************************
var_symbols='u,nu_lam,C_mu,nu_T,k,omega,y,alpha_om,sigma_om,
  ↪  alpha_k,sigma_k,gamma,sigma_d,beta'
exec(var_symbols+'=symbols("'+var_symbols+'")', globals())
var_symbols='u_C,u_N,u_S,k_C,k_N,k_S,omega_C,omega_N,omega_S,cNd2,cCd2,
  ↪  cSd2,cNd1,cCd1,cSd1,beta_C'
exec(var_symbols+'=symbols("'+var_symbols+'")', globals())
var_symbols='vDISCR_d2nutdy2,vDISCR_dnutdy,vDISCR_d2omdy2,vDISCR_domdy,
  ↪  vDISCR_d2kdy2,vDISCR_dkdy,vDISCR_d2udy2,vDISCR_dudy,vnutC'
exec(var_symbols+'=symbols("'+var_symbols+'")', globals())
# ****************************************************
# # # Build main variables
# ****************************************************

uC   =u_C
uN   =u_N
uS   =u_S

kC   =k_C
```

```
kN   =k_N
kS   =k_S

omC  =omega_C
omN  =omega_N
omS  =omega_S

bC=beta_C

om=omega
u=u(y)
k=k(y)
om=om(y)
beta=beta(y)

nu_T=nu_T(y)#C_mu*k/om

nutN=C_mu*kN/omN
nutC=C_mu*kC/omC
nutS=C_mu*kS/omS

# Express discretized derivatives as sums of values and coefficients
DISCR_d2udy2    =cNd2* uN+cCd2* uC+cSd2* uS
DISCR_d2kdy2    =cNd2* kN+cCd2* kC+cSd2* kS
DISCR_d2omdy2   =cNd2*omN+cCd2*omC+cSd2*omS

DISCR_dnutdy   = cNd1* nutN+cCd1* nutC+cSd1* nutS
DISCR_d2nutdy2 = cNd2* nutN+cCd2* nutC+cSd2* nutS

DISCR_d2BetaDy2 =0

DISCR_dudy   =cNd1* uN+cCd1* uC+cSd1* uS
DISCR_dkdy   =cNd1* kN+cCd1* kC+cSd1* kS
DISCR_domdy  =cNd1*omN+cCd1*omC+cSd1*omS
DISCR_BetaDy =0

# Build Effective Viscosities beta
nuEffU =nu_lam+nu_T
nueffK =nu_lam+nu_T*sigma_k /C_mu
nueffOm=nu_lam+nu_T*sigma_om/C_mu

# ***************************************************
# # # Build residual eqs.
# ***************************************************
#
#   U-eq:  0 = ddy[(nu+nut)dudy]
R_U =                                                ( nuEffU *(u.diff(y)
  ↪ ) ).diff(y)
#
#   k-eq:  0 =  beta*nut  *(du/dy)^2 - alpha_k*k*om
#              + ddy[(nu+nut*sigma_k /C_mu)dkdy]
R_k =beta*nu_T *((u.diff(y))**2)  - alpha_k *k *om + ( nueffK *(k.diff(y)
  ↪ ) ).diff(y)
#
#   om-eq: 0 = gamma*(du/dy)^2 - alpha_om*om^2
#              + ddy[(nu+nut*sigma_om/C_mu)domdy] + dkdy*domdy*sigma_d/om
```

```
R_Om=gamma*((u.diff(y))**2)       - alpha_om*om*om + (
  ↪  nueffOm*(om.diff(y)) ).diff(y)  +  (k.diff(y)*om.diff(y))*sigma_d/om
#
# ***************************************************
# # # Build discretizations for R_i....
# ***************************************************
def f_discretize(i):
    # Substitute values by discretized approximations
    i=i.subs(u.diff(y).diff(y),DISCR_d2udy2)
    i=i.subs(u.diff(y),DISCR_dudy)
    i=i.subs(u,uC)

    i=i.subs(k.diff(y).diff(y),DISCR_d2kdy2)
    i=i.subs(k.diff(y),DISCR_dkdy)
    i=i.subs(k,kC)

    i=i.subs(om.diff(y).diff(y),DISCR_d2omdy2)
    i=i.subs(om.diff(y),DISCR_domdy)
    i=i.subs(om,omC)

    i=i.subs(nu_T.diff(y).diff(y),DISCR_d2nutdy2)
    i=i.subs(nu_T.diff(y)        ,DISCR_dnutdy  )
    i=i.subs(nu_T,nutC)

    i=i.subs(beta.diff(y).diff(y),DISCR_d2BetaDy2)
    i=i.subs(beta.diff(y),DISCR_BetaDy)
    i=i.subs(beta,bC)
    return i
# Discretize equations
R_Udiscr =f_discretize(R_U )
R_kdiscr =f_discretize(R_k )
R_Omdiscr=f_discretize(R_Om)

# # # # Build the following required residual matrices:
# R,ui
# R,bi
# # # # Initialize residual vector and general variables
dimRn  =3
dimWvar=9
dim_Bvar=1
Rn_discr=MutableDenseNDimArray([R_Udiscr,R_kdiscr,R_Omdiscr],(dimRn,1))
wVar=[uS,uC,uN,kS,kC,kN,omS,omC,omN] # Vector of DOF variables (u,k,om)
bVar=[bC] # Vector of ”beta” coefficients (can be larger!!)

# Rn_discr_wi
Rn_discr_wi=MutableDenseNDimArray([0 for i in
  ↪  range(dimRn*dimWvar)],(dimRn,dimWvar))
for i in range(dimRn):
    for j in range(dimWvar):
        Rn_discr_wi[i,j]= Rn_discr[i].diff(wVar[j])
#
# Rn_discr_bi
Rn_discr_bi=MutableDenseNDimArray([0 for i in
  ↪  range(dimRn*dim_Bvar)],(dimRn,dim_Bvar))
for i in range(dimRn):
    for j in range(dim_Bvar):
        Rn_discr_bi[i,j]= Rn_discr[i].diff(bVar[j])
#
def f_removeGradsLaplas(i):
    i=i.replace(str(DISCR_d2udy2),'vDISCR_d2udy2')
    i=i.replace(str(DISCR_dudy  ),'vDISCR_dudy')

    i=i.replace(str(DISCR_d2kdy2),'vDISCR_d2kdy2')
    i=i.replace(str(DISCR_dkdy  ),'vDISCR_dkdy')

    i=i.replace(str(DISCR_d2omdy2),'vDISCR_d2omdy2')
    i=i.replace(str(DISCR_domdy  ),'vDISCR_domdy')

    i=i.replace(str(DISCR_d2nutdy2),'vDISCR_d2nutdy2')
    i=i.replace(str(DISCR_dnutdy  ),'vDISCR_dnutdy')
    i=i.replace(str(nutC  ),'vnutC')
    i=i.replace('**1.0','')
    return i
#
def fstrM(xx):
    vShape=xx.shape
    if len(vShape)==1:
        xxStr='np.array(['
        for j in range(vShape[0]):
            xxStr+='(' + str(xx[j]) + '),'
        xxStr+=']);'
    elif len(vShape)==2:
        xxStr='np.array(['
        for i in range(vShape[0]):
            xxStr+='['
            for j in range(vShape[1]):
                xxStr+='(' + str(xx[i,j]) + '),'
            xxStr+='],'
        xxStr+=']);'
    else:
        raise ValueError('fstrM: Matrix out of shape!!',vShape, str(xx))
    xxStr=xxStr
    return xxStr
print('')
print('# PYTHON CODE....')
print('');print('vdRn_wi = %s' % f_removeGradsLaplas(fstrM(Rn_discr_wi)));
print('');print('vdRn_bi = %s' % f_removeGradsLaplas(fstrM(Rn_discr_bi)));
```

A.3.2. Main Routine for the Field Inversion Optimizer

```
# %reset -f
import numpy as np
import matplotlib
import time
import datetime
from copy import deepcopy
from scipy.interpolate import splrep,splev
plt   = matplotlib.pyplot
array = np.array
pi    = np.pi
sin   = np.sin
exp   = np.exp
myprint=print
```

```
# -----------------------------------------------------
# # Optimization of the k-Omega Turbulence Model
# -----------------------------------------------------
#
# -----------------------------------------------------
# # # # Source:
# Eric J. Parish, Karthik Duraisamy,
# A paradigm for data-driven predictive modeling using field inversion and
  ↪ machine learning,
# Journal of Computational Physics,
# Volume 305,
# 2016,
# Pages 758-774,
# ISSN 0021-9991,
# https://doi.org/10.1016/j.jcp.2015.11.012.
# (http://www.sciencedirect.com/science/article/pii/S0021999115007524)
# Keywords: Data-driven modeling; Machine learning; Closure modeling
# -----------------------------------------------------

# Matlab tic-toc
def tic():
    global GlobalTimeTic
    GlobalTimeTic=time.time()
def toc(BoolPrint=1):
    vToc=time.time()-GlobalTimeTic
    if BoolPrint: myprint( ”Elapsed Time: %.0f s   ( %.3f h ) ” %
      ↪ (vToc,vToc/3600.) )
    return vToc
tic()
class Class_Mesh (): pass
class Class_Model(): pass
class Class_DNS  (): pass
###  ---------------------------------------------
# Read Jimenez DNS Data
# File   : Re950.prof
# Website:
  ↪ torroja.dmt.upm.es/channels/data/statistics/Re950/profiles/Re950.prof
###  ---------------------------------------------
Filename_DNS_data= 'Re950.prof'
def f_read_DNS_data(Filename_DNS_data):
    with open(Filename_DNS_data,'r') as f_read:
        DNS_Text=[i.strip('\n').strip('\r') for i in f_read.readlines()]
    i_Header= [i for i in range(len(DNS_Text)) if '       y/h
      ↪ y+                U+    ' in DNS_Text[i]]
    if len(i_Header)>1: raise Exception('Multiple Headers Found',
      ↪ i_Header)
    f_weed_void= lambda x: [i for i in x if i!='']
    i_Header=i_Header[0]
    v_Headers= f_weed_void(DNS_Text[i_Header].strip('%').split(' '))
    Dict_Headers=dict(zip(v_Headers, range(len(v_Headers)) ))
    DNS_array = [f_weed_void(DNS_Text[i].split(' ')) for i in
      ↪ range(len(DNS_Text)) if ( (i>i_Header) and not ('%' in
      ↪ DNS_Text[i]) )]
    DNS_array = np.array([list(map(float,i)) for i in DNS_array])
    y_Plus_DNS=DNS_array[:,Dict_Headers['y+']]
    U_Plus_DNS=DNS_array[:,Dict_Headers['U+']]
    ReT = [i for i in DNS_Text if ('Re_{\tau}' in i) or (r'Re_{\tau}' in
      ↪  i)]
    if len(ReT)>1: raise Exception('Multiple Re_tau Entries Found', ReT)
    ReT = float(ReT[0].split('tau}')[1].split('=')[1])
    nu = 1./ReT
    vDNSdata=Class_DNS()
    vDNSdata.y_Plus   = y_Plus_DNS
    vDNSdata.U_Plus   = U_Plus_DNS
    vDNSdata.y_coords = y_Plus_DNS*nu
    vDNSdata.nu       = nu
    vDNSdata.ReT      = ReT
    return vDNSdata
#
DNSdata=f_read_DNS_data(Filename_DNS_data)
#
# Data Points from Parish & Duraisamy (2016)
# ---------------------------------------------------------------------
# # # # Source:
# Eric J. Parish, Karthik Duraisamy,
# A paradigm for data-driven predictive modeling using field inversion and
  ↪  machine learning,
# Journal of Computational Physics,
# Volume 305,
# 2016,
# Pages 758-774,
# ISSN 0021-9991,
# https://doi.org/10.1016/j.jcp.2015.11.012.
# (http://www.sciencedirect.com/science/article/pii/S0021999115007524)
# Keywords: Data-driven modeling; Machine learning; Closure modeling
# ---------------------------------------------------------------------
#
YplusBeta_Parish=array([[4.93197382e-02, 9.99989846e-01],
       [1.86249468e+00, 1.00076551e+00], [2.20396226e+00, 1.00313332e+00],
         ↪  [2.45046687e+00, 1.00741988e+00], [2.57982455e+00,
         ↪  1.01119614e+00],
       [2.71353911e+00, 1.01666661e+00], [2.85144245e+00, 1.02444365e+00],
         ↪  [2.99393005e+00, 1.03534376e+00], [3.14099469e+00,
         ↪  1.05040796e+00],
       [3.29295051e+00, 1.07086097e+00], [3.44981381e+00, 1.09805001e+00],
         ↪  [3.61195635e+00, 1.13344474e+00], [3.77923369e+00,
         ↪  1.17853527e+00],
       [3.95205842e+00, 1.23450550e+00], [4.13048755e+00, 1.30202902e+00],
         ↪  [4.31457112e+00, 1.38086090e+00], [4.50480748e+00,
         ↪  1.46953145e+00],
       [5.11387039e+00, 1.75513882e+00], [5.33017432e+00, 1.83586903e+00],
         ↪  [5.55366057e+00, 1.89641158e+00], [5.78446874e+00,
         ↪  1.92925480e+00],
       [6.02273630e+00, 1.92884655e+00], [6.26859834e+00, 1.89204337e+00],
         ↪  [6.52284711e+00, 1.81868196e+00], [6.78534833e+00,
         ↪  1.71184456e+00],
       [7.05627159e+00, 1.57759359e+00], [7.33615654e+00, 1.42360403e+00],
         ↪  [8.23169855e+00, 9.34977021e-01], [8.54998533e+00,
         ↪  7.92704057e-01],
       [8.87833325e+00, 6.72925326e-01], [9.21789190e+00, 5.80376456e-01],
         ↪ [9.56801704e+00, 5.17261965e-01], [9.92993409e+00,
         ↪ 4.84153393e-01],
       [1.03039772e+01, 4.79519826e-01], [1.06899467e+01, 4.99217589e-01],
         ↪ [1.10881302e+01, 5.38388582e-01], [1.14999820e+01,
         ↪ 5.91031620e-01],
       [1.28166443e+01, 7.68107368e-01], [1.32846330e+01, 8.16953739e-01],
         ↪ [1.37683169e+01, 8.56083907e-01], [1.42674462e+01,
         ↪ 8.83864897e-01],
       [1.47824265e+01, 8.99745581e-01], [1.53152202e+01, 9.05318109e-01],
         ↪ [1.58648095e+01, 9.02113395e-01], [1.64324583e+01,
         ↪ 8.92458429e-01],
       [1.70186960e+01, 8.78374018e-01], [1.88944343e+01, 8.29935891e-01],
         ↪ [1.95605875e+01, 8.18280531e-01], [2.02492026e+01,
         ↪ 8.11054616e-01],
       [2.09599393e+01, 8.08992984e-01], [2.16934278e+01, 8.12524293e-01],
         ↪ [2.24514489e+01, 8.21791428e-01], [2.32347818e+01,
         ↪ 8.36733151e-01],
       [2.40430127e+01, 8.57165753e-01], [2.48780999e+01, 8.82742227e-01],
         ↪ [2.57395880e+01, 9.12931856e-01], [2.66295611e+01,
         ↪ 9.47142686e-01],
       [2.84985615e+01, 1.02501519e+00], [3.04910254e+01, 1.11082804e+00],
         ↪ [3.48824192e+01, 1.28578092e+00], [3.72985607e+01,
         ↪ 1.36632742e+00],
       [3.98760054e+01, 1.43793338e+00], [4.12276405e+01, 1.46973557e+00],
         ↪ [4.26229345e+01, 1.49857800e+00], [4.40654504e+01,
         ↪ 1.52429736e+00],
       [4.55544820e+01, 1.54683242e+00], [4.70914479e+01, 1.56606070e+00],
         ↪ [4.86778072e+01, 1.58202303e+00], [5.03176059e+01,
         ↪ 1.59457653e+00],
       [5.20126441e+01, 1.60384366e+00], [5.37593439e+01, 1.60982443e+00],
         ↪ [5.55647016e+01, 1.61260049e+00], [5.74306873e+01,
         ↪ 1.61221266e+00],
       [5.93563344e+01, 1.60882424e+00], [6.13434454e+01, 1.60257811e+00],
         ↪ [6.33970801e+01, 1.59351509e+00], [6.55161517e+01,
         ↪ 1.58190056e+00],
       [6.77060540e+01, 1.56783656e+00], [6.99691546e+01, 1.55154763e+00],
         ↪ [7.23042428e+01, 1.53315625e+00], [7.72029967e+01,
         ↪ 1.49112755e+00],
       [8.24294811e+01, 1.44346509e+00], [9.09285483e+01, 1.36528640e+00],
         ↪ [1.07045557e+02, 1.23252551e+00], [1.14257639e+02,
         ↪ 1.18333214e+00],
       [1.21943291e+02, 1.13822120e+00], [1.30139342e+02, 1.09800918e+00],
         ↪ [1.38879242e+02, 1.06314516e+00], [1.48206096e+02,
         ↪ 1.03362914e+00],
       [1.58143323e+02, 1.00913451e+00], [1.68746841e+02, 9.88844790e-01],
         ↪ [1.80052219e+02, 9.71637314e-01], [1.98436129e+02,
         ↪ 9.49020608e-01],
       [2.74237804e+02, 8.77455470e-01], [2.92551465e+02, 8.67433084e-01],
         ↪ [3.02161936e+02, 8.64044661e-01], [3.12072330e+02,
         ↪ 8.61962617e-01],
       [3.22324071e+02, 8.61146130e-01], [3.32895748e+02, 8.61554374e-01],
         ↪ [3.55108636e+02, 8.65208156e-01], [3.91208031e+02,
         ↪ 8.72719842e-01],
       [4.04038984e+02, 8.73515917e-01], [4.17290771e+02, 8.72127888e-01],
         ↪  [4.30977193e+02, 8.67616794e-01], [4.45089992e+02,
         ↪  8.58900789e-01],
       [4.59688182e+02, 8.44898027e-01], [4.74741152e+02, 8.24526661e-01],
         ↪  [4.90311849e+02, 7.96908968e-01], [5.06367623e+02,
         ↪  7.61432582e-01],
       [5.22975614e+02, 7.18015854e-01], [5.40100998e+02, 6.67291561e-01],
         ↪  [5.76052495e+02, 5.51268674e-01], [5.94946030e+02,
         ↪  4.92297857e-01],
       [6.14428160e+02, 4.39022041e-01], [6.34548253e+02, 3.97381173e-01],
         ↪  [6.55327200e+02, 3.73682621e-01], [6.76786575e+02,
         ↪  3.73458086e-01],
       [6.98913306e+02, 3.99953109e-01], [7.21799953e+02, 4.53208513e-01],
         ↪  [7.45436048e+02, 5.29448042e-01], [7.95055549e+02,
         ↪  7.19260997e-01],
       [8.21048942e+02, 8.12585530e-01], [8.47935049e+02, 8.92213483e-01],
         ↪  [8.75701570e+02, 9.51919139e-01], [9.04789142e+02,
         ↪  9.68800020e-01],
       [9.33944873e+02, 9.99989846e-01]])
# # -----------------------------------------------------


f_arg_unique=lambda x: np.unique(x,return_index=True)[1]
def f_build_interp(X_target,X0_data,Y0_data):
    # Build Bezier interpolation curve :)
    vsort=np.argsort(X0_data)
    X_data=X0_data[vsort]
    Y_data=Y0_data[vsort]
    X_data = np.append(X_data,2-np.flipud(X_data))
    Y_data = np.append(Y_data,  np.flipud(Y_data))

    vfilter=f_arg_unique(X_data)
    X_data = X_data[vfilter]
    Y_data = Y_data[vfilter]

    f_fit = splrep(X_data, Y_data)
    Y_target = splev(X_target, f_fit)
    #
    return Y_target
#
#
# # -----------------------------------------------------
# Create Mesh
# # -----------------------------------------------------
YplusFirst = 0.045
YplusEnd   = 1./DNSdata.nu
MESH                  = Class_Model()
MESH.Grid             = Class_Mesh()
MESH.Grid.N_points    = N_points = 100
MESH.Grid.y_Plus      = y_Plus   =
 ↪  10**np.linspace(np.log10(YplusFirst),np.log10(YplusEnd),N_points)
MESH.y        = y_coords = y_Plus*DNSdata.nu
MESH.N_points = N_points

#
# Finite Difference Coefficients:
```

```
fCoeff_df_dy_Center = lambda dXN,dXS :  (dXN - dXS)/(dXN*dXS);
fCoeff_df_dy_North  = lambda dXN,dXS :  dXS/(dXN*(dXN + dXS));
fCoeff_df_dy_South  = lambda dXN,dXS :  -dXN/(dXS*(dXN + dXS));
fCoeff_d2f_dy_Center= lambda dXN,dXS :  -2./(dXN*dXS);
fCoeff_d2f_dy_North = lambda dXN,dXS :  2./(dXN*(dXN + dXS));
fCoeff_d2f_dy_South = lambda dXN,dXS :  2./(dXS*(dXN + dXS));
#
MESH.Grid.G1_matrix=np.zeros((N_points,N_points))
MESH.Grid.G2_matrix=np.zeros((N_points,N_points))
MESH.Grid.trapz_int=np.zeros((N_points,N_points))
for ii in range(N_points):
    indC=ii;
    indN=ii+1;
    indS=ii-1;
    if ii==(N_points-1):
        indN=ii-1; # Use point from the south (repeated), since the
          ↪  channel is symmetric
    if ii==0:
        indS=ii+2; # Use one further point to the north (interpolate from
          ↪  interior)
    dYN=abs( MESH.y[indN]- MESH.y[indC] ); # must be ”abs” due to the
      ↪  symmetry (at i=n-1)
    dYS=( MESH.y[indC]- MESH.y[indS] );     # must have a sign (at i=1)
    #
    MESH.Grid.G2_matrix[ii,indC]+=fCoeff_d2f_dy_Center(dYN,dYS);
    MESH.Grid.G2_matrix[ii,indN]+=fCoeff_d2f_dy_North (dYN,dYS);
    MESH.Grid.G2_matrix[ii,indS]+=fCoeff_d2f_dy_South (dYN,dYS);
    #
    MESH.Grid.G1_matrix[ii,indC]+=fCoeff_df_dy_Center(dYN,dYS);
    MESH.Grid.G1_matrix[ii,indN]+=fCoeff_df_dy_North (dYN,dYS);
    MESH.Grid.G1_matrix[ii,indS]+=fCoeff_df_dy_South (dYN,dYS);
    #
for ii in range(1,N_points):
    dY = MESH.y[ii]-MESH.y[ii-1]
    MESH.Grid.trapz_int[ii:,ii-1]+=dY/2.
    MESH.Grid.trapz_int[ii:,ii]+=dY/2.
#
# # -----------------------------------------------------
# Mesh Created
# # -----------------------------------------------------

MESH.kOmConstants = Class_Model()
MESH.kOmConstants.C_mu     =1.0;
MESH.kOmConstants.alpha_k  =0.09;
MESH.kOmConstants.sigma_k  =0.6;
MESH.kOmConstants.alpha_om =0.0708
MESH.kOmConstants.sigma_om =0.5;
MESH.kOmConstants.gamma    =13./25.
MESH.kOmConstants.sigma_d  =0.#
MESH.kOmConstants.beta_1   = 0.075;

MESH.kOmConstants.underrelaxK  = 0.9;
MESH.kOmConstants.underrelaxOm = 0.9;
MESH.kOmConstants.indsKcond  =[0]
MESH.kOmConstants.indsKiter
  ↪  =np.setdiff1d(range(len(MESH.y)),MESH.kOmConstants.indsKcond)
```

```
MESH.kOmConstants.indsOmcond = MESH.kOmConstants.indsKcond
MESH.kOmConstants.indsOmiter = MESH.kOmConstants.indsKiter

MESH.FieldVars       =Class_Model()
MESH.FieldVars.nu    =np.copy( DNSdata.nu     );
MESH.FieldVars.u_DNS
 ↪ =f_build_interp(MESH.y,DNSdata.y_coords,DNSdata.U_Plus)#np.copy(
 ↪ DNSdata.U_Plus ); f_build_interp(X_target,X0_data,Y0_data):
MESH.FieldVars.u     = MESH.y*0.
MESH.FieldVars.k     = MESH.y*0. + 0.1 ;
MESH.FieldVars.Om    = MESH.y*0. + 1. ;
MESH.FieldVars.BetaY = MESH.y*0. + 1. ;
MESH.tol_solver=1e-11
# Create Solver
#
f_broadcast = lambda x: np.array(np.matrix(x).transpose())
maxAbs = lambda x: abs(x).max()
#
f_Mult_Ab          = lambda xA,yb:  np.array(    np.matrix(xA)*(
 ↪ np.matrix(yb.flatten()).transpose()    )    )[:,0]
def f_solveEq(A,b,x,v_inds_iter,v_inds_BC,omegaUnderRelax=1):
    b-= f_Mult_Ab(A[:,v_inds_BC],x[v_inds_BC])
    A=A[v_inds_iter,:]; A=A[:,v_inds_iter];
    b=b[v_inds_iter];
    if omegaUnderRelax<1:
        b     +=
          ↪ (1-omegaUnderRelax)/omegaUnderRelax*np.diag(A)*x[v_inds_iter];
          ↪ # b     = b + (1-omega)/omega*diag(A).*x(2:n);
        A = A - np.diag(np.diag(A)) + np.diag(np.diag(A))/omegaUnderRelax
          ↪ # A(logical(eye(size(A))))=diag(A)/omega;
    x[v_inds_iter]=np.linalg.solve(A,b)
    return x
def f_solve_U(self):
    # Nabla * ( mu_eff* Nabla(u)) -1
    # mu_eff*GradL_U=(1 - y/H)
    mu_eff = self.FieldVars.nu + self.FieldVars.nut # )*1. (rho)
    Grad_U = (1-self.y)/mu_eff
    self.FieldVars.u = self.Grid.trapz_int @ Grad_U
    return self
#
def f_solve_KOm(self):
    # %   k-eq:  0 = beta* nut  *(du/dy)^2 - alpha_k*k*om
    # %              + ddy[(nu+nut*sigma_k /C_mu)dkdy]
    # %   om-eq: 0 = gamma*(du/dy)^2 - alpha_om*om^2
    # %              + ddy[(nu+nut*sigma_om/C_mu)domdy]
    C_mu          = self.kOmConstants.C_mu;
    sigma_om      = self.kOmConstants.sigma_om;
    alpha_om      = self.kOmConstants.alpha_om;
    gamma         = self.kOmConstants.gamma;
    sigma_d       = self.kOmConstants.sigma_d;
    beta_1        = self.kOmConstants.beta_1;
    sigma_k       = self.kOmConstants.sigma_k;
    alpha_k       = self.kOmConstants.alpha_k;
    #
    nu =  self.FieldVars.nu
    BetaY         = self.FieldVars.BetaY
```

```
#
underrelaxK  = self.kOmConstants.underrelaxK
underrelaxOm = self.kOmConstants.underrelaxOm

indsKcond    = self.kOmConstants.indsKcond
indsKiter    = self.kOmConstants.indsKiter
indsOmcond   = self.kOmConstants.indsOmcond
indsOmiter   = self.kOmConstants.indsOmiter
#
wallDist      = self.y
#
G1_mat = self.Grid.G1_matrix
G2_mat = self.Grid.G2_matrix

u  = self.FieldVars.u
k  = self.FieldVars.k
Om = self.FieldVars.Om
dkdy  = G1_mat @ k
dOmdy = G1_mat @ Om
dUdy  = G1_mat @ u

# strMag = np.fabs(dUdy)
nut = C_mu*k/Om
nut = np.minimum(np.maximum(nut,0.),100.)

# %   om-eq: 0 = gamma*(du/dy)^2 - alpha_om*om^2
# %                + ddy[(nu+nut*sigma_om/C_mu)domdy]
nu_eff_om = nu + nut*sigma_om/C_mu
A_om = f_broadcast(nu_eff_om)*G2_mat + f_broadcast(G1_mat @ nu_eff_om
  ↪ )*G1_mat
A_om-= np.diag(alpha_om*Om)

gradKgradOm = dkdy*dOmdy
gradKgradOm[gradKgradOm<0]=0.
b_om = -gamma*(dUdy**2) - sigma_d/Om* gradKgradOm

Om[0]=60.*nu/beta_1/(wallDist[1]**2)
Om = f_solveEq(A_om,b_om,Om,indsOmiter,indsOmcond,underrelaxOm);

# %   k-eq:  0 = beta* nut  *(du/dy)^2 - alpha_k*k*om
# %                + ddy[(nu+nut*sigma_k /C_mu)dkdy]
nu_eff_k = nu + nut*sigma_k/C_mu
A_k = f_broadcast(nu_eff_k)*G2_mat + f_broadcast(G1_mat @ nu_eff_k
  ↪ )*G1_mat
A_k-= np.diag(alpha_k*Om)

b_k=-BetaY*nut*(dUdy**2)

k[0]= 0.
k = f_solveEq(A_k,b_k,k,indsKiter,indsKcond,underrelaxK);

nut = C_mu*k/Om
self.FieldVars.k  = k
self.FieldVars.Om = Om
self.FieldVars.nut= nut
return self
```

```
#
def f_CFDsolve(self,BoolPrintProgress=0):
    n=self.N_points;
    nmax  = 40000;   nmin=5;
    tol=self.tol_solver
    # tol  = 1e-10;   # iteration limits
    nResid= 25;     # steps to print the residual
    residual = 1e20; iter = 0;

    while ( (residual > tol) and (iter<nmax) ) or (iter<=nmin) :
        u_old=np.copy(self.FieldVars.u)
        self=f_solve_KOm(self);
        self=f_solve_U (self);
        residual = np.linalg.norm(self.FieldVars.u-u_old);
        if ( (iter % nResid) == 0) and BoolPrintProgress:
            myprint('%d    %12.6e' % (iter, residual) );
        iter += 1;
    if BoolPrintProgress:
        myprint('%d    %12.6e' % (iter, residual) );
    self.Niters_CFDsolve=iter
    return self
#
MESH=f_CFDsolve(MESH,1)
#
# Solver Validated!
#
MESH.Optim=Class_Model()
MESH.Optim.DJDB=MESH.FieldVars.BetaY*0.
MESH.Optim.IndsWcond=[0,N_points,2*N_points]
MESH.Optim.IndsWkeep=np.setdiff1d(range(3*N_points),MESH.Optim.IndsWcond)
#
def f_build_DisAadjMet(self):
    N_points= self.N_points

    self.Optim.dRdB = np.zeros((3*N_points,N_points))
    self.Optim.dRdW = np.zeros((3*N_points,3*N_points))

    C_mu          = self.kOmConstants.C_mu;
    sigma_om      = self.kOmConstants.sigma_om;
    alpha_om      = self.kOmConstants.alpha_om;
    gamma         = self.kOmConstants.gamma;
    sigma_d       = self.kOmConstants.sigma_d;
    sigma_k       = self.kOmConstants.sigma_k;
    alpha_k       = self.kOmConstants.alpha_k;
    nu_lam        = self.FieldVars.nu

    arr_DISCR_d2nutdy2 = self.Grid.G2_matrix @ self.FieldVars.nut
    arr_DISCR_dnutdy   = self.Grid.G1_matrix @ self.FieldVars.nut
    arr_DISCR_d2omdy2  = self.Grid.G2_matrix @ self.FieldVars.Om
    arr_DISCR_domdy    = self.Grid.G1_matrix @ self.FieldVars.Om
    arr_DISCR_d2kdy2   = self.Grid.G2_matrix @ self.FieldVars.k
    arr_DISCR_dkdy     = self.Grid.G1_matrix @ self.FieldVars.k
    arr_DISCR_d2udy2   = self.Grid.G2_matrix @ self.FieldVars.u
    arr_DISCR_dudy     = self.Grid.G1_matrix @ self.FieldVars.u
```

```
ii_last = N_points-1
for ii in range(1,N_points):
    ind_C= ii
    ind_S= ii-1
    ind_N= ind_S if ii==ii_last else (ii+1)
    u_C     = self.FieldVars.u[ind_C]
    u_N     = self.FieldVars.u[ind_N]
    u_S     = self.FieldVars.u[ind_S]
    k_C     = self.FieldVars.k[ind_C]
    k_N     = self.FieldVars.k[ind_N]
    k_S     = self.FieldVars.k[ind_S]
    omega_C = self.FieldVars.Om[ind_C]
    omega_N = self.FieldVars.Om[ind_N]
    omega_S = self.FieldVars.Om[ind_S]
    cCd2    = self.Grid.G2_matrix[ii,ind_C]
    cNd2    = self.Grid.G2_matrix[ii,ind_N] if ii!=ii_last else 0.
    cSd2    = self.Grid.G2_matrix[ii,ind_S]
    cCd1    = self.Grid.G1_matrix[ii,ind_C]
    cNd1    = self.Grid.G1_matrix[ii,ind_N] if ii!=ii_last else 0.
    cSd1    = self.Grid.G1_matrix[ii,ind_S]
    beta_C  = self.FieldVars.BetaY[ind_C]
    vnutC   = self.FieldVars.nut  [ind_C]
    vDISCR_d2nutdy2 = arr_DISCR_d2nutdy2 [ind_C]
    vDISCR_dnutdy   = arr_DISCR_dnutdy   [ind_C]
    vDISCR_d2omdy2  = arr_DISCR_d2omdy2  [ind_C]
    vDISCR_domdy    = arr_DISCR_domdy    [ind_C]
    vDISCR_d2kdy2   = arr_DISCR_d2kdy2   [ind_C]
    vDISCR_dkdy     = arr_DISCR_dkdy     [ind_C]
    vDISCR_d2udy2   = arr_DISCR_d2udy2   [ind_C]
    vDISCR_dudy     = arr_DISCR_dudy     [ind_C]
```

```
        vdRn_wi = np.array([[(cSd1* (vDISCR_dnutdy) + cSd2* (vnutC +
          ↪ nu_lam)), (cCd1* (vDISCR_dnutdy) + cCd2* (vnutC + nu_lam)),
          ↪ (cNd1* (vDISCR_dnutdy) + cNd2* (vnutC + nu_lam)), (C_mu* cSd1*
          ↪ (vDISCR_dudy)/ omega_S), (C_mu* cCd1* (vDISCR_dudy)/ omega_C +
          ↪ C_mu* (vDISCR_d2udy2)/ omega_C), (C_mu* cNd1* (vDISCR_dudy)/
          ↪ omega_N), (-C_mu* cSd1* k_S* (vDISCR_dudy)/ omega_S**2),
          ↪ (-C_mu* cCd1* k_C* (vDISCR_dudy)/ omega_C**2 - C_mu* k_C*
          ↪ (vDISCR_d2udy2)/ omega_C**2), (-C_mu* cNd1* k_N*
          ↪ (vDISCR_dudy)/ omega_N**2), ], [(2* C_mu* beta_C* cSd1* k_C*
          ↪ (vDISCR_dudy)/ omega_C), (2* C_mu* beta_C* cCd1* k_C*
          ↪ (vDISCR_dudy)/ omega_C), (2* C_mu* beta_C* cNd1* k_C*
          ↪ (vDISCR_dudy)/ omega_C), (cSd1* sigma_k* (vDISCR_dkdy)/
          ↪ omega_S + cSd2* (k_C* sigma_k/ omega_C + nu_lam) + cSd1*
          ↪ sigma_k* (vDISCR_dnutdy)/ C_mu), (C_mu* beta_C*
          ↪ (vDISCR_dudy)**2/ omega_C - alpha_k* omega_C + cCd1* sigma_k*
          ↪ (vDISCR_dkdy)/ omega_C + cCd2* (k_C* sigma_k/ omega_C +
          ↪ nu_lam) + sigma_k* (vDISCR_d2kdy2)/ omega_C + cCd1* sigma_k*
          ↪ (vDISCR_dnutdy)/ C_mu), (cNd1* sigma_k* (vDISCR_dkdy)/ omega_N
          ↪ + cNd2* (k_C* sigma_k/ omega_C + nu_lam) + cNd1* sigma_k*
          ↪ (vDISCR_dnutdy)/ C_mu), (-cSd1* k_S* sigma_k* (vDISCR_dkdy)/
          ↪ omega_S**2), (-C_mu* beta_C* k_C* (vDISCR_dudy)**2/ omega_C**2
          ↪ - alpha_k* k_C - cCd1* k_C* sigma_k* (vDISCR_dkdy)/ omega_C**2
          ↪ - k_C* sigma_k* (vDISCR_d2kdy2)/ omega_C**2), (-cNd1* k_N*
          ↪ sigma_k* (vDISCR_dkdy)/ omega_N**2), ], [(2* cSd1* gamma*
          ↪ (vDISCR_dudy)), (2* cCd1* gamma* (vDISCR_dudy)), (2* cNd1*
          ↪ gamma* (vDISCR_dudy)), (cSd1* sigma_om* (vDISCR_domdy)/
          ↪ omega_S + cSd1* sigma_d* (vDISCR_domdy)/ omega_C), (cCd1*
          ↪ sigma_d* (vDISCR_domdy)/ omega_C + cCd1* sigma_om*
          ↪ (vDISCR_domdy)/ omega_C + sigma_om* (vDISCR_d2omdy2)/
          ↪ omega_C), (cNd1* sigma_om* (vDISCR_domdy)/ omega_N + cNd1*
          ↪ sigma_d* (vDISCR_domdy)/ omega_C), (-cSd1* k_S* sigma_om*
          ↪ (vDISCR_domdy)/ omega_S**2 + cSd1* sigma_d* (vDISCR_dkdy)/
          ↪ omega_C + cSd2* (k_C* sigma_om/ omega_C + nu_lam) + cSd1*
          ↪ sigma_om* (vDISCR_dnutdy)/ C_mu), (-2* alpha_om* omega_C -
          ↪ cCd1* k_C* sigma_om* (vDISCR_domdy)/ omega_C**2 + cCd1*
          ↪ sigma_d* (vDISCR_dkdy)/ omega_C + cCd2* (k_C* sigma_om/
          ↪ omega_C + nu_lam) - k_C* sigma_om* (vDISCR_d2omdy2)/
          ↪ omega_C**2 - sigma_d* (vDISCR_dkdy)* (vDISCR_domdy)/
          ↪ omega_C**2 + cCd1* sigma_om* (vDISCR_dnutdy)/ C_mu), (-cNd1*
          ↪ k_N* sigma_om* (vDISCR_domdy)/ omega_N**2 + cNd1* sigma_d*
          ↪ (vDISCR_dkdy)/ omega_C + cNd2* (k_C* sigma_om/ omega_C +
          ↪ nu_lam) + cNd1* sigma_om* (vDISCR_dnutdy)/ C_mu), ], ]);
        vdRn_bi = np.array([[(0),],
          ↪ [(C_mu*k_C*(vDISCR_dudy)**2/omega_C),], [(0),],]);
        rows_map=np.array([ii])
        inds_map=np.array([ind_S,ind_C,ind_N])
        rows_map=np.append(np.append(rows_map, rows_map+N_points),
          ↪ rows_map+2.*N_points).astype(int)
        inds_map=np.append(np.append(inds_map, inds_map+N_points),
          ↪ inds_map+2.*N_points).astype(int)
        for i in range(3):
            for j in range(9):
                self.Optim.dRdW[rows_map[i],inds_map[j]]+=vdRn_wi[i,j]
            self.Optim.dRdB[rows_map[i],ii]+=vdRn_bi[i,0]
    self.Optim.dJdW = np.zeros(3*N_points)
    self.Optim.dJdW[:N_points]=2*(self.FieldVars.u-self.FieldVars.u_DNS)
    self.Optim.dJdB= 0.
    return self
#
def f_get_Jacobian(self):
    self=f_CFDsolve        (self,0)
    self=f_build_DisAadjMet(self)

    IndsWkeep = self.Optim.IndsWkeep
    dRdW=self.Optim.dRdW
    dRdB=self.Optim.dRdB

    dJdW=self.Optim.dJdW
    dJdB=self.Optim.dJdB

    dRdB=dRdB[IndsWkeep,:] ; dRdB=dRdB[:,1:]
    dRdW=dRdW[IndsWkeep,:] ; dRdW=dRdW[:,IndsWkeep]

    dJdW=dJdW[IndsWkeep]

    Psi=np.linalg.solve(dRdW.transpose(),-dJdW)
    DJDB = Psi @ dRdB   + dJdB
    self.Optim.DJDB[1:] = DJDB
    self.Optim.Jcost    =
      ↪ np.sum((self.FieldVars.u-self.FieldVars.u_DNS)**2)
    return self
#
MESH= f_get_Jacobian(MESH)
# plt.plot(MESH.Optim.DJDB)
# Bold Drive Method with Added Momentum
kPlus  = 1.2
kMinus = 0.5
c_1    = 0.9
m_n    = np.zeros(MESH.Optim.DJDB.shape)
alpha  = 0.1 / np.fabs(MESH.Optim.DJDB).max()

MESH.Ref_Betas=Class_Model()
MESH.Ref_Betas.float_formatter = lambda x: "%.20e" % x


MESH.Ref_Betas.BetaYFinal_Former= np.array([1.00000000000000000000e+00,
  ↪ 1.00000000000000000000e+00,
    1.00000000000000000000e+00,    1.00000000000000000000e+00,
      ↪ 1.00000000000000000000e+00,    1.00000000000000000000e+00,
    1.00000000000000000000e+00,    1.00000000000000000000e+00,
      ↪ 1.00000000000000000000e+00,    1.00000000000000000000e+00,
    1.00000000000000000000e+00,    1.00000000000000000000e+00,
      ↪ 1.00000000000000000000e+00,    1.00000000000000000000e+00,
    1.00000000000000000000e+00,    9.99999999999999000799e-01,
      ↪ 9.99999999999900412995e-01,    9.99999999999477751089e-01,
    9.99999999998402389068e-01,    9.99999999995481170245e-01,
      ↪ 9.99999999987771670540e-01,    9.99999999967868147266e-01,
    9.99999999916960979895e-01,    9.99999999789069282663e-01,
      ↪ 9.99999999473851652887e-01,    9.99999998713570370512e-01,
    9.99999996926072887149e-01,    9.99999992854866737169e-01,
      ↪ 9.99999983964405148029e-01,    9.99999965696333759091e-01,
    9.9999993173931078782 6e-01,    9.99999880470572644064e-01,
     ↪ 9.99999846315008933395e-01,    1.00000001744106215007e+00,
    1.00000112813359054442e+00,    1.00000567509086213391e+00,
     ↪ 1.00002149114692295839e+00,    1.00007177916658829275e+00,
    1.00022208170786286452e+00,    1.00064945813862604673e+00,
     ↪ 1.00181034968439997002e+00,    1.00481957994648674060e+00,
    1.01222227292733779613e+00,    1.02932485157628494932e+00,
     ↪ 1.06582547989120013376e+00,    1.13602558313960422254e+00,
    1.25323014253253028905e+00,    1.41311267974014453941e+00,
     ↪ 1.57067099404560539533e+00,    1.63872435372018476762e+00,
    1.53865140924188481542e+00,    1.27381899822109656562e+00,
     ↪ 9.44272273181597232572e-01,    6.82431956893036928413e-01,
    5.72934304661665394498e-01,    6.14119949348205684814e-01,
     ↪ 7.33209704904382753021e-01,    8.40129072704467017019e-01,
    8.82892644120351288706e-01,    8.64308247313457900241e-01,
     ↪ 8.21135504431931395608e-01,    7.94666589535528600230e-01,
    8.11632961216389836601e-01,    8.78641299151327048733e-01,
     ↪ 9.86273666427910922216e-01,    1.11717095579077185796e+00,
    1.25312389096782927922e+00,    1.37890696367545695367e+00,
     ↪ 1.48346418923621947705e+00,    1.55978108109781476998e+00,
    1.60436549319607135722e+00,    1.61670964664977456771e+00,
     ↪ 1.59878653723584540636e+00,    1.55457322225655270032e+00,
    1.48967829440887400416e+00,    1.41095280498090436261e+00,
     ↪ 1.32592845985855500857e+00,    1.24216443957985145374e+00,
    1.16630443953191620388e+00,    1.10303332798444220231e+00,
     ↪ 1.05455078400233470681e+00,    1.02023548768708360868e+00,
    9.96597354409062052838e-01,    9.77960745153325761159e-01,
     ↪ 9.58141954184070110401e-01,    9.33132116665294497615e-01,
    9.02525986952313563627e-01,    8.70428428558836175810e-01,
     ↪ 8.45552402526347113287e-01,    8.35412200334732291118e-01,
    8.41008681102708988142e-01,    8.53601322143027196887e-01,
     ↪ 8.49910810991996279462e-01,    7.92945233570415908275e-01,
    6.51682713463897500539e-01,    4.46060074499367797962e-01,
     ↪ 3.16852329457211256969e-01,    4.53915025614118783359e-01,
    7.90252239852732185632e-01,    1.00000000000000000000e+00])
#
MESH.Ref_Betas.BetaParish_Orig =
  ↪ f_build_interp(MESH.y,YplusBeta_Parish[:,0]* MESH.FieldVars.nu,
  ↪ YplusBeta_Parish[:,1])
#
MESH.Ref_Betas.BetaParish_Worked= np.array([9.99921178972190460854e-01,
    9.99996696905905757546e-01,    1.00007923613135152330e+00,
     ↪ 1.00016933125418239214e+00,    1.00026752921643957706e+00,
    1.00037438145584323124e+00,    1.00049043343924703997e+00,
     ↪ 1.00061621097600483843e+00,    1.00075220261293629243e+00,
    1.00089883729854345340e+00,    1.00105645638293783151e+00,
     ↪ 1.00122527889684254099e+00,    1.00140535893672155687e+00,
    1.00159653388767422300e+00,    1.00179836216278639327e+00,
     ↪ 1.00201004915953317820e+00,    1.00223036027832601569e+00,
    1.00245752018511313786e+00,    1.00268909812768192147e+00,
     ↪ 1.00292188017778216569e+00,    1.00315173096706389266e+00,
    1.00337345009769696169e+00,    1.00358063233109606749e+00,
     ↪ 1.00376554644194793120e+00,    1.00391905602519226903e+00,
    1.00403061761614909031e+00,    1.00408840866639348377e+00,
     ↪ 1.00407966218544308923e+00,    1.00399131889183723665e+00,
    1.003811155127086518494e+00,    1.003529610378195213 73e+00,
      ↪ 1.003142628306337824 64e+00,    1.002655947685496595 77e+00,
```

```
    1.0038111551270865184 9e+00,    1.0035296103781952137 3e+00,
```

```
    1.00381115512708651849e+00,    1.00352961037819521373e+00,
      ↪ 1.00314262830633782464e+00,    1.00265594768549659577e+00,
    1.00209144444900366544e+00,    1.00149634473112580935e+00,
      ↪ 1.00095641465650753155e+00,    1.00061460134762092444e+00,
    1.00069707282714448482e+00,    1.00154921748764724754e+00,
      ↪ 1.00368807630879719461e+00,    1.00832824889219607734e+00,
    1.01845226571757652145e+00,    1.03962079073003588192e+00,
      ↪ 1.08185717450504403914e+00,    1.15994310019833624104e+00,
    1.29018763400864289181e+00,    1.47632379727627416344e+00,
      ↪ 1.68417360179347963545e+00,    1.82726520979930251443e+00,
    1.79899985579087928755e+00,    1.55614263292952981210e+00,
      ↪ 1.17103404385366194340e+00,    7.91221740339477008774e-01,
    5.58466582882908513241e-01,    5.30468979662427009281e-01,
      ↪ 6.56802826073821788277e-01,    8.15873915305869057413e-01,
    9.07552724502361463088e-01,    9.05955129620918730993e-01,
      ↪ 8.47909008267437758199e-01,    7.86868726926705619462e-01,
    7.63712437344371330994e-01,    7.94099919879319604554e-01,
      ↪ 8.73984300337309782947e-01,    9.90120468860779645581e-01,
    1.12514672685111905004e+00,    1.26217747955314751884e+00,
      ↪ 1.38743715432280567690e+00,    1.49040669902597033847e+00,
    1.56484391316037485531e+00,    1.60774435568543139219e+00,
      ↪ 1.61879260063217333965e+00,    1.60002384373409078933e+00,
    1.55543221614227378247e+00,    1.49050698720611718429e+00,
      ↪ 1.41202835597652498478e+00,    1.32724178299619466337e+00,
    1.24359963721695243599e+00,    1.16751505938493616021e+00,
      ↪ 1.10365680591107584441e+00,    1.05418080121671553506e+00,
    1.01856536592182145284e+00,    9.93567864921946819479e-01,
      ↪ 9.73703936861988861295e-01,    9.52838443215049402113e-01,
    9.27192427609206504258e-01,    8.98126280729280912496e-01,
      ↪ 8.69402407923599240824e-01,    8.48732767611185323631e-01,
    8.43829029650253525929e-01,    8.53734887738358705356e-01,
      ↪ 8.66761408136853095385e-01,    8.55975652340188131184e-01,
    7.82697323845186643254e-01,    6.21976754296574063652e-01,
      ↪ 4.17180149563377900002e-01,    3.32452683951485805647e-01,
    5.22994482057492882099e-01,    8.44630480914004611037e-01,
      ↪ 9.99900799055203570 73e-01,])
#

MESH.FieldVars.BetaY = np.ones(MESH.y.shape)+0.
BoolOptimize=True
iiter   =0

tic() # N_iters_print = 100
vkey = True if BoolOptimize else False
while vkey:
    iiter  +=1
    BackupModel = deepcopy(MESH)
    m_n = (c_1*m_n + (1-c_1)*MESH.Optim.DJDB)
    MESH.FieldVars.BetaY -= alpha*m_n
    #
    MESH=f_get_Jacobian(MESH)
    if MESH.Optim.Jcost<BackupModel.Optim.Jcost:
        alpha*=kPlus
    else:
        alpha *=kMinus
        MESH   = deepcopy(BackupModel)
        m_n    = np.copy(MESH.Optim.DJDB)
    if alpha<1e-14: vkey=False
    if toc(0)>=60: print('Iteration %8d: Jcost=%12.4e (alpha = %12.4e)' %
        ↪ (iiter,MESH.Optim.Jcost,alpha));tic()
print('Iteration END: Jcost=%12.4e (alpha = %12.4e)' %
  ↪ (MESH.Optim.Jcost,alpha))
#
def f_get_Pk(self):
    self=f_get_Jacobian(self)
    dudy = self.Grid.G1_matrix @ self.FieldVars.u
    nut = self.FieldVars.nut
    BetaY = self.FieldVars.BetaY
    Mod_Pk = BetaY*nut*(dudy**2)
    self.FieldVars.Mod_Pk=np.copy(Mod_Pk)
    self.yPlus=self.Grid.y_Plus
    return self
#
MESH_Beta1 = deepcopy(MESH)
MESH_Beta1.FieldVars.BetaY = MESH_Beta1.y*0.+1.
MESH_Beta1=f_get_Pk(MESH_Beta1)

MESH_ParishOrig = deepcopy(MESH)
MESH_ParishOrig.FieldVars.BetaY =
  ↪ MESH_ParishOrig.Ref_Betas.BetaParish_Orig +0.
MESH_ParishOrig=f_get_Pk(MESH_ParishOrig)

MESH_ParishWorked = deepcopy(MESH)
MESH_ParishWorked.FieldVars.BetaY =
  ↪ MESH_ParishWorked.Ref_Betas.BetaParish_Worked +0.
MESH_ParishWorked=f_get_Pk(MESH_ParishWorked)

MESH_Former = deepcopy(MESH)
MESH_Former.FieldVars.BetaY = MESH_Former.Ref_Betas.BetaYFinal_Former +0.
MESH_Former=f_get_Pk(MESH_Former)

N_total=3
fig, axarr = plt.subplots(N_total,1)
ax1,ax2,ax3=axarr

matplotlib.rcParams.update({'font.size': 20})
matplotlib.rcParams.update({'font.family': ”Times New Roman”})
plt.rcParams.update({'font.size': 20})
plt.rcParams.update({'font.family': ”Times New Roman”})

self=MESH_Beta1       ;ax1.semilogx(self.yPlus, self.FieldVars.u, 'y--',
  ↪ linewidth=5, label='Standard')
self=MESH_Former
  ↪ ;ax1.semilogx(self.yPlus,self.FieldVars.u,'b',linewidth=5,label='First
  ↪ Local Min.')
self=MESH_ParishWorked;ax1.semilogx(self.yPlus, self.FieldVars.u, 'r--',
  ↪ linewidth=5, label='Second Local Min.')
self=MESH_Beta1       ;ax1.semilogx(self.yPlus[::3],
  ↪ self.FieldVars.u_DNS[::3], 'go', markersize=10, label='DNS Data')
ax1.legend(loc='best')
ax1.set_ylabel(r'$U^+$')
ax1.grid()
```

```
# self=MESH_Beta1       ;ax2.semilogx(self.yPlus, self.FieldVars.BetaY,
  ↪ 'y--', linewidth=5, label='Standard')
self=MESH_Former       ;ax2.semilogx(self.yPlus, self.FieldVars.BetaY, 'b',
  ↪ linewidth=5, label='First Local Min.')
self=MESH_ParishWorked;ax2.semilogx(self.yPlus, self.FieldVars.BetaY, 'r',
  ↪ linewidth=5, label='Second Local Min.')
self=MESH_ParishOrig  ;ax2.semilogx(self.yPlus[::],
  ↪ self.FieldVars.BetaY[::],  'g--', linewidth=5, label='Parish et al.
  ↪ (2016)')
ax2.legend(loc='best')
ax2.set_ylabel(r'$\beta(y)$')
ax2.grid()

self=MESH_Beta1        ;ax3.semilogx(self.yPlus, self.FieldVars.Mod_Pk,
  ↪ 'y--', linewidth=5, label='Standard')
self=MESH_Former       ;ax3.semilogx(self.yPlus, self.FieldVars.Mod_Pk,
  ↪ 'b', linewidth=5, label='First Local Min.')
self=MESH_ParishWorked;ax3.semilogx(self.yPlus, self.FieldVars.Mod_Pk,
  ↪ 'r', linewidth=5, label= 'Second Local Min.')
ax3.legend(loc='best')
ax3.set_ylabel(r'$\beta\cdot P_k$')
ax3.set_xlabel(r'$Y^+$')
ax3.grid()

fig.set_size_inches(fig.get_size_inches()*1.1*np.array([1.6,3.2]))

plt.savefig(”kOmOptim_Parish2015.pdf”, bbox_inches = 'tight')
plt.show()
```

A.4. Deep Dreaming the Reynolds Number

```
# %reset -f

# Import Libraries
import numpy as np
import datetime
import os
import shutil
import time
import tensorflow as tf
from copy import deepcopy
import matplotlib
plt= matplotlib.pyplot
import svgpathtools
class Class_Generic: pass
tf.reset_default_graph() # Clean Tensorflow
ThisFile=deepcopy(__file__) # Store this filename
#
array=np.array
np.random.seed(0)
Bool_Use_Re_Directly=0
BoolTrain           =0
#
def tic():
    global GlobalTimeTic
    GlobalTimeTic=time.time()
def toc(BoolPrint=1):
    vToc=time.time()-GlobalTimeTic
    if BoolPrint: myprint( ”Elapsed Time: %f s” % vToc )
    return vToc
# Create a New Folder to store the Log file and the Original Version of
  ↪ the Script employed
if BoolTrain:
    f_formated_date_now = lambda :
      ↪ datetime.datetime.now().strftime(”%Y%m%d_%H%M%S_”)
    vLogFilename='%sOptimLog_%s_UseReDirectly_%d.log' %
      ↪ (f_formated_date_now(),os.path.basename(ThisFile).replace('.py',
      ↪ '').replace('.ipy',''),Bool_Use_Re_Directly);
    source_file = ThisFile
    dest_file   = os.path.join(os.path.dirname(ThisFile),
      ↪ vLogFilename.replace('.log' ,''),  os.path.basename(ThisFile))
    Path_dest_file = os.path.dirname(dest_file)
    if not os.path.exists(Path_dest_file): os.makedirs(Path_dest_file)
    # os.chdir(Path_dest_file)
    shutil.copy(source_file,dest_file)
    vLogFilename = os.path.join(Path_dest_file,vLogFilename)
    #
    def myprint(vstr,to_console=1,vLogFilename=vLogFilename):
        if to_console: print(vstr)
        with open(vLogFilename, ”a”) as myfile:
            myfile.write(vstr)
            myfile.write('\n')
else:
    myprint=lambda x,to_console=1: print(x)
# ----------------------------------------------------------------
# Drag Coefficient Curve for a Smooth cylinder
# ----------------------------------------------------------------
# # Source : Frank M. White, Fluid Mechanics, 7th Edition, Fig. 5.3, Page
  ↪ 318
def build_Cd_Curve_White(): # Build the Bezier Curve from the Textbook
    Text_Re_1e2={'transform=': '
      ↪ transform=”matrix(1.3333333,0,0,-1.3333333,278.03866,281.01334)”',
        'd=': '       d=”m 30.884766,-38.699219 v 6 h 0.5 v -6 z”',
        'id=': '       id=”Re_1e2”',}
    Text_Re_1e6={'transform=': '
      ↪ transform=”matrix(1.3333333,0,0,-1.3333333,278.03866,281.01334)”',
        'd=': '       d=”m 179.52539,-38.699219 v 6 h 0.5 v -6 z”',
        'id=': '       id=”Re_1e6”',}
    Text_CD_1={'transform=': '
      ↪ transform=”matrix(1.3333333,0,0,-1.3333333,278.03866,281.01334)”',
        'd=': '       d=”m -6,-0.25 v 0.5 h 6 v -0.5 z”',
        'id=': '       id=”Cd_1”'}
    Text_CD_4={'transform=':
      ↪ 'transform=”matrix(1.3333333,0,0,-1.3333333,278.03866,281.01334)”',
        'd=': '       d=”m -6,115.85156 v 0.5 h 6 v -0.5 z”',
        'id=': '       id=”Cd_4”'}
    Text_CD_Re_curve={'transform=':
      ↪ 'transform=”matrix(1.3333333,0,0,-1.3333333,270.30533,161.68801)”',
        'd=': '       d=”m 0,0 c 2.575,-13.9 10.279,-60.616 44.437,-78.938
          ↪ 9.438,-5.063 20,-2.775 25.5,-5.875 5.501,-3.1 9.688,-5.938
            ↪ 17.188,-5.938 7.5,0 13.125,0.5 21.625,4.563 6.934,3.313
            ↪ 35.139,8.679 54.425,3.087 13.45,-3.9 9.888,-28.837 17.3,-32.4
            ↪ 2.241,-1.077 20.713,6.25 25.9,5.1”',
        'id=': '       id=”Cd_curve”'}
    #
    def f_parsepath(x):
        x_path      = svgpathtools.parse_path(x['d='].split('”')[1])
        x_translate =
            ↪ ((x['transform='].split('(')[1]).split(')')[0]).split(',')
        a,b,c,d,e,f = list(map(float,x_translate))
        x_path=x_path.translated(e/a+f/d*1j)
        return x_path
    #
    Cd_Re_curve_Brute = f_parsepath(Text_CD_Re_curve)
    Re_1e2_Brute      = f_parsepath(Text_Re_1e2     )
    Re_1e6_Brute      = f_parsepath(Text_Re_1e6     )
    CD_1_Brute        = f_parsepath(Text_CD_1       )
    CD_4_Brute        = f_parsepath(Text_CD_4       )
    #
    i_samples = np.linspace(0,1,1000)
    X0_pic = np.mean([Re_1e2_Brute.point(i).real for i in i_samples])
    X0_real    = 1e2
    #
    X1_pic = np.mean([Re_1e6_Brute.point(i).real for i in i_samples])
    X1_real    = 1e6
    #
    Y0_pic = np.mean([CD_1_Brute.point(i).imag for i in i_samples])
    Y0_real    = 1.
    #
    Y1_pic = np.mean([CD_4_Brute.point(i).imag for i in i_samples])
    Y1_real    = 4.
    #
    def f_convert_X_to_real_log(X0=np.nan,X1=np.nan,X0_real= np.nan,
      ↪ X1_real=np.nan):
        # X_pic= c*log(X_real)+d
        # X_real= exp((X_pic-d)/c)
        b=np.array([X0,X1])
        A=np.zeros((2,2))
        A[0,0]=np.log(X0_real)
        A[1,0]=np.log(X1_real)
        A[:,1]=1.
        c,d=np.linalg.solve(A,b)
        f_real_from_pic = lambda x,c=c,d=d: np.exp((x-d)/c)
        return f_real_from_pic # np.exp((x-d)/c)
    #
    f_Xreal_from_X_pic=f_convert_X_to_real_log(X0=X0_pic,X1=X1_pic,
      ↪ X0_real=X0_real,X1_real=X1_real)
    f_Yreal_from_Y_pic= lambda y,
        ↪ Y0_real=Y0_real,Y1_real=Y1_real,Y0_pic=Y0_pic,Y1_pic=Y1_pic:
        ↪ (y-Y0_pic)*(Y1_real-Y0_real)/(Y1_pic-Y0_pic) + Y0_real
    #
    def f_get_CdRe_from_t(t,Cd_Re_curve_Brute= Cd_Re_curve_Brute,
        ↪ f_Xreal_from_X_pic=f_Xreal_from_X_pic,
        ↪ f_Yreal_from_Y_pic=f_Yreal_from_Y_pic):
        fval=Cd_Re_curve_Brute.point(t)
        xfval=f_Xreal_from_X_pic(fval.real)
        yfval=f_Yreal_from_Y_pic(fval.imag)
        return np.array([xfval,yfval])
    #
    return f_get_CdRe_from_t
#
CurveDrag=Class_Generic()
CurveDrag.f_get_ReCd_pairs=build_Cd_Curve_White()
#

# ---------------------------------------------------------------
# Generate Experimental Data: U,D,nu,rho (Re will be hidden)
# ---------------------------------------------------------------
def f_log_sample_distrib (x0_range,N_samples):
    x_range=np.log10(x0_range)
    return 10**( min(x_range) +
      ↪  (max(x_range)-min(x_range))*np.random.rand(N_samples) )
def f_gen_Gen_data(N_samples=10000,CurveDrag=CurveDrag):
    # Generate fake experimental data for U,D,nu,rho
    #
    ExpData=Class_Generic()
    ReCd_samples = np.array([CurveDrag.f_get_ReCd_pairs(i) for i in
      ↪ np.linspace(0.,1.,N_samples)])
    ExpData.Re_data      = np.copy(ReCd_samples[:,0])
    ExpData.Cd_Orig      = np.copy(ReCd_samples[:,1])
    ExpData.Cd_Noisy = ExpData.Cd_Orig *( 1+
      ↪ 0.3*(np.random.rand(N_samples) -0.5)) # Add Moise To measurementx

    ExpData.U_data  = f_log_sample_distrib  ( np.array([0.1 ,10. ]
      ↪ ),N_samples)
    ExpData.D_data  = f_log_sample_distrib  ( np.array([0.1 ,10. ]
      ↪ ),N_samples)
    # Re = U*d/nu
    ExpData.nu_data = ExpData.U_data*ExpData.D_data/ExpData.Re_data
    ExpData.rho_data= f_log_sample_distrib(np.array([0.1,10.]),N_samples)
    return ExpData
#
N_samples=int(2e3)
myprint('N_samples= %d' % N_samples)
ExpData=f_gen_Gen_data(N_samples=N_samples)
#
# ---------------------------------------------------------------
# # # # Create Feature and Targets Vectors
# ---------------------------------------------------------------
if Bool_Use_Re_Directly:
    X0      = np.log10( np.array([ExpData.Re_data]).transpose() ) # Re
else:
    X0     =np.log10( np.array([ExpData.U_data,ExpData.D_data,
      ↪ ExpData.nu_data, ExpData.rho_data]).transpose() ) # U,d,nu,rho
Y0     =np.copy([ExpData.Cd_Noisy]).transpose()
Y0_real=np.copy([ExpData.Cd_Orig]).transpose()

# ---------------------------------------------------------------
# # # # Prepare the Simulation Data
# ---------------------------------------------------------------
# Note: Y0=Cd will be scaled to the range [-1,1]
#
def f_scale(x0):
    x0_mean=np.mean(x0,axis=0)
    temp_x=x0-x0_mean
    x_amp= np.maximum(np.fabs(temp_x.min(axis=0)),
      ↪ np.fabs(temp_x.max(axis=0)))*1.1
    x=(x0-x0_mean)/x_amp
    return x,x0_mean,x_amp
NNclass=Class_Generic()
NNclass.Yscaling=Class_Generic()
_ , Y0_mean , Y0_amp  =  f_scale(Y0)
NNclass.Yscaling.Y0_mean=Y0_mean
NNclass.Yscaling.Y0_amp =Y0_amp
NNclass.Yscaling.f_Y0_to_Ytrain = lambda y0,Y0_mean=Y0_mean,Y0_amp=Y0_amp:
  ↪  (y0-Y0_mean)/Y0_amp # = Y_train
NNclass.Yscaling.f_Ytrain_to_Y0 = lambda y ,Y0_mean=Y0_mean,Y0_amp=Y0_amp:
  ↪ y*Y0_amp+Y0_mean  # = Y0

NNclass.X_train = X_train=np.copy(X0)
NNclass.Y_train = Y_train=NNclass.Yscaling.f_Y0_to_Ytrain(Y0)

n_features=len(X_train[0])

# --------------------------------------------------------------
# Build the NN model (tf)
# --------------------------------------------------------------
tf_Y       = tf.placeholder (shape=[None, 1         ], dtype=tf.float64)
tf_X       = tf.placeholder (shape=[None, n_features], dtype=tf.float64)

NNclass.feed_train = feed_train = { tf_X: X_train , tf_Y: Y_train }

tf_nn      = tf.layers.dense(tf_X   , 1 if Bool_Use_Re_Directly else 2,
  ↪ activation=None,use_bias=False,kernel_constraint=
  ↪ tf.keras.constraints.MaxNorm(1.2*n_features) )
tf_nn      = tf.layers.dense(tf_nn  , 5                        ,
  ↪ activation=tf.nn.tanh)
tf_nn      = tf.layers.dense(tf_nn  , 2                        ,
  ↪ activation=tf.nn.tanh)
tf_nn      = tf.layers.dense(tf_nn  , 1                        ,
  ↪ activation=tf.nn.tanh)

tf_cost           = tf.losses.mean_squared_error(tf_nn,tf_Y)
tf_optimizer      = tf.train.AdamOptimizer().minimize( tf_cost )
tf_init           = tf.global_variables_initializer()
tf_run            = [tf_optimizer, tf_cost]
tf_Trainable_Dict = {i.name:i for i in tf.trainable_variables()}
f_predict_Y0     = lambda vsess,tf_nn=tf_nn,tf_X=tf_X,NNclass=NNclass:
  ↪ NNclass.Yscaling.f_Ytrain_to_Y0(vsess.run(tf_nn,feed_dict={tf_X:
  ↪ NNclass.X_train}))
# --------------------------------------------------------------
# Run the NN Architecture
# --------------------------------------------------------------
N_iters_trials=10000   if BoolTrain else 1
N_trial_sessions = 20 if BoolTrain else 1
Jcost_best = 1e20
```

```
for isess in range(N_trial_sessions):
    sess = tf.Session()
    sess.run(tf_init)
    Jcost_min = 1e20
    tic()
    for i in range(N_iters_trials):
        _, Jcost = sess.run(tf_run,feed_dict = feed_train)
        if Jcost<Jcost_min:
            Jcost_min=Jcost+0.
        if toc(0)>60.: myprint('    Iter %12d: (Jcost= %14.8e)
          ↪  (Jcost_min=%14.8e) ' % (i,Jcost,Jcost_min));tic()
    if (Jcost_min<Jcost_best) or (isess==0):
        sess_best=sess ; Jcost_best=Jcost_min+0.
    else:
        sess.close()
    myprint('End Loop %12d: (Jcost_min= %14.8e) (Jcost_best=%14.8e) ' %
      ↪  (isess,Jcost_min,Jcost_best))
#
#
# Train the best session found
sess=sess_best
N_iters_final=int(1e7) if BoolTrain else 1
tic()
for i in range(N_iters_final):
    _, Jcost = sess.run(tf_run,feed_dict = feed_train)
    if Jcost<Jcost_best:
        Jcost_best=Jcost+0.
        Backup_best = sess.run(tf_Trainable_Dict)
    if toc(0)>60.:
        myprint('',to_console=0)
        myprint('    Iter %12d: (Jcost= %14.8e) (Jcost_best=%14.8e) ' %
          ↪  (i,Jcost,Jcost_best))
        myprint('--------J_best Dict------------',to_console=0)
        myprint(str(Backup_best),to_console=0)
        myprint('',to_console=0)
        tic()
#

myprint('',to_console=0)
myprint('    Iter %12d: (Jcost= %14.8e) (Jcost_best=%14.8e) ' %
  ↪  (i,Jcost,Jcost_best))
myprint('--------J_best Dict------------',to_console=0)
myprint(str(Backup_best),to_console=0)
myprint('',to_console=0)



# for ikey in tf_Trainable_Dict.keys():
  ↪  sess.run(tf_Trainable_Dict[ikey].assign(Backup_best[ikey]))

# plt.plot(Y0);plt.plot(f_predict_Y0(sess_best));plt.plot(Y0_real)
if not BoolTrain:
    Backup_best={'dense/kernel:0': array([[-1.39331497e+00,
      ↪  9.49538089e-01],
        [-1.39606363e+00,  9.40983341e-01],
        [ 1.40037414e+00, -9.36161694e-01],
        [ 3.25517968e-03,  7.18397510e-04]]), 'dense_1/kernel:0': array([[
          ↪ 2.42057539, -3.9581056 , -0.27209229, -0.17950244,  1.3334501
          ↪ ],
        [-0.45185046,  2.28074223, -1.69290475,  1.40506615,
          ↪ -0.02079725]]), 'dense_1/bias:0': array([23.45792659,
          ↪ 2.62607019,  3.47403063, -5.22521379,  0.45628362]),
          ↪ 'dense_2/kernel:0': array([[-0.14453812, 17.53733587],
        [ 1.8486003 , -4.88565624],
        [ 4.68923558,  4.43605566],
        [ 4.28549187,  2.33767319],
        [ 5.23769164,  5.01456166]]), 'dense_2/bias:0': array([
          ↪ 1.67622862, -4.72989951]), 'dense_3/kernel:0':
          ↪ array([[4.26540699],
        [0.22430622]]), 'dense_3/bias:0': array([3.95017455])}

    for ikey in tf_Trainable_Dict.keys():
      ↪ sess.run(tf_Trainable_Dict[ikey].assign(Backup_best[ikey]))
    Jcost = sess.run(tf_cost,feed_dict = feed_train)
# sess.close()


matplotlib.rcParams.update({'font.size': 20})
matplotlib.rcParams.update({'font.family': ”Times New Roman”})
plt.rcParams.update({'font.size': 20})
plt.rcParams.update({'font.family': ”Times New Roman”})
fig=plt.figure()
ax1 = fig.add_subplot(111)
ax1.semilogx(ExpData.Re_data,ExpData.Cd_Orig,'b',linewidth=5)
fig.set_size_inches(fig.get_size_inches()*1.1*np.array([1.6,1.0]))

ax1.set_xlabel('$Re_D$')
ax1.set_ylabel('$C_D$')
ax1.grid()
print('Font, Size:
  ↪ ',plt.rcParams[”font.family”],plt.rcParams[”font.size”])
plt.savefig(”Drag_Coeff_Cylinder_Orig.pdf”, bbox_inches = 'tight')
plt.show()

# ------------------------------
# Generate Test Data
N_sample_plot=int(5e3)
test_Re=np.sort(f_log_sample_distrib([ min(ExpData.Re_data),
  ↪ max(ExpData.Re_data)], N_sample_plot))
test_U=f_log_sample_distrib([0.1,10],N_sample_plot)
test_D=f_log_sample_distrib([0.1,10],N_sample_plot)
test_nu=test_U*test_D/test_Re
test_rho=f_log_sample_distrib([0.1,10],N_sample_plot)
test_XStack=np.log10(np.matrix([test_U,test_D,test_nu,test_rho])
  ↪ .transpose())
Cd_NN_test =
  ↪ NNclass.Yscaling.f_Ytrain_to_Y0(sess.run(tf_nn,feed_dict={tf_X:
  ↪ test_XStack}))
# ------------------------------

matplotlib.rcParams.update({'font.size': 20})
matplotlib.rcParams.update({'font.family': ”Times New Roman”})
```

```
plt.rcParams.update({'font.size': 20})
plt.rcParams.update({'font.family': ”Times New Roman”})
fig=plt.figure()
ax1 = fig.add_subplot(111)
ax1.semilogx(ExpData.Re_data,ExpData.Cd_Noisy,'g',linewidth=5,label='Noisy
  ↪  Data')
ax1.semilogx(test_Re,Cd_NN_test,'b',linewidth=5,label='NN Predictions')
ax1.semilogx(ExpData.Re_data,ExpData.Cd_Orig, 'r--', linewidth=5,
  ↪  label='Original Curve')
fig.set_size_inches(fig.get_size_inches()*1.1*np.array([1.6,1.0]))

ax1.set_xlabel('$Re_D$')
ax1.set_ylabel('$C_D$')
ax1.grid()
ax1.legend(loc='best')
print('Font, Size:
  ↪  ',plt.rcParams[”font.family”],plt.rcParams[”font.size”])
plt.savefig(”NN_drag_cylinder.pdf”, bbox_inches = 'tight')
plt.show()
```

B

Machine Learning System for the MK Turbulence Model

B.1. Source Code for the Intelligent Relaxation Factor Methodology

```
%reset -f
from IPython.display import display,Markdown,Latex,Math
import os
import sys
import time
import datetime
import dill
from   importlib.machinery import SourceFileLoader
from   copy import deepcopy
import matplotlib
plt = matplotlib.pyplot # as plt
FormatStrFormatter = matplotlib.ticker.FormatStrFormatter
import numpy as np
from collections import defaultdict

class Class_CFDdata: pass
class Class_RunData: pass
class ClassGeneric : pass

array=np.array
GlobalTimeTic=0.
def tic():
    global GlobalTimeTic
    GlobalTimeTic=time.time()+0.
def toc(BoolPrint=1):
    global GlobalTimeTic
    vToc=time.time()-GlobalTimeTic+0.
    if BoolPrint: print( ”Elapsed Time: %f ms” % (vToc*1e3) )
    return vToc
#
scaled_Delt_K_true = array([ 0.00000000e+00,  1.42625929e-04,
  ↪ 3.01190770e-04,  4.72656306e-04,
        6.54156248e-04,  8.41081474e-04,  1.02617805e-03,  1.19868398e-03,
        1.34338342e-03,  1.43961681e-03,  1.46034737e-03,  1.37114448e-03,
        1.12909177e-03,  6.81685646e-04, -3.45962117e-05, -1.09640420e-03,
       -2.59644022e-03, -4.64722262e-03, -7.38482170e-03, -1.09713445e-02,
       -1.55948140e-02, -2.14646024e-02, -2.88017461e-02, -3.78189909e-02,
       -4.86847804e-02, -6.14693029e-02, -7.60877779e-02, -9.22723428e-02,
       -1.09572555e-01, -1.27374279e-01, -1.44939857e-01, -1.61484475e-01,
       -1.76252229e-01, -1.88570334e-01, -1.97913710e-01, -2.03934971e-01,
       -2.06462381e-01, -2.05498435e-01, -2.01194347e-01, -1.93823156e-01,
       -1.83746721e-01, -1.71373007e-01, -1.57140532e-01, -1.41477924e-01,
       -1.24769954e-01, -1.07389732e-01, -8.96500394e-02, -7.18683756e-02,
       -5.43635802e-02, -3.74742282e-02, -2.15510379e-02, -6.93428065e-03,
        6.06953884e-03,  1.72179925e-02,  2.63523704e-02,  3.34018369e-02,
        3.83858243e-02,  4.13992934e-02,  4.25973514e-02,  4.21844031e-02,
        4.03865981e-02,  3.74520857e-02,  3.36263665e-02,  2.91538482e-02,
        2.42567913e-02,  1.91510573e-02,  1.40278450e-02,  9.05643368e-03,
        4.38334354e-03,  1.30272278e-04, -3.60850162e-03, -6.76618227e-03,
       -9.30279707e-03, -1.12047360e-02, -1.24844501e-02, -1.31782238e-02,
       -1.33423726e-02, -1.30516514e-02, -1.23900929e-02, -1.14493290e-02,
       -1.03205875e-02, -9.08873543e-03, -7.82787766e-03, -6.60021375e-03,
       -5.45025968e-03, -4.41064503e-03, -3.50010262e-03, -2.72742846e-03,
       -2.09264984e-03, -1.58822048e-03, -1.20004834e-03, -9.08809845e-04,
       -6.94606394e-04, -5.39064778e-04, -4.27089266e-04, -3.47220454e-04,
       -2.91492567e-04, -2.54674715e-04, -2.33597981e-04,
         ↪  -1.13281938e-04])
scaled_Delt_K_orig = array([ 0.00000000e+00,  1.42625929e-04,
  ↪  3.01190772e-04,  4.72656320e-04,
        6.54156299e-04,  8.41081617e-04,  1.02617839e-03,  1.19868468e-03,
        1.34338479e-03,  1.43961931e-03,  1.46035172e-03,  1.37115174e-03,
        1.12910352e-03,  6.81704164e-04, -3.45676954e-05, -1.09636114e-03,
       -2.59637628e-03, -4.64712907e-03, -7.38468661e-03, -1.09711516e-02,
       -1.55945416e-02, -2.14642212e-02, -2.88012170e-02, -3.78182624e-02,
       -4.86837844e-02, -6.14679501e-02, -7.60859518e-02, -9.22698921e-02,
       -1.09569284e-01, -1.27369936e-01, -1.44934121e-01, -1.61476934e-01,
       -1.76242365e-01, -1.88557494e-01, -1.97897076e-01, -2.03913532e-01,
       -2.06434889e-01, -2.05463371e-01, -2.01149878e-01, -1.93767103e-01,
       -1.83676531e-01, -1.71285748e-01, -1.57032921e-01, -1.41346411e-01,
       -1.24610878e-01, -1.07199598e-01, -8.94259527e-02, -7.16086883e-02,
       -5.40688107e-02, -3.71483196e-02, -2.12030116e-02, -6.58031766e-03,
        6.40361052e-03,  1.74939195e-02,  2.65166726e-02,  3.33834774e-02,
        3.80951539e-02,  4.07294615e-02,  4.14298048e-02,  4.04004833e-02,
        3.78874207e-02,  3.41885860e-02,  2.96402328e-02,  2.46275756e-02,
        1.95668367e-02,  1.49118545e-02,  1.11066875e-02,  8.53154448e-03,
        7.41314054e-03,  7.70626599e-03,  8.96734906e-03,  1.02653065e-02,
        1.01940451e-02,  7.06652319e-03, -6.41696069e-04, -1.35916708e-02,
       -3.07438504e-02, -4.86867288e-02, -6.15904435e-02, -6.23129001e-02,
       -4.49208880e-02, -8.28800409e-03,  4.05122421e-02,  8.51588961e-02,
        1.03626974e-01,  7.71480455e-02,  2.39372682e-03, -9.90304829e-02,
       -1.81597194e-01, -1.92584888e-01, -1.01524155e-01,  7.14941291e-02,
        2.48906481e-01,  3.24635531e-01,  2.23531588e-01, -3.89576123e-02,
       -3.39254892e-01, -4.96845441e-01, -3.79007387e-01,
         ↪  -1.13281938e-04])
scaled_Prod_K      = array([0.00000000e+00, 4.96819315e-06,
  ↪  3.22953161e-05, 1.05698380e-04,
       2.57410406e-04, 5.30921425e-04, 9.84586099e-04, 1.69635874e-03,
       2.76996103e-03, 4.34275683e-03, 6.59610024e-03, 9.76881369e-03,
       1.41735732e-02, 2.02166348e-02, 2.84216057e-02, 3.94566993e-02,
       5.41585794e-02, 7.35494679e-02, 9.88434266e-02, 1.31408208e-01,
      1.72666523e-01, 2.23920958e-01, 2.86046978e-01, 3.59144856e-01,
      4.42213040e-01, 5.32960767e-01, 6.27873486e-01, 7.22502398e-01,
      8.12033945e-01, 8.91830535e-01, 9.58022279e-01, 1.00804358e+00,
      1.04078739e+00, 1.05643923e+00, 1.05621474e+00, 1.04202949e+00,
      1.01609297e+00, 9.80725868e-01, 9.38168493e-01, 8.90453693e-01,
      8.39378617e-01, 7.86482618e-01, 7.33073919e-01, 6.80164905e-01,
      6.28683022e-01, 5.79172579e-01, 5.32201577e-01, 4.88046298e-01,
      4.46929895e-01, 4.08943371e-01, 3.74082470e-01, 3.42299356e-01,
      3.13451267e-01, 2.87400675e-01, 2.63936262e-01, 2.42879960e-01,
      2.24017483e-01, 2.07129301e-01, 1.92042563e-01, 1.78543612e-01,
      1.66482279e-01, 1.55673721e-01, 1.45992031e-01, 1.37291450e-01,
      1.29453638e-01, 1.22375560e-01, 1.15950938e-01, 1.10091675e-01,
      1.04708610e-01, 9.97309345e-02, 9.50748504e-02, 9.06810860e-02,
      8.64837310e-02, 8.24268137e-02, 7.84604883e-02, 7.45430230e-02,
      7.06367968e-02, 6.67227000e-02, 6.27814134e-02, 5.88105213e-02,
      5.48130736e-02, 5.08015776e-02, 4.67897624e-02, 4.28044931e-02,
      3.88615761e-02, 3.49886399e-02, 3.12020918e-02, 2.75219299e-02,
      2.39635149e-02, 2.05402517e-02, 1.72708905e-02, 1.41702643e-02,
      1.12643751e-02, 8.58300571e-03, 6.16762064e-03, 4.06937253e-03,
      2.34776051e-03, 1.06334785e-03, 2.68878392e-04, 0.00000000e+00])
Ystar= array([0.00000000e+00, 1.22337292e-01, 2.52287273e-01,
  ↪  3.90322528e-01,
      5.36950697e-01, 6.92717338e-01, 8.58207343e-01, 1.03404313e+00,
      1.22087694e+00, 1.41937503e+00, 1.63024565e+00, 1.85427136e+00,
      2.09228032e+00, 2.34512749e+00, 2.61371955e+00, 2.89905082e+00,
      3.20215206e+00, 3.52408001e+00, 3.86600440e+00, 4.22914305e+00,
      4.61474096e+00, 5.02415331e+00, 5.45874233e+00, 5.92003609e+00,
      6.40977980e+00, 6.92970708e+00, 7.48136086e+00, 8.06639711e+00,
      8.68717912e+00, 9.34598048e+00, 1.00444624e+01, 1.07848606e+01,
      1.15700052e+01, 1.24020843e+01, 1.32837688e+01, 1.42185074e+01,
      1.52087091e+01, 1.62575026e+01, 1.73683059e+01, 1.85443926e+01,
      1.97898057e+01, 2.11073045e+01, 2.25026778e+01, 2.39782929e+01,
      2.55412699e+01, 2.71918191e+01, 2.89390078e+01, 3.07853166e+01,
      3.27380609e+01, 3.48005176e+01, 3.69763501e+01, 3.92765490e+01,
      4.17045596e+01, 4.42685133e+01, 4.69687180e+01, 4.98214893e+01,
      5.28253576e+01, 5.59885248e+01, 5.93237828e+01, 6.28286402e+01,
      6.65212969e+01, 7.03964531e+01, 7.44794123e+01, 7.87565560e+01,
      8.32508421e+01, 8.79627699e+01, 9.29016081e+01, 9.80733616e+01,
      1.03483806e+02, 1.09142338e+02, 1.15051805e+02, 1.21218729e+02,
      1.27648977e+02, 1.34345907e+02, 1.41316520e+02, 1.48562970e+02,
      1.56084208e+02, 1.63886752e+02, 1.71968621e+02, 1.80330593e+02,
      1.88973586e+02, 1.97890628e+02, 2.07072324e+02, 2.16533102e+02,
      2.26248214e+02, 2.36222288e+02, 2.46433152e+02, 2.56881455e+02,
      2.67558102e+02, 2.78433478e+02, 2.89515726e+02, 3.00763835e+02,
      3.12187082e+02, 3.23753818e+02, 3.35447182e+02, 3.47244576e+02,
      3.59137652e+02, 3.71099523e+02, 3.83095313e+02, 3.95138019e+02])
comp_factor=0.6

def f_get_Delta_with_Compliance_Factor(delta_ML,dest,comp_factor=0.9):
    K=(comp_factor*np.linalg.norm(delta_ML))**2
    #
    lambd=1e-8
    lambd_old=np.copy(lambd)
    iiter=0 ; vkey=True ; NmaxIters=200
    while vkey:
       iiter+=1
       lambda_add_dest2=lambd+dest**2
       deltaML_dest=delta_ML*dest
       #
       f_residual      = np.sum(         (deltaML_dest*dest
         ↪ /lambda_add_dest2)**2     )-K
       Grad_f_residual = np.sum(    -2*(
         ↪ (deltaML_dest*dest)**2)/(lambda_add_dest2**3)     )
       #
       lambd_old=np.copy(lambd)
       lambd -= f_residual/Grad_f_residual
       if (np.fabs(lambd-lambd_old)<1e-13) or (iiter>NmaxIters):
         ↪ vkey=False
    DBeta_now=deltaML_dest/lambda_add_dest2
    improved_Delta = DBeta_now*dest
    return improved_Delta
#
tic()
scaled_Delt_K_corrected =
    ↪ f_get_Delta_with_Compliance_Factor(scaled_Delt_K_orig, scaled_Prod_K,
    ↪ comp_factor)
toc(1)

# plt.plot(scaled_Delt_K_orig)
# plt.plot(scaled_Prod_K)
# plt.plot(scaled_Delt_K_corrected)
# plt.plot(scaled_Delt_K_true)

fig, axarr = plt.subplots(1,1)
axarr=[axarr]
ax1=axarr[0]

matplotlib.rcParams.update({'font.size': 34})
matplotlib.rcParams.update({'font.family': "Times New Roman"})
plt.rcParams.update({'font.size': 34})
plt.rcParams.update({'font.family': "Times New Roman"})

ax1.semilogx(Ystar,scaled_Prod_K
  ↪ ,'g',markersize=10,linewidth=5,markevery=5,label='Production')
ax1.semilogx(Ystar,scaled_Delt_K_orig
    ↪ ,'r-',markersize=10,linewidth=5,markevery=5,label=r'$\delta_{ML}$
    ↪ (Initial)')
ax1.semilogx(Ystar,scaled_Delt_K_corrected,'b', markersize=10,
  ↪ linewidth=5, markevery=5   ,label=r'$\delta_f$     (Corrected)')
ax1.semilogx(Ystar,scaled_Delt_K_true
    ↪ ,'k:',markersize=10,linewidth=5,markevery=5,label=   r'$\delta^*$
    ↪ (Real)')

ax1.set_ylabel(r'k-eq. [1/$S_k$]')
ax1.set_xlabel(r'$Y^*$')

for axi in axarr:
    axi.grid()
    axi.yaxis.set_major_locator(plt.MaxNLocator(4))
    # axi.xaxis.set_major_locator(plt.LogLocator(numticks=3))
    box = axi.get_position()
    axi.set_position([box.x0, box.y0, box.width * 0.8, box.height])
    # Put a legend to the right of the current axis
    axi.legend(loc='center left', bbox_to_anchor=(1, 0.5))
# Source Legend Formatting Code:
  ↪ https://stackoverflow.com/questions/4700614/
  ↪ how-to-put-the-legend-out-of-the-plot
fig.set_size_inches(fig.get_size_inches()*1.1*np.array([2.0,1.0]))
print('Font, Size:
  ↪ ',plt.rcParams[”font.family”],plt.rcParams[”font.size”])
plt.show()
#
```

B.2. Extended K-fold Validation Results for the Predicted $\boldsymbol{\delta_k}$ Corrections

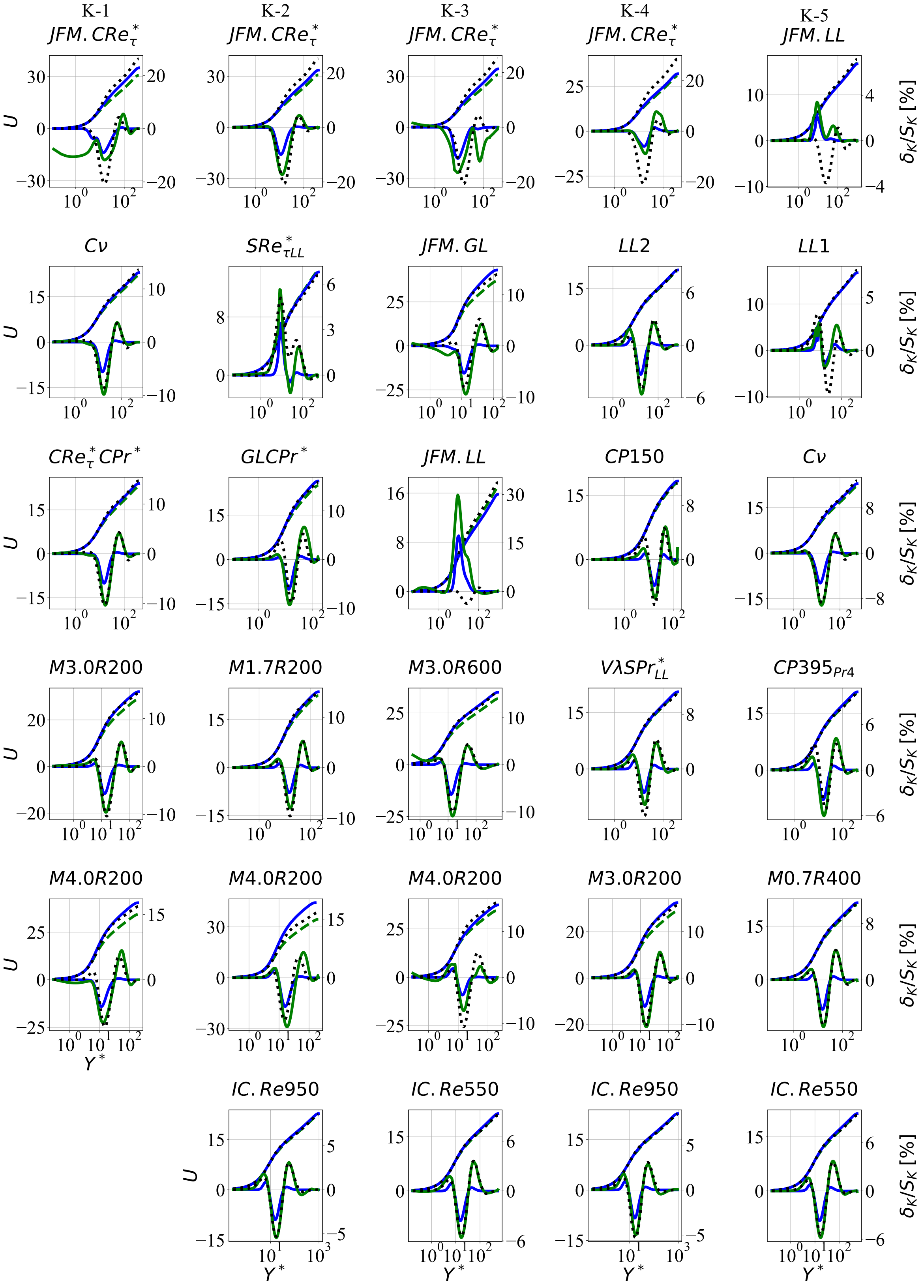


Figure B.1: Results obtained for the K-fold validation trials {1-5} during the prediction of the independent sets of $\boldsymbol{\delta_k}$ corrections for the MK turbulence model. The upper curves represent the initial RANS velocity profiles (green dashed lines), the data-augmented velocity predictions (blue solid lines) and the reference DNS data (red dotted lines). The lower curves present the initial predictions made by the Neural Network system (green solid lines), the results obtained after applying an intelligent relaxation factor of 0.5 (blue solid lines) and the ground-truth labels for the Field Inversion values (red dotted lines).

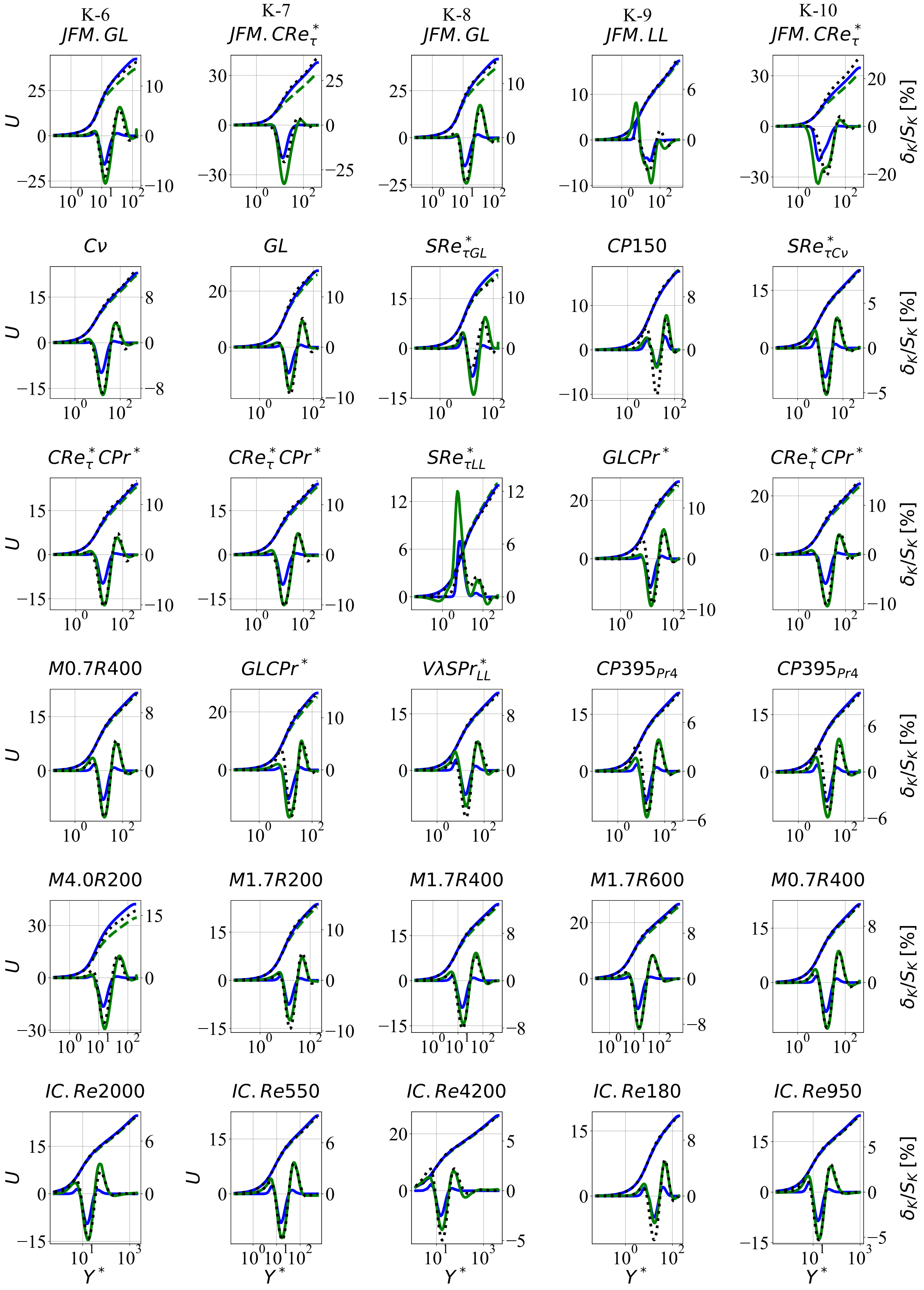


Figure B.2: Results obtained for the K-fold validation trials {6-10} during the prediction of the independent sets of δ_k corrections for the MK turbulence model. The upper curves represent the initial RANS velocity profiles (green dashed lines), the data-augmented velocity predictions (blue solid lines) and the reference DNS data (red dotted lines). The lower curves present the initial predictions made by the Neural Network system (green solid lines), the results obtained after applying an intelligent relaxation factor of 0.5 (blue solid lines) and the ground-truth labels for the Field Inversion values (red dotted lines).

B.3. Extended K-fold Validation Results for the Predicted $\boldsymbol{\delta_\epsilon}$ Corrections

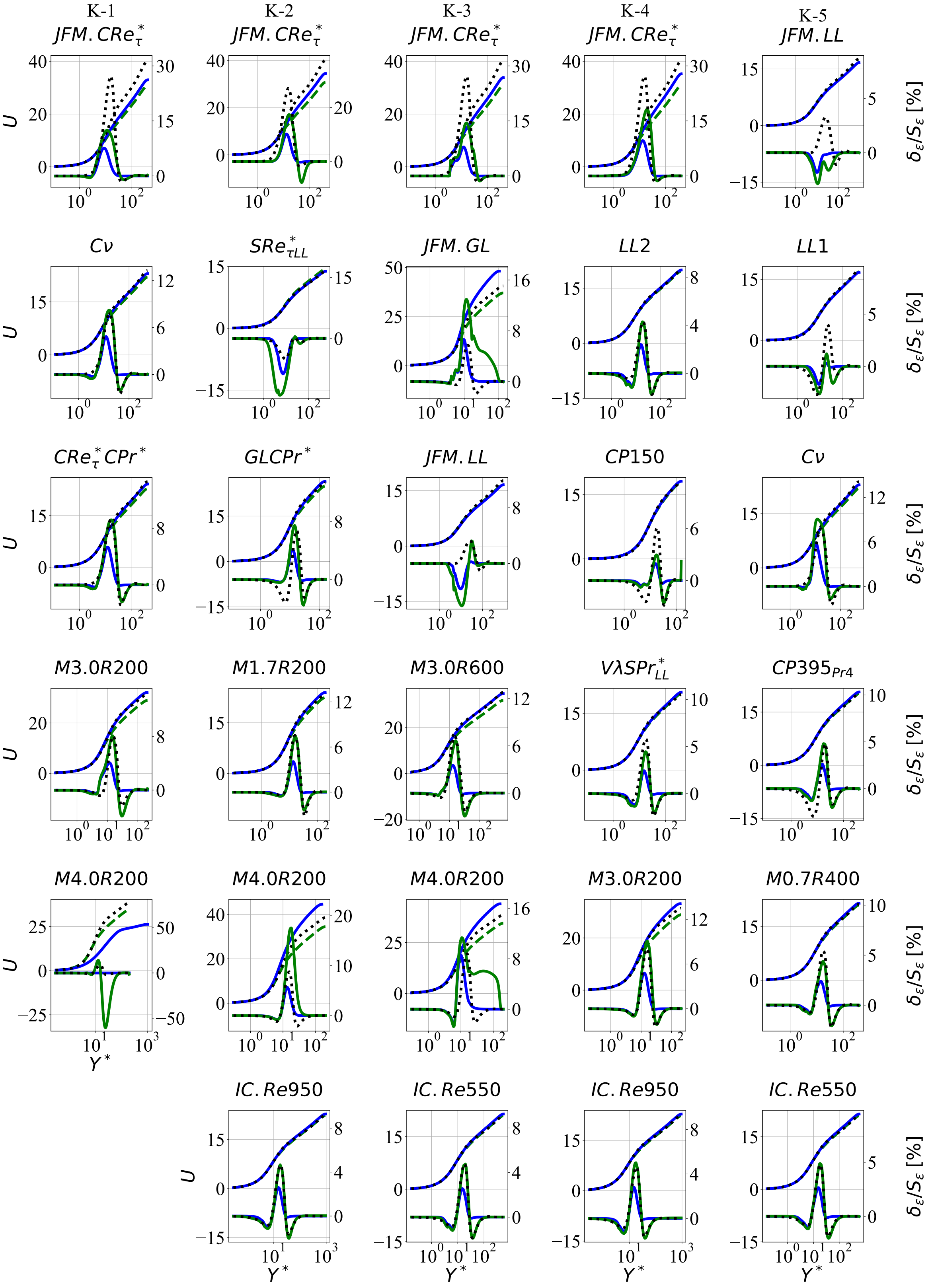


Figure B.3: Results obtained for the K-fold validation trials {1-5} during the prediction of the independent sets of $\boldsymbol{\delta_\epsilon}$ corrections for the MK turbulence model. The upper curves represent the initial RANS velocity profiles (green dashed lines), the data-augmented velocity predictions (blue solid lines) and the reference DNS data (red dotted lines). The lower curves present the initial predictions made by the Neural Network system (green solid lines), the results obtained after applying an intelligent relaxation factor of 0.5 (blue solid lines) and the ground-truth labels for the Field Inversion values (red dotted lines).

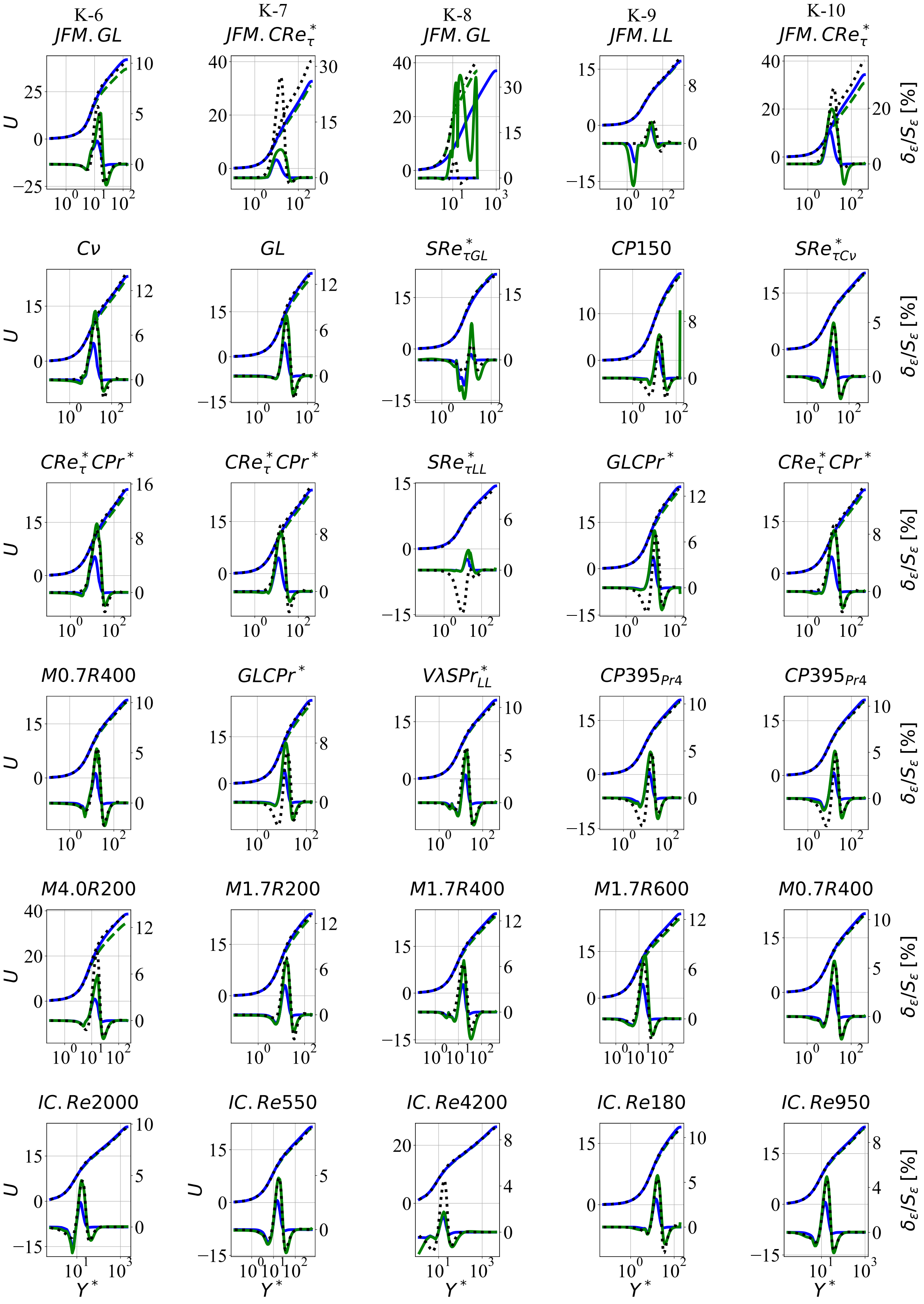


Figure B.4: Results obtained for the K-fold validation trials {6-10} during the prediction of the independent sets of $\boldsymbol{\delta_\epsilon}$ corrections for the MK turbulence model. The upper curves represent the initial RANS velocity profiles (green dashed lines), the data-augmented velocity predictions (blue solid lines) and the reference DNS data (red dotted lines). The lower curves present the initial predictions made by the Neural Network system (green solid lines), the results obtained after applying an intelligent relaxation factor of 0.5 (blue solid lines) and the ground-truth labels for the Field Inversion values (red dotted lines).

B.4. Log-Space Features Created by the Trained Neural Networks

B.4.1. Neural Network Weights for the Predicted δ_k Corrections

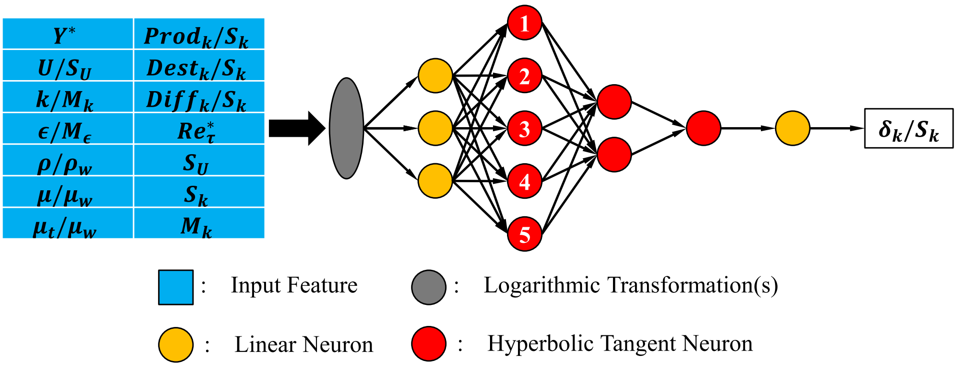


Figure B.5: Neural Network architecture created to predict the Field Inversion corrections ($\boldsymbol{\delta}_\mathbf{k}$) required by the MK turbulence model.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	-4.49E-01	1.38E+00	-9.85E+00	-7.82E-01	-1.03E+00	7.62E-01	3.84E+00	7.47E-01	-6.91E-01	6.76E+00
u/S_U	3.77E-01	-6.68E-01	1.04E+01	-1.26E+00	1.14E-01	-9.93E-01	-4.00E+00	-5.93E-01	-9.10E-01	-4.69E+00
k/M_k	1.14E-01	-1.36E+00	-2.40E+00	-3.02E-01	2.10E-01	-1.30E+00	-1.06E+01	-2.45E+00	-5.39E+00	1.02E+00
ϵ/M_ϵ	7.37E-02	1.09E+00	-5.98E-01	3.71E-01	6.12E-02	1.82E+00	7.38E+00	1.29E+00	-2.82E+00	3.61E+00
ρ/ρ_w	-6.91E-02	3.87E-01	8.02E+00	-4.51E-01	9.75E-02	1.55E+00	2.61E+00	1.86E-01	-5.45E+00	6.24E+00
μ/μ_w	-9.57E-02	2.76E-01	-9.63E+00	4.40E-02	-1.41E-01	-3.31E-01	-2.82E+00	-9.68E-01	-3.14E+00	-1.70E+00
μ_t/μ_w	1.49E-01	8.03E-02	2.19E+00	5.72E-01	4.57E-01	2.78E-01	3.38E+00	7.06E-01	3.37E+00	-8.82E-01
$Prod_k/S_k$	-2.09E-01	-4.48E-01	1.62E-02	9.26E-03	-8.69E-03	-5.29E-01	6.93E-01	-3.99E-02	9.65E+00	-2.78E+00
$Dest_k/S_k$	-2.28E-01	1.66E+00	-1.82E+00	1.77E-01	-2.35E-01	1.23E+00	1.33E+00	1.02E+00	-3.26E+00	2.12E+00
$Diff_k/S_k$	2.98E-03	8.65E-03	4.80E-02	-9.92E-03	-6.73E-03	-3.26E-02	-1.22E-01	-1.18E-02	1.31E-02	5.23E-02
Re_τ^*	4.26E-01	-7.38E-01	-5.55E+00	9.24E-01	4.84E-01	-1.89E+00	-7.23E+00	-1.62E+00	-2.18E+00	-9.01E+00
S_U	5.68E-02	7.49E-01	1.62E+01	-1.88E-01	3.40E-01	7.59E-01	6.56E+00	1.37E+00	2.01E+00	-1.75E+00
S_k	-4.15E-01	6.08E-01	5.13E+00	-1.10E+00	-6.87E-01	1.68E+00	6.52E+00	1.56E+00	1.92E+00	9.40E+00
M_k	1.91E-01	-9.09E-01	-6.28E+00	4.96E-02	1.11E+00	-1.27E+00	-9.56E+00	-2.59E+00	-6.38E+00	-5.42E+00

Table B.1: Neural Network weights in log-space for each input feature perceived by the Neuron (N-1) presented in Figure B.5.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	1.29E+00	3.95E-01	-5.25E-01	2.44E+00	-6.89E+00	-1.92E+00	2.90E-01	1.08E+00	-1.16E-01	3.06E+00
U/S_U	-1.39E+00	-8.62E-01	-4.38E-01	3.65E+00	1.24E+00	4.21E+00	-5.57E-01	1.13E-01	-1.83E+00	-1.09E+00
k/M_k	1.05E+00	-1.51E+00	3.19E-01	8.72E-01	9.96E+00	-2.38E+00	8.91E+00	-3.42E+00	-5.08E+00	1.96E+00
ϵ/M_ϵ	-2.87E-01	1.39E+00	-1.76E-01	-7.90E-01	-6.41E+00	2.30E-01	-6.48E+00	1.38E+00	-3.74E+00	2.37E-02
ρ/ρ_w	7.96E-01	2.81E-01	-9.51E-01	1.32E+00	-1.37E+00	-1.35E+00	-4.21E+00	2.77E-01	-6.26E+00	2.46E+00
μ/μ_w	8.79E-01	-3.38E-01	6.25E-01	-1.86E-01	4.76E+00	-1.92E+00	5.28E+00	-1.61E+00	-2.32E+00	-5.20E-01
μ_t/μ_w	-9.93E-01	7.01E-01	1.19E-01	-1.95E+00	-1.12E+00	1.34E+00	-3.40E+00	8.59E-01	3.03E+00	-1.98E+00
$Prod_k/S_k$	-1.70E-01	-6.10E-01	3.35E-02	1.21E-01	5.75E+00	7.56E-01	2.62E-03	-2.20E-02	1.10E+01	-1.28E+00
$Dest_k/S_k$	3.42E-01	1.36E+00	3.01E-01	-1.62E-01	-9.17E+00	5.91E-01	8.66E-02	1.29E+00	-4.06E+00	6.22E-01
$Diff_k/S_k$	-7.75E-03	1.07E-02	-2.61E-02	2.83E-02	1.35E-01	-4.63E-02	8.26E-02	-1.24E-02	2.29E-02	1.92E-02
Re_τ^*	-4.25E-01	-6.03E-01	1.26E+00	-3.25E+00	6.51E+00	2.06E-01	6.69E+00	-2.83E+00	-1.38E+00	-1.42E+00
S_U	-1.09E+00	4.01E-01	-5.19E-01	-1.16E+00	-4.57E+00	3.97E+00	-9.31E+00	2.56E+00	1.60E+00	-8.27E-01
S_k	4.31E-01	4.57E-01	-1.32E+00	4.10E+00	-8.10E+00	-6.33E-01	-6.11E+00	2.75E+00	1.08E+00	1.97E+00
M_k	3.31E-01	-1.28E+00	1.46E+00	-1.48E+00	1.77E+01	-7.38E-02	8.97E+00	-3.44E+00	-5.90E+00	-3.85E+00

Table B.2: Neural Network weights in log-space for each input feature perceived by the Neuron (N-2) presented in Figure B.5.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	3.88E+00	1.95E-01	-1.07E+00	2.89E+00	1.22E+00	-8.92E+00	2.59E+01	-2.67E-01	-1.01E+00	2.39E+00
U/S_U	-4.04E+00	1.06E+00	-2.29E-01	6.91E-01	7.40E-01	7.67E+00	-4.79E+01	1.06E+00	1.35E-01	-1.05E+00
k/M_k	3.32E+00	1.72E+00	3.38E-01	-1.16E-01	3.93E-01	3.21E+00	-3.14E+00	1.19E+00	-6.59E-01	1.86E+00
ϵ/M_ϵ	-8.92E-01	-1.69E+00	-2.10E-01	-4.41E-01	-1.29E+00	-4.02E+00	-1.83E+00	-9.48E-01	3.37E-01	-4.83E-03
ρ/ρ_w	2.56E+00	-1.99E-01	-9.43E-01	2.16E-01	-6.18E-01	-1.56E+00	-3.37E+01	-1.40E-01	2.82E-01	2.35E+00
μ/μ_w	2.62E+00	6.71E-01	5.47E-01	2.34E-02	2.63E-01	-3.58E+00	3.48E+01	3.20E-01	-1.14E+00	-7.01E-01
μ_t/μ_w	-3.07E+00	-1.16E+00	2.16E-01	-8.24E-01	-9.34E-01	2.70E+00	1.81E+00	-4.88E-01	9.18E-01	-1.59E+00
$Prod_k/S_k$	-6.86E-01	7.09E-01	4.72E-02	8.20E-03	7.93E-01	2.32E-01	6.46E+00	4.83E-02	3.67E-01	-1.18E+00
$Dest_k/S_k$	1.03E+00	-1.25E+00	3.81E-01	-1.32E-01	-6.44E-01	-1.50E+00	1.37E+01	-5.96E-01	4.10E-01	6.35E-01
$Diff_k/S_k$	-2.40E-02	-1.31E-02	-3.20E-02	1.16E-02	4.18E-02	-4.53E-02	-3.83E-01	9.27E-03	-7.89E-03	1.87E-02
Re_τ^*	-1.28E+00	5.69E-01	1.48E+00	-1.84E+00	-8.86E-01	5.64E+00	2.00E+01	2.90E-01	-1.01E+00	-1.38E+00
S_U	-3.24E+00	-7.63E-02	-3.16E-01	1.77E-01	-3.14E-01	5.69E+00	-4.97E+01	-4.00E-01	7.46E-01	-7.72E-01
S_k	1.31E+00	-3.87E-01	-1.56E+00	2.05E+00	1.07E+00	-6.19E+00	-2.07E+01	-2.20E-01	8.66E-01	1.84E+00
M_k	9.85E-01	1.43E+00	1.66E+00	-6.70E-01	-2.53E-01	6.19E+00	1.72E+01	1.38E+00	-7.41E-01	-3.45E+00

Table B.3: Neural Network weights in log-space for each input feature perceived by the Neuron (N-3) presented in Figure B.5.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	-1.88E+00	1.05E+00	1.67E+00	1.80E+00	-1.25E+00	2.10E+00	-2.10E+00	-1.03E+00	-9.27E-01	3.79E+00
U/S_U	-6.65E-01	4.75E-01	-2.28E+00	-2.24E+00	-1.10E+00	-4.36E-01	-1.40E-01	1.72E+00	8.46E-01	-1.04E+00
k/M_k	4.77E-01	4.51E-01	5.31E-01	-8.99E-01	-2.55E-01	-2.79E+00	4.17E-01	3.55E+00	-1.34E+00	-1.45E+00
ϵ/M_ϵ	5.44E-01	-6.87E-01	4.35E-01	3.70E-01	1.50E+00	1.97E+00	-4.33E-01	-2.26E+00	5.82E-01	1.92E+00
ρ/ρ_w	-1.51E+00	3.00E-01	-1.47E+00	-8.36E-01	7.45E-01	1.76E-01	-1.63E+00	-2.76E-01	2.11E-02	5.17E-01
μ/μ_w	8.01E-01	2.73E-01	2.46E+00	1.35E-01	-3.64E-02	1.30E-02	4.78E-01	1.19E+00	-1.61E+00	1.14E+00
μ_t/μ_w	8.62E-01	-9.51E-01	-5.66E-01	3.31E-01	9.22E-01	-6.49E-02	7.81E-01	-1.15E+00	1.10E+00	-4.49E-01
$Prod_k/S_k$	2.16E-01	-1.01E-02	1.84E-02	4.42E-02	-8.57E-01	2.09E-01	-7.68E-02	8.92E-02	2.29E-01	-3.33E-01
$Dest_k/S_k$	-1.44E+00	2.71E-01	7.69E-01	2.94E-01	6.75E-01	1.17E+00	-9.70E-02	-1.61E+00	4.01E-01	1.03E-01
$Diff_k/S_k$	2.21E-02	-5.93E-03	-6.24E-03	-9.68E-03	-4.82E-02	-2.02E-02	-1.01E-02	2.09E-02	-9.78E-03	2.93E-03
Re_τ^*	2.94E+00	-7.34E-03	1.33E+00	-1.24E-01	1.20E+00	-2.28E+00	2.29E+00	1.83E+00	-1.40E+00	-2.71E+00
S_U	-1.57E+00	1.18E+00	-4.85E+00	-3.61E-01	-4.11E-02	5.68E-01	-1.43E+00	-1.48E+00	9.15E-01	-1.18E-01
S_k	-3.06E+00	1.49E-01	-1.13E+00	-6.68E-04	-1.37E+00	2.17E+00	-2.29E+00	-1.72E+00	1.38E+00	2.82E+00
M_k	2.25E+00	-4.79E-01	1.09E+00	-8.39E-01	3.25E-01	-2.46E+00	1.66E+00	3.92E+00	-1.70E+00	-2.79E-01

Table B.4: Neural Network weights in log-space for each input feature perceived by the Neuron (N-4) presented in Figure B.5.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	1.37E+00	4.70E+00	-1.67E+00	-1.51E+00	-5.03E-01	-1.35E+00	-4.19E+00	3.76E-01	-7.29E-01	1.34E+00
U/S_U	3.78E+00	1.52E+00	3.59E-01	-2.63E+00	-7.01E-03	4.36E-01	1.17E+00	1.53E+00	-8.78E-01	2.99E-01
k/M_k	-3.69E+00	4.37E+00	1.90E-01	-6.50E-01	1.22E-01	1.79E+00	1.56E-01	-3.43E-01	2.97E-01	-3.30E-01
ϵ/M_ϵ	-4.77E-01	-3.28E+00	-1.65E-01	6.53E-01	6.81E-02	-1.41E+00	-2.09E-01	-4.57E-01	-2.24E-01	6.85E-02
ρ/ρ_w	1.03E+00	-3.92E+00	-3.92E-01	-9.48E-01	6.97E-02	-3.35E-01	-1.42E+00	-1.49E-01	3.04E-01	-4.87E-01
μ/μ_w	-3.42E+00	1.24E+01	6.63E-02	1.16E-01	-3.10E-02	2.03E-02	-5.75E-01	-3.85E-01	-1.73E-01	8.73E-01
μ_t/μ_w	4.81E-01	-3.26E+00	3.07E-01	1.26E+00	2.20E-01	1.18E-02	1.46E+00	-2.51E-01	4.26E-01	-7.28E-01
$Prod_k/S_k$	4.77E-01	7.48E+00	5.06E-02	-3.34E-02	-1.74E-02	-4.02E-02	-3.10E-01	5.63E-02	7.54E-01	9.59E-02
$Dest_k/S_k$	2.35E+00	-4.57E+00	3.43E-01	2.39E-01	-1.05E-01	-8.50E-01	-5.54E-01	-4.78E-02	1.32E-01	-2.84E-01
$Diff_k/S_k$	-2.88E-02	-1.00E-02	-2.68E-02	-2.01E-02	-4.42E-03	1.59E-02	-1.13E-02	6.03E-03	-1.71E-03	-5.34E-03
Re_τ^*	-5.38E+00	4.76E-02	1.13E+00	2.04E+00	2.86E-01	1.60E+00	2.72E+00	-1.32E+00	-1.68E-01	4.40E-01
S_U	5.45E+00	-1.32E+01	3.05E-01	2.33E-01	1.05E-01	-4.00E-01	-2.83E-01	4.51E-01	2.96E-01	1.02E-01
S_k	5.57E+00	-1.24E+00	-1.20E+00	-2.52E+00	-3.83E-01	-1.51E+00	-2.74E+00	1.43E+00	-8.08E-02	-2.81E-01
M_k	-5.41E+00	2.41E+01	1.17E+00	5.42E-01	5.43E-01	1.61E+00	1.60E+00	-1.28E-01	5.68E-01	-1.13E-01

Table B.5: Neural Network weights in log-space for each input feature perceived by the Neuron (N-5) presented in Figure B.5.

B.4.2. Neural Network Weights for the Predicted δ_ϵ Corrections

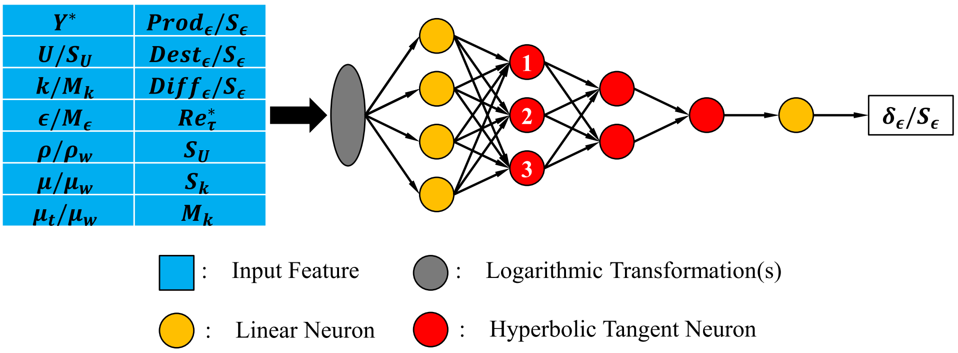

Y^*
$Prod_\epsilon/S_\epsilon$
U/S_U
$Dest_\epsilon/S_\epsilon$
k/M_k
$Diff_\epsilon/S_\epsilon$
ϵ/M_ϵ
Re_τ^*
ρ/ρ_w
S_U
μ/μ_w
S_k
μ_t/μ_w
M_k
1
2
3
$\delta_\epsilon/S_\epsilon$
: Input Feature
: Logarithmic Transformation(s)
: Linear Neuron
: Hyperbolic Tangent Neuron

Figure B.6: Neural Network architecture created to predict the Field Inversion corrections (δ_ϵ) required by the MK turbulence model.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	5.10E-01	1.13E+00	1.88E+01	-4.18E-01	-9.97E+00	-3.76E+00	-8.56E-01	5.38E+00	-9.19E+00	1.62E-01
u/S_U	3.40E-01	1.14E+00	-2.59E+01	-8.26E-01	1.57E+00	1.93E+00	-3.14E-01	1.61E+00	6.65E+00	3.06E+00
k/M_k	-1.63E+00	1.12E+00	3.16E+00	-1.95E+00	7.93E+00	3.93E+00	1.28E+00	-1.05E+00	-3.76E+00	-1.03E+00
ϵ/M_ϵ	4.52E+00	1.54E-02	-6.01E-01	4.21E+00	-7.01E+00	-1.05E+01	-1.33E-01	1.19E+01	4.41E+00	8.85E-02
ρ/ρ_w	3.92E+00	1.64E-01	-8.12E+00	1.61E+00	-2.26E+00	-6.03E+00	-4.33E-01	1.84E+01	1.44E+00	3.76E-01
μ/μ_w	-5.95E-01	9.78E-01	9.92E+00	-1.17E-01	4.19E-01	1.67E-01	-2.15E-01	-1.43E+00	-2.87E+00	-9.34E-01
μ_t/μ_w	-5.86E-01	-8.37E-01	-8.49E-01	5.65E-01	-2.48E-01	1.86E-01	9.02E-02	-1.42E+00	2.53E+00	1.80E-02
$Prod_\epsilon/S_\epsilon$	5.62E-02	-7.21E-01	-1.64E+00	-8.71E-02	2.61E-01	-6.26E-02	9.04E-02	-2.00E-01	-2.49E-01	3.65E-01
$Dest_\epsilon/S_\epsilon$	-2.87E+00	2.19E-01	6.49E+00	-1.55E+00	-8.37E-01	4.80E+00	-1.48E+00	-6.10E+00	-2.72E+00	-1.15E+00
$Diff_\epsilon/S_\epsilon$	-3.24E-03	1.93E-02	7.81E-02	-7.58E-03	8.73E-03	1.86E-01	4.99E-02	-1.92E-01	-1.84E-03	3.55E-02
Re_τ^*	-1.13E+00	3.44E-01	9.52E-01	2.99E-01	9.44E+00	6.65E-01	2.30E+00	2.24E+00	4.12E+00	-9.25E-01
S_U	1.03E+00	-1.81E+00	-2.20E+01	-1.38E+00	6.32E-01	1.96E+00	-1.30E+00	-2.17E+00	6.50E+00	2.62E+00
S_k	1.08E+00	-4.74E-02	-1.26E+00	-2.64E-01	-9.57E+00	-7.86E-01	-2.27E+00	-2.14E+00	-4.23E+00	9.21E-01
M_k	-3.98E-01	1.42E+00	-1.45E+00	-5.92E-01	6.58E+00	2.81E+00	1.53E+00	1.03E+00	2.69E+00	1.27E+00

Table B.6: Neural Network weights in log-space for each input feature perceived by the Neuron (N-1) presented in Figure B.6.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	-1.06E-01	-8.95E-01	-4.63E+00	-7.48E+00	6.55E+00	-1.39E+00	7.42E-01	5.43E+00	-7.17E+00	-1.34E+00
u/S_U	2.71E+00	-1.47E+00	1.09E+00	-1.81E+01	-5.60E-01	6.95E-01	-4.55E-01	-2.21E-01	5.24E+00	8.89E-02
k/M_k	-2.66E+00	-9.32E-01	-1.20E+00	-1.88E+00	-5.46E+00	1.59E+00	1.60E+00	-3.90E+00	-3.08E+00	-1.02E+00
ϵ/M_ϵ	4.16E+00	8.10E-01	-9.00E-01	-2.09E+01	4.33E+00	-5.99E+00	-7.74E-01	5.44E-01	3.95E+00	1.33E+00
ρ/ρ_w	2.84E+00	5.44E-01	-2.51E+00	-3.50E+01	1.73E+00	-4.03E+00	-4.05E-01	9.97E-01	1.46E+00	4.75E-01
μ/μ_w	-1.56E+00	-8.72E-01	-1.77E+00	4.19E+01	-4.98E-01	3.00E-01	1.12E+00	-1.10E+00	-2.16E+00	-6.63E-01
μ_t/μ_w	-5.83E-03	8.45E-01	2.31E+00	9.64E+00	-8.64E-02	2.94E-01	-7.95E-01	-6.09E-01	1.89E+00	6.02E-01
$Prod_\epsilon/S_\epsilon$	-2.46E-01	9.54E-02	-7.05E-02	2.97E+00	1.21E-01	5.90E-01	2.72E-01	1.40E-01	-1.68E-01	-5.78E-02
$Dest_\epsilon/S_\epsilon$	-2.25E+00	1.74E-01	4.54E-01	1.01E+01	5.82E-01	2.44E+00	-5.94E-01	2.09E+00	-2.39E+00	-1.63E-01
$Diff_\epsilon/S_\epsilon$	-1.66E-03	-1.62E-02	6.08E-02	-9.29E-02	-3.15E-03	2.57E-01	2.87E-02	-1.76E-01	-2.47E-03	-3.16E-02
Re_τ^*	-1.08E+00	-4.87E-02	1.60E+00	1.32E+01	-7.24E+00	-4.98E-01	1.21E+00	-8.34E+00	3.39E+00	5.37E-01
S_U	2.73E+00	1.46E+00	7.46E-01	-1.11E+01	5.79E-01	3.13E-01	-1.59E+00	1.80E+00	5.10E+00	1.23E+00
S_k	1.14E+00	-2.42E-01	-1.73E+00	-2.58E+01	7.29E+00	4.46E-01	-1.14E+00	8.46E+00	-3.44E+00	-6.67E-01
M_k	-1.62E+00	-1.47E+00	-8.73E-01	9.74E+01	-4.76E+00	1.10E+00	1.47E+00	-5.61E+00	1.78E+00	-3.64E-01

Table B.7: Neural Network weights in log-space for each input feature perceived by the Neuron (N-2) presented in Figure B.6.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	5.30E-01	-2.96E-01	-4.49E+00	1.73E+00	-9.92E+00	1.40E+00	-4.78E-01	7.54E+00	6.88E+00	-1.54E+00
u/S_U	2.31E+00	-2.17E+00	2.70E+00	-1.38E+00	1.34E+00	-7.74E-01	-2.83E-01	-1.42E+00	-5.08E+00	2.02E-01
k/M_k	-4.34E+00	-8.01E-01	2.83E+00	2.42E-01	8.43E+00	6.75E-01	1.43E+00	-2.32E+00	9.28E-01	-1.75E+00
ϵ/M_ϵ	6.01E+00	9.73E-01	-1.27E+01	-3.99E-01	-8.06E+00	-4.22E-01	-3.96E-01	-1.57E+00	1.40E+00	2.18E+00
ρ/ρ_w	3.38E+00	1.12E+00	-4.19E+00	-5.90E-01	-3.70E+00	-3.05E-01	-4.98E-01	1.25E+00	1.22E+00	7.37E-01
μ/μ_w	-1.99E+00	-3.93E-01	-3.71E+00	8.70E-01	3.86E-01	9.12E-01	-1.43E-01	-4.59E-01	1.81E+00	-8.22E-01
μ_t/μ_w	5.43E-01	7.11E-01	1.24E+00	-3.15E-01	4.14E-01	-7.05E-01	1.12E-02	-1.42E+00	-1.79E+00	7.87E-01
$Prod_\epsilon/S_\epsilon$	-1.05E-01	1.04E-01	3.14E+00	3.98E-01	-2.05E-01	5.01E-02	4.78E-01	1.26E-01	1.54E-01	7.46E-01
$Dest_\epsilon/S_\epsilon$	-2.75E+00	2.87E-01	3.38E+00	3.99E-01	2.01E-01	-2.99E-03	-1.74E+00	3.67E+00	-2.71E-01	-1.33E+00
$Diff_\epsilon/S_\epsilon$	-1.57E-03	-1.53E-02	-4.34E-01	-5.73E-03	4.32E-03	4.68E-03	5.30E-02	-2.74E-01	2.15E-03	-4.37E-02
Re_τ^*	-2.25E+00	-4.55E-01	-3.08E+00	-7.29E-01	1.05E+01	3.66E-01	1.98E+00	-9.78E+00	-1.74E+00	5.21E-01
S_U	2.25E+00	1.23E+00	9.54E+00	-1.80E+00	-4.43E-01	-6.52E-01	-1.57E+00	2.27E+00	-4.75E+00	1.53E+00
S_k	2.32E+00	1.36E-01	2.37E+00	7.79E-01	-1.06E+01	-3.57E-01	-1.91E+00	9.72E+00	1.95E+00	-6.72E-01
M_k	-3.14E+00	-2.10E+00	-2.23E+00	-4.33E-01	6.57E+00	4.11E-01	1.54E+00	-5.31E+00	-4.24E+00	-5.18E-01

Table B.8: Neural Network weights in log-space for each input feature perceived by the Neuron (N-3) presented in Figure B.6.

C

Additional Results for the Alternative Machine Learning Formulation Presented in Chapter 5

C.1. Feature Weights Perceived by the First Layer of Hyperbolic Tangent Neurons

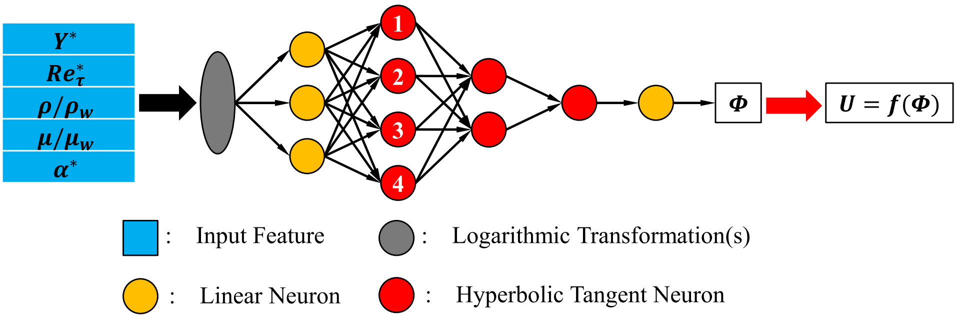


Figure C.1: Neural Network architecture chosen to predict the values of Φ in the channel flows studied.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	-2.48E-01	-5.15E-01	1.34E+00	-2.84E-01	1.95E+00	-7.89E-01	2.55E+00	6.65E-01	3.43E-01	1.86E+00
Re_τ^*	3.16E-01	1.68E-01	1.88E-01	-2.91E-01	4.12E-01	4.78E-01	4.76E-01	-8.57E-04	-6.10E-03	2.01E-01
μ/μ_w	1.04E-01	-7.79E-02	-2.54E-01	3.89E-02	-9.00E-02	-5.80E-01	2.46E-01	-2.71E-02	4.51E-04	-1.90E-01
ρ/ρ_w	-5.86E-01	3.81E-01	3.59E-01	-3.49E-01	4.93E-01	-2.12E+00	-9.21E-01	8.53E-02	-3.82E-02	4.83E-01
α^*	1.05E+00	-8.16E-01	-3.76E-01	8.10E-01	-2.86E-01	2.58E+00	1.82E+00	-1.17E-01	6.58E-02	-6.03E-01

Table C.1: Neural Network weights in log-space for each input feature perceived by the Neuron (N-1) presented in Figure C.1.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	2.94E-01	-9.28E-01	1.01E+00	-4.27E-01	-5.29E+00	9.23E-01	4.05E-01	2.79E-01	1.21E+00	9.40E-01
Re_τ^*	-5.99E-02	2.03E-01	-1.01E+00	6.83E-02	-4.92E+00	-4.46E-01	-1.36E-03	-2.91E-01	-5.51E-02	-8.89E-01
μ/μ_w	-5.32E-02	-8.63E-02	-8.47E-02	-1.23E-03	-7.84E-01	2.93E-01	7.25E-03	-1.77E-02	1.97E-02	-1.03E-01
ρ/ρ_w	2.29E-01	4.53E-01	-9.76E-01	5.68E-02	-2.29E+00	5.40E-01	-4.46E-02	-5.88E-01	1.53E-01	5.53E-01
α^*	-3.85E-01	-1.02E+00	2.05E+00	-1.24E-01	-6.17E+00	-4.07E-01	8.76E-02	1.14E+00	-2.59E-01	-9.90E-01

Table C.2: Neural Network weights in log-space for each input feature perceived by the Neuron (N-2) presented in Figure C.1.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	1.78E-02	-7.64E+00	3.84E-01	2.81E+00	7.88E-01	-5.39E-01	1.23E+01	-3.11E-01	-1.30E+01	3.86E-01
Re_τ^*	-1.48E-01	-1.60E+00	1.09E-02	4.01E+00	-1.94E-02	5.56E-01	-6.74E-01	1.69E-02	-3.23E+00	2.25E-03
μ/μ_w	-1.23E-01	9.92E-01	-3.97E-02	-7.09E+00	1.54E-02	-5.05E-01	-1.13E+01	-5.57E-03	-3.29E-01	4.30E-03
ρ/ρ_w	5.88E-01	-3.23E+00	3.44E-02	-2.22E+00	-3.33E-02	-1.36E+00	4.16E+00	1.38E-02	-5.17E+00	-3.45E-02
α^*	-1.01E+00	4.17E+00	-1.60E-02	2.34E+00	7.14E-02	1.44E+00	-8.16E+00	-2.12E-02	5.07E+00	6.22E-02

Table C.3: Neural Network weights in log-space for each input feature perceived by the Neuron (N-3) presented in Figure C.1.

	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
	K-1	K-2	K-3	K-4	K-5	K-6	K-7	K-8	K-9	K-10
Y^*	-1.58E+00	1.18E-01	-3.14E-01	-1.02E+00	3.42E-01	8.08E-01	1.14E+00	6.98E-01	1.43E+00	-9.09E+00
Re_τ^*	-1.35E-02	-3.27E-01	2.45E-02	-1.20E-01	-4.54E-03	-2.16E-01	-1.08E+00	1.72E-01	-3.53E+00	7.52E-01
μ/μ_w	1.77E-01	1.88E-01	-1.16E-01	2.67E-01	-1.09E-02	3.88E-02	-2.33E-01	1.20E+00	-7.02E-02	6.74E+00
ρ/ρ_w	-5.96E-01	-8.99E-01	1.20E-01	6.14E-02	2.05E-02	-2.71E-01	1.22E+00	-7.54E-01	-2.80E+00	-2.23E+01
α^*	9.17E-01	1.96E+00	-8.17E-02	-2.60E-02	-5.79E-02	5.26E-01	-2.42E+00	-2.45E-01	1.42E+00	3.24E+01

Table C.4: Neural Network weights in log-space for each input feature perceived by the Neuron (N-4) presented in Figure C.1.

C.2. Extended Results of the K-fold Validation Runs

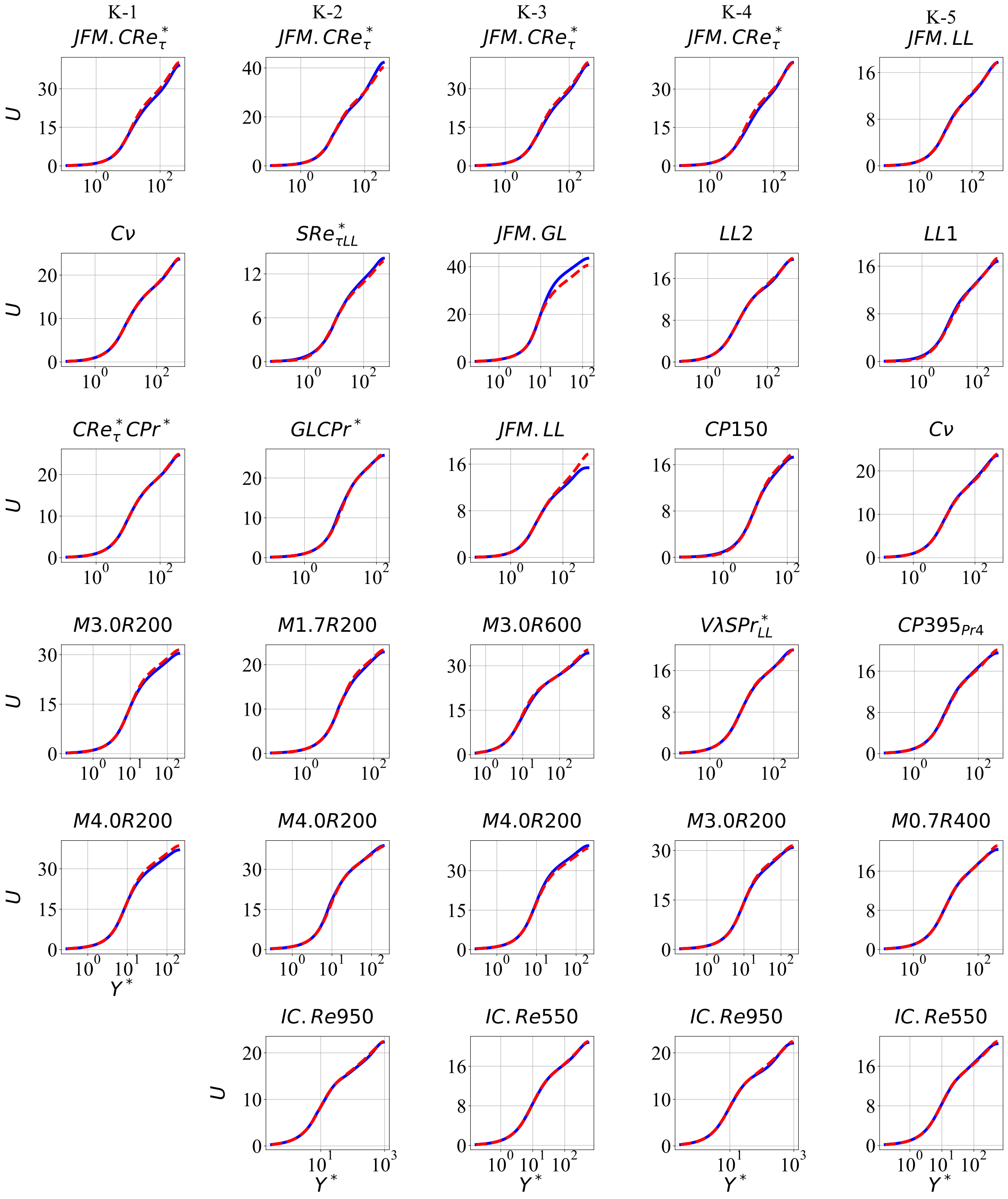


Figure C.2: Results obtained for the velocity profiles considered in the test sets of the K-fold trial runs {1-5} using the Alternative Machine formulation presented in Chapter 5.

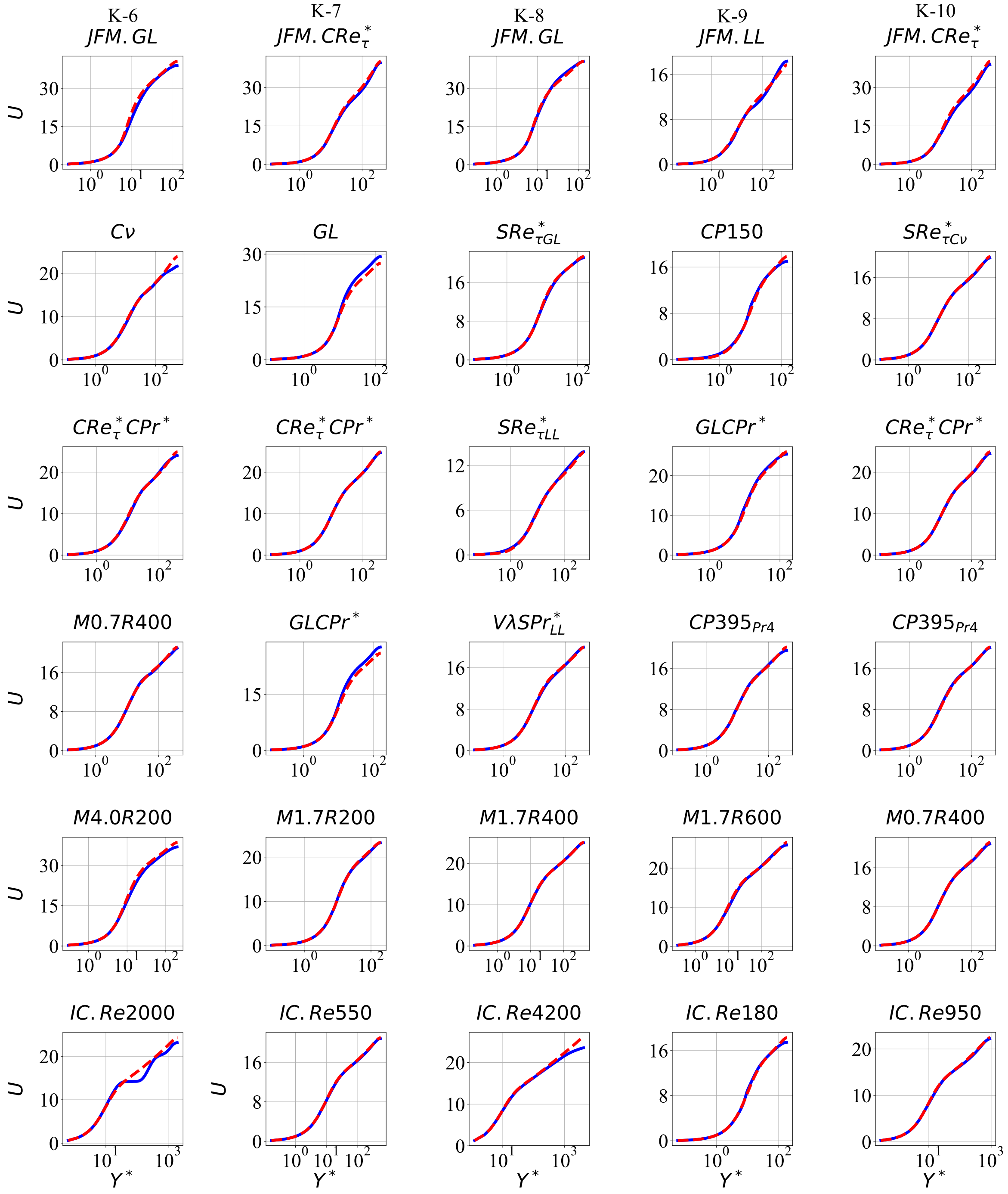


Figure C.3: Results obtained for the velocity profiles considered in the test sets of the K-fold trial runs {6-10} using the Alternative Machine formulation presented in Chapter 5.

Bibliography

[1] F. P. Incropera, *Fundamentals of Heat and Mass Transfer* (John Wiley & Sons, Inc., USA, 2006).

[2] H. H. Rosenbrock, *An automatic method for finding the greatest or least value of a function,* The Computer Journal **3**, 175 (1960).

[3] A. N. (Stanford), *Deep learning specialization,* (2018).

[4] A. P. Singh, R. Matai, A. Mishra, K. Duraisamy, and P. A. Durbin (American Institute of Aeronautics and Astronautics, 2017) Chap. Data-driven augmentation of turbulence models for adverse pressure gradient flows.

[5] E. Parish and K. Duraisamy, *A paradigm for data-driven predictive modeling using field inversion and machine learning,* Journal of Computational Physics **305**, 758 (2016).

[6] R. Pecnik and A. Patel, *Scaling and modelling of turbulence in variable property channel flows,* Journal of Fluid Mechanics **823**, R1 (2017).

[7] H. Myong and N. Kasagi, *A new approach to the improvement of k-ϵ; turbulence model for wall-bounded shear flows,* JSME international journal. Ser. 2, Fluids engineering, heat transfer, power, combustion, thermophysical properties **33**, 63 (1990).

[8] P. Spalart and S. Allmaras (American Institute of Aeronautics and Astronautics, 1992) Chap. A one-equation turbulence model for aerodynamic flows.

[9] A. Patel, *Universal characterization of wall turbulence for fluids with strong property variations,* Dissertation, Delft University of Technology (2017).

[10] J. Jiménez and S. Hoyas, *Turbulent fluctuations above the buffer layer of wall-bounded flows,* Journal of Fluid Mechanics **611**, 215–236 (2008).

[11] A. Trettel and J. Larsson, *Mean velocity scaling for compressible wall turbulence with heat transfer,* Physics of Fluids **28**, 026102 (2016).

[12] G. J. O. R., A. Patel, R. D. S., and R. Pecnik, *Turbulence modelling for flows with strong variations in thermo-physical properties,* International Journal of Heat and Fluid Flow **73**, 114 (2018).

[13] K. Belyaev, A. Garbaruk, M. Shur, M. Strelets, and P. Spalart, *Experience of direct numerical simulation of turbulence on supercomputers,* Springer International Publishing AG, Second Russian Supercomputing Days **CCIS 687**, 67–77 (2016).

[14] A. Kolmogorov, *The Local Structure of Turbulence in Incompressible Viscous Fluid for Very Large Reynolds Numbers,* in *Dokl. Akad. Nauk SSSR*, Vol. 30 (1941) pp. 301–305.

[15] H. Versteeg and W. Malalasekera, *Introduction to Computational Fluid Dynamics, An : The Finite Volume Method* (Pearson Education, 2007).

[16] S. B. Pope, *Turbulent flows* (Cambridge Univ. Press, Cambridge, 2011).

[17] A. Patel, J. Peeters, B. Boersma, and R. Pecnik, *Semi-local scaling and turbulence modulation in variable property turbulent channel flows,* Physics of Fluids **27**, 095101 (2015), https://aip.scitation.org/doi/pdf/10.1063/1.4929813 .

[18] K. Duraisamy, G. Iaccarino, and H. Xiao, *Turbulence Modeling in the Age of Data,* ArXiv e-prints (2018), arXiv:1804.00183 [physics.flu-dyn] .

[19] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, *Sympy: symbolic computing in python,* PeerJ Computer Science **3**, e103 (2017).

[20] J. Hines, *A logarithmic neural network architecture for unbounded non-linear function approximation,* in *Neural Networks, 1996., IEEE International Conference on*, Vol. 2 (1996) pp. 1245–1250 vol.2.

[21] H. Liu, *Pipeline Engineering (2004)*, CRC Press Revivals (CRC Press, 2017).

[22] J. Anderson, *Computational fluid dynamics: the basics with applications*, Mechanical engineering series (Mcgraw-Hill, New York, NY, 1995) international edition.

[23] H. Schlichting, *Boundary-layer theory* (McGraw-Hill, New York, 1979).

[24] F. M. White, *Fluid Mechanics, Seventh Edition* (McGraw-Hill, 2011).

[25] C. L. Fefferman, *Existence and smoothness of the navier-stokes equations,* in *Princeton; NJ 08544-1000; 2000* (2000).

[26] G. Coleman, J. Kim, and R. Moser, *A numerical study of turbulent supersonic isothermal-wall channel flow,* Journal of Fluid Mechanics **305**, 159–183 (1995).

[27] F. Nieuwstadt, J. Westerweel, and B. Boersma, *Introduction to Theory and Applications of Turbulent Flows* (Springer International Publishing, 2016).

[28] K. Duraisamy, Z. Zhang, and A. Singh (American Institute of Aeronautics and Astronautics, 2015) Chap. New Approaches in Turbulence and Transition Modeling Using Data-driven Techniques.

[29] S. Pirozzoli and M. Bernardini, *Turbulence in supersonic boundary layers at moderate reynolds number,* Journal of Fluid Mechanics **688**, 120–168 (2011).

[30] *ANSYS Fluent 18.1 Documentation*, ANSYS Inc. (2017).

[31] J. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, and D. Mavriplis, *Cfd vision 2030 study: A path to revolutionary computational aerosciences,* (2014).

[32] R. King, P. Hamlington, and W. Dahm (American Institute of Aeronautics and Astronautics, 2015) Chap. Autonomic Subgrid-Scale Closure for Large Eddy Simulations.

[33] G. Borrell, J. Sillero, and J. Jiménez, *A code for direct numerical simulation of turbulent boundary layers at high reynolds numbers in bg/p supercomputers,* Computers & Fluids **80**, 37 (2013), selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[34] G. Eitel-Amor, R. Örlü, and P. Schlatter, *Simulation and validation of a spatially evolving turbulent boundary layer up to* $re_\theta = 8300$, International Journal of Heat and Fluid Flow **47**, 57 (2014).

[35] E. V. Driest, *Turbulent boundary layer in compressible fluids,* Journal of the Aeronautical Sciences **18**, 145 (1951).

[36] B. Launder and D. Spalding, *The numerical computation of turbulent flows,* Computer Methods in Applied Mechanics and Engineering **3**, 269 (1974).

[37] F. Menter (American Institute of Aeronautics and Astronautics, 1993) Chap. Zonal Two Equation k-ω Turbulence Models For Aerodynamic Flows.

[38] D. Wilcox, *Formulation of the k-ω turbulence model revisited,* AIAA Journal **46**, 2823 (2008).

[39] P. Durbin, *Separated flow computations with the k-epsilon-v-squared model,* AIAA Journal **33**, 659 (1995).

[40] R. Cess, *A survey of the literature on heat transfer in turbulent tube flow.*, Tech. Rep. (Westinghouse Research, 1958) report 8-0529-R24.

[41] A. Hussain and W. Reynolds, *Measurements in fully developed turbulent channel flow,* Journal of Fluids Engineering **97**, 568 (1975).

[42] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems,* (2015), software available from tensorflow.org.

[43] O. Bousquet, U. von Luxburg, and G. Rätsch, *Advanced Lectures On Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2-14, 2003, Tubingen, Germany, August 4-16, 2003, Revised Lectures (Lecture Notes in Computer Science)* (SpringerVerlag, 2004).

[44] M. Hutson, *Has artificial intelligence become alchemy?* Science **360**, 478 (2018), http://science.sciencemag.org/content/360/6388/478.full.pdf .

[45] L. Surhone, M. Timpledon, and S. Marseken, *Quasi-Newton Method: Maxima and Minima, Newton's Method in Optimization, Stationary Point, Hessian Matrix, Gradient, Argonne National Laboratory, Positive-Definite Matrix* (Betascript Publishing, 2010).

[46] J. Dennis, Jr. and J. Moré, *Quasi-newton methods, motivation and theory,* SIAM Review **19**, 46 (1977), https://doi.org/10.1137/1019005 .

[47] J. Henry, F. Guyon, and J. Yvon, *Optimal weighting for improving practical identifiability in dps parameter estimation problems,* IFAC Proceedings Volumes **22**, 361 (1989), 5th IFAC Symposium on Control of Distributed Parameter Systems 1989, Perpignan, France, 26-29 June 1989.

[48] K. Giannakoglou, D. Papadimitriou, E. Papoutsis-Kiachagias, and C. Othmer, *Adjoint methods in cfd-based optimization - gradient computation & beyond, ECCOMAS 2012 - European Congress on Computational Methods in Applied Sciences and Engineering, e-Book Full Papers,* , 8523 (2012).

[49] G. Goh, *Why momentum really works,* Distill (2017), 10.23915/distill.00006.

[50] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization,* CoRR **abs/1412.6980** (2014), arXiv:1412.6980 .

[51] R. Battiti, *Accelerated backpropagation learning: Two optimization methods,* Complex Systems **3** (1989).

[52] D. Hubel and T. Wiesel, *Receptive fields of single neurones in the cat's striate cortex,* J Physiol **148**, 574 (1959), 14403679[pmid].

[53] D. Hubel and T. Wiesel, *Receptive fields, binocular interaction and functional architecture in the cat's visual cortex,* J Physiol **160**, 106 (1962), 14449617[pmid].

[54] G. Cybenko, *Approximation by superpositions of a sigmoidal function,* Mathematics of Control, Signals and Systems **2**, 303 (1989).

[55] K. Hornik, *Approximation capabilities of multilayer feedforward networks,* Neural Networks **4**, 251 (1991).

[56] H. Mhaskar, Q. Liao, and T. Poggio, *When and why are deep networks better than shallow ones?* (2017).

[57] Y. Lecun, L. Bottou, G. Orr, and K. Müller, *Efficient backprop,* in *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop* (Springer-Verlag, London, UK, UK, 1998) pp. 9–50.

[58] M. Milano and P. Koumoutsakos, *Neural network modeling for near wall turbulent flow,* J. Comput. Phys. **182**, 1 (2002).

[59] J. Ling, A. Kurzawski, and J. Templeton, *Reynolds averaged turbulence modelling using deep neural networks with embedded invariance,* Journal of Fluid Mechanics **807**, 155–166 (2016).

[60] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition,* Proceedings of the IEEE **86**, 2278 (1998).

[61] I. Aizenberg and A. Gonzalez, *Image recognition using mlmvn and frequency domain features.* in *Rio De Janeiro: Proceedings of the 2018 IEEE International Joint Conference on Neural Networks (IJCNN 2018)* (2018) p. 1550–1557.

[62] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: A simple way to prevent neural networks from overfitting,* J. Mach. Learn. Res. **15**, 1929 (2014).

[63] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going deeper with convolutions,* CoRR **abs/1409.4842** (2014), arXiv:1409.4842 .

[64] J. W. Hill, *Deep learning for emotion recognition in cartoons,* (2017), undergraduate Honors Thesis.

[65] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin, *Accelerating eulerian fluid simulation with convolutional networks,* CoRR **abs/1607.03597** (2016), arXiv:1607.03597 .

[66] Y. Zhang, W. J. Sung, and D. N. Mavris (American Institute of Aeronautics and Astronautics, 2018) Chap. Application of Convolutional Neural Network to Predict Airfoil Lift Coefficient.

[67] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, *Empirical evaluation of gated recurrent neural networks on sequence modeling,* CoRR **abs/1412.3555** (2014), arXiv:1412.3555 .

[68] C. Olah, *Understanding lstm networks,* Github (2015).

[69] T. White, *A neural network architecture for reduced order modeling of pdes,* Department of Mechanical Engineering, Stanford University (2017).

[70] A. Mohan and D. Gaitonde, *A deep learning based approach to reduced order modeling for turbulent flow control using lstm neural networks,* Arxiv, Computational Physics (2018).

[71] J. Karhunen and J. Joutsensalo, *Generalizations of principal component analysis, optimization problems, and neural networks,* Neural Networks **8**, 549 (1995).

[72] S. Pan and K. Duraisamy, *Data-driven Discovery of Closure Models,* ArXiv e-prints (2018), arXiv:1803.09318 [math.DS] .

[73] M. Kaandorp, *Machine Learning for Data-Driven RANS Turbulence Modelling*, Master's thesis, Delft University of Technology, the Netherlands (2018).

[74] L. Ladický, S. Jeong, B. Solenthaler, M. Pollefeys, and M. Gross, *Data-driven fluid simulations using regression forests,* ACM Trans. Graph. **34**, 199:1 (2015).

[75] G. Baker, *Machine learning: Classification,* Computational Data Science (CMPT 353), Simon Fraser University, Canada (2018).

[76] A. P. Singh, S. Medida, and K. Duraisamy, *Machine-learning-augmented predictive modeling of turbulent separated flows over airfoils,* AIAA Journal **55**, 2215 (2017).

[77] D. Yuret, *Autograd: an automatic differentiation package for julia,* `https://github.com/denizyuret/AutoGrad.jl` (2016).

[78] A. Port, *Svgpathtools,* `https://github.com/mathandy/svgpathtools` (2016).

[79] K. Duraisamy, A. Singh, and Z. J. Zhang (American Institute of Aeronautics and Astronautics, 2017) Chap. Augmentation of Turbulence Models Using Field Inversion and Machine Learning.