UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Atmospheric Boundary Layer Stability and its Application to Computational Fluid Dynamics

Author:
Hendrik Johannes Breedt

Supervisor:
Prof KJ Craig

Submitted in partial fulfilment for the degree of

Master of Engineering (Mechanical)

Department of Mechanical and Aeronautical Engineering

January 2018

Abstract

Atmospheric Boundary Layer Stability and its Application to Computational Fluid Dynamics

Author:	Hendrik Johannes Breedt
Supervisor:	Prof KJ Craig
Department:	Mechanical and Aeronautical Engineering
Degree:	Master of Engineering (Mechanical)

In the wind resource and wind turbine suitability industry Computational Fluid Dynamics has gained widespread use to model the airflow at proposed wind farm locations. These models typically focus on the neutrally stratified surface layer and ignore physical process such as buoyancy and the Coriolis force. These physical processes are integral to the accurate description of the atmospheric boundary layer and reductions in uncertainties of turbine suitability and power production calculations can be achieved if these processes are included. The present work focuses on atmospheric flows in which atmospheric stability and the Coriolis force are included.

The study uses Monin-Obukhov Similarity Theory to analyse time series data output from a proposed wind farm location to determine the prevalence and impact of stability at the location. The output provides the necessary site data required for the CFD model as well as stability-dependent wind profiles from measurements. The results show non-neutral stratification to be the dominant condition onsite with impactful windfield changes between stability conditions.

The wind flows considered in this work are classified as high Reynolds number flows and are based on numerical solutions of the Reynolds-Averaged Navier-Stokes equations. A two-equation closure method for turbulence based on the $k - \epsilon$ turbulence model is utilized. Modifications are introduced to standard CFD model equations to account for the impact of atmospheric stability and ground roughness effects. The modifications are introduced by User Defined Functions that describe the profiles, source terms and wall functions required for the ABL CFD model. Two MOST models and two wall-function methods are investigated.

The modifications are successfully validated using the horizontal homogeneity test in which the modifications are proved to be in equilibrium by the model's ability to maintain inlet profiles of velocity and turbulence in an empty domain. The ABL model is applied to the complex terrain of the proposed wind farm location used in the data analysis study. The inputs required for the stability modifications are generated using the available measured data. Mesoscale data are used to describe the inlet boundary conditions. The model is successfully validated by cross prediction of the stability-dependent wind velocity profiles between the two onsite masts.

The advantage of the developed model is the applicability into standard wind industry loading and power production calculations using outputs from typical onsite measurement campaigns. The model is tuning-free and the site-specific modifications are input directly into the developed User Defined Functions. In summary, the results show that the implemented modifications and developed methods are applicable and reproduce the main wind flow characteristics in neutral and non-neutral flows over complex wind farm terrains. In additions, the developed method reduce modelling uncertainties compared against models and measurements that neglect non-neutral stratification.

Keywords: Atmospheric Boundary Layer, Atmospheric Stability, Monin Obukhov Similarity Theory, Computational Fluid Dynamics, Wind Energy, Buoyancy.

Acknowledgements

The following individuals and institutions are thanked for their support and guidance:

- Prof. Ken Craig from the University of Pretoria for his overall supervision and support throughout the project
- Mr. Venkatesh Duraisamy Jothiprakasam, PhD from Siemens Wind Power for his guidance in CFD and stability related topics
- CHPC (Centre for High Performance Computing) in Cape Town for providing the computing capacity required
- Mr. Paul van der Laan from the Technical University of Denmark for assisting in the correct implementation of the M.P. van der Laan et al. [1] developed stability model
- Mr. Alessandro Parente from the Université Libre de Bruxelles for assisting in the correct implementation of the wall function developed by A. Parente et al. [2]

Gratitude is also extended to Siemens Wind Power for their financial support

Contents

List of Figures

List of Tables

List of Abbreviations

ABL	Atmospheric Boundary Layer
AGL	Above Ground Level
AM	Alinot and Masson
ASL	Above Sea Level
CFD	Computational Fluid Dynamics
DES	Detached Eddy Simulation
DNS	Direct Numerical Simulation
DTU	Technical University of Denmark
LES	Large Eddy Simulation
MOST	Monin Obukhov Similarity Theory
MOL	Monin Obukhov Length
RANS	Reynolds-Averaged Navier-Stokes
RSM	Reynolds Stress Model
SFS	Sub Filter Scale
SRTM	Shuttle Radar Topography Mission
SST	Shear Stress Transport
TKE	Turbulent Kinetic Energy
TI	Turbulence Intensity
WRF	Weather Research and Forecasting

Physical Constants

C_p	Constant Pressure Specific Heat Air	= 1006.43 J (kg K)$^{-1}$
C_s	Roughness Constant	= 0.5
E	Wall Function Empirical Constant	= 9.793
g	Gravitational Acceleration	= -9.81 m s^{-2}
L_b	Standard temperature lapse rate	= -0.0065 K m^{-1}
M	Molar Mass Dry Air	= 29 g mol^{-1}
M_v	Molar Mass Water	= 18.015 g mol^{-1}
p_0	Reference Pressure	= 1×10^5 Pa
R	Universal Gas Constant	= 8.314 J (K mol)$^{-1}$
β	Thermal Expansion Coefficient of Air	= 0.0032 K^{-1}
μ	Dynamic Viscosity of Air	= 1.7894 $\times 10^{-5}$ kg (m s)$^{-1}$
Θ_E	Earth Rotational Speed	= 7.292×10^{-5} rad s^{-1}

List of Symbols

Roman Symbols

B	Turbulent Kinetic Energy Production Term	$m^2\ s^{-3}$
C_ϵ	Turbulence Model Constants	–
D	Diffusion Coefficient	–
d	Wall Distance	m
e_{tot}	Total Energy	J
f_c	Coriolis parameter	rad s^{-1}
F_i	Body Force	N
G	Turbulent Kinetic Energy Source/Sink	$m^2\ s^{-3}$
K	Von Karman Constant	–
K_s	Physical Roughness Height	m
k	Turbulent Kinetic Energy	$m^2\ s^{-2}$
L	Monin Obukhov Length	m
l_s	Length Scale	m
O	Turbulent Kinetic Energy Transport Term	$m^2\ s^{-3}$
P	Turbulent Kinetic Energy from Shear Term	$m^2\ s^{-3}$
p	Pressure	Pa
Q_h	Ground Heat Flux	W m^{-2}
q	Heat Flux	W m^{-2}
$Ri_{gradient}$	Gradient Richardson Number	–
Ri_{bulk}	Bulk Richardson Number	–
Ri_{Δ}	Finite Difference Gradient Richardson Number	–
S	Source Term	–
T	Temperature	K
u_*	Frictional Velocity	m s^{-1}
U	Velocity vector	m s^{-1}
u	x Velocity	m s^{-1}
v	y Velocity	m s^{-1}
V_s	Velocity Scale	m s^{-1}
w	z Velocity	m s^{-1}
x_v	Fraction of Water Vapour	–
z	Height AGL	m
z_0	Roughness Length	m
z_{ref}	Reference Height	m
Z	Compressibility Factor	–
z_s	Length Scale	m

Greek Symbols

α	Shear Exponent	–
Γ	Velocity Transfer Function	–
Δ	Finite Difference	–
Δ_B	Wall Function Additive Constant	–
δ_{ij}	Kronecker Delta	–
ϵ	Turbulent Kinetic Dissipation	$m^2\ s^{-3}$
θ	Potential Temperature	K
θ_*	Temperature Length Scale	K
Λ	Latitude	rad
μ	Dynamic Eddy Viscosity	$N\ s\ m^{-2}$
μ_t	Dynamic Turbulent Eddy Viscosity	$N\ s\ m^{-2}$
υ	Kinematic Viscosity	$m^2\ s^{-1}$
υ_t	Kinematic Turbulent Eddy Viscosity	$m^2\ s^{-1}$
Υ	General Variable	–
ρ	Air Density	$kg\ m^{-3}$
ρ_0	Reference Air Density	$kg\ m^{-3}$
σ	Prandtl Number	–
τ	Shear Stress	Pa
τ_w	Wall Shear Stress	Pa
χ	User Defined Scalar	–
ψ	Universal Stability Function	–
Ψ	Integrated Universal Stability Function	–

Sub- and Superscripts

$'$	Fluctuating Part of a Quantity
$\bar{}$	Time Averaged Part of a Quantity
$+$	Non Dimensional
$\tilde{}$	Modified Quantity
0	Surface Value
k	Parameters Associated to Turbulent Kinetic Energy k
ϵ	Parameters Associated to Dissipation Rate ϵ
θ	Parameters Associated to Energy
p	Parameters at Wall Adjacent Cell
t	Turbulent Property
T	Transpose
MO	Monin Obukhov Similarity Theory Formulation

Chapter 1

Introduction

1.1 Motivation

In the wind energy field knowledge about the flow properties of the atmospheric boundary layer (ABL) is important as the wind field is crucial to the design of wind turbines, suitability of turbines to the site and also the energy production of the wind farm. The wind fields over wind farm location vary spatially due to topographical influences caused by complex terrain effects such as valleys, hills, mountains and cliffs. Roughness changes are caused by varying ground cover from vegetation, trees and buildings and also obstacles such as large buildings or forests. These changes in terrain and roughness cause significant variation in the wind speed, wind direction and turbulence intensity.

Site specific information about the wind fields is required and can be obtained by either a measurement campaign where meteorological masts are erected onsite to measure data for a period in excess of one year or by use of synthesized mesoscale data sets. The output of these methods is time series data (typically every 10 minutes or 1 hour) that contain measurements of the flow field at various heights above ground level (AGL). Wind flow modelling is then used to extrapolate this information to areas onsite where no data are available. In this way the on-site data are used to determine the wind field at a turbine location.

Computational Fluid Dynamics (CFD) is widely used for this application and focuses primarily on modelling the neutrally stratified atmospheric surface-layer. The atmospheric surface-layer covers approximately the bottom 10 % of the atmospheric boundary layer (ABL) [3]. Using a typical logarithmic wind profile in this layer guarantees a valid approximation and CFD models can account for the factors that cause wind field variations, however, these neutrally stratified simulations ignore atmospheric stability [3]. In order to reduce uncertainty in wind farm predictions it is necessary to model the whole ABL and its physical mechanisms. This is especially important in complex terrain where strong ABL fluctuations are present spatially.

The ABL can be in three main states namely stable, neutral and unstable. In stable conditions ambient turbulence and vertical fluxes are suppressed by buoyancy forces. This suppression of turbulence leads to delays in wake recovery from wind turbines and can lead to increased energy losses associated with velocity deficits caused by turbine wakes. The lack of vertical motion increases vertical wind shear, defined as the change in velocity with height, and can lead to uneven wind turbine blade loading.

Unstable conditions are characterized by higher ambient turbulence as well as an increased boundary layer height due to the vertical motion experienced. The higher turbulence levels effects the turbine blade fatigue loads. In a diurnal cycle stable conditions are typically seen at night with cooler land temperatures while unstable conditions appear in day times with elevated temperatures. Typically, non-neutral conditions dominate for wind speeds lower than 15 m s^{-1} [1]. This leads to the conclusion that it is important to include atmospheric stability in wind farm simulations and that a change of the standard model equations are necessary. These changes should account for atmospheric stability and the Coriolis force due to the rotation of the earth.

1.2 Aim

The aim of this work is to develop a CFD model for the ABL using a Reynolds-Averaged Navier-Stokes (RANS) model. The model must be able to solve both neutral and non-neutral flow over a typical wind farm terrain with its parameters derived from onsite time series wind data. The model results must also be able to be used in standard wind industry turbine loading software.

1.3 Objectives

The method can be briefly described as follows: Monin-Obukhov Similarity Theory (MOST) is applied to measured time series data from meteorological masts at a proposed wind farm location to understand the effects of atmospheric stability onsite and also to obtain the characteristic values. The data analysis is conducted using a developed Matlab 16a code [4]. Modifications based on MOST are made to the standard RANS CFD model equations to account for atmospheric stability. These modifications are tested to be in equilibrium using the horizontal homogeneity test in an empty domain. Two $k-\epsilon$ RANS turbulence model modifications for MOST are investigated, two wall function methods and two methods for the turbulent production due to buoyancy. The model is applied to the wind farm location using boundary conditions obtained by applying MOST to a mesoscale data set and source terms from the meteorological mast. The model is validated by cross predicting the velocity profiles obtained from the two meteorological masts. Ansys Fluent 18.1 is used for all CFD simulations [5].

The objectives of this work are as follows:

- Development of an empirical data analysis method for computing the stability conditions using wind speed, wind direction, temperature, pressure and relative humidity from onsite measured and mesoscale data. The method determines the stability conditions based on the Monin-Obukhov Length (MOL) and includes the calculation of density, frictional velocity, heat flux and temperature scale in each 30° sector.

- Calculation of the corresponding velocity, turbulence and temperature profiles using Monin-Obukhov Similarity Theory.

- Investigation of the developed method to analyse a proposed wind farm location using time series data.
- Development of a CFD model that demonstrates horizontal homogeneity in an empty domain and accounts for neutral and non-neutral stratification.
- Investigation of the turbulence model and buoyancy production term modifications for MOST along with modifications to the standard log law wall function.
- Application of user defined functions to appropriately modify the RANS model equations.
- Extension of the developed CFD model to investigate the same proposed wind farm location by simulating neutral and non-neutral wind flow over complex wind farm terrain for which the inlet and source descriptions are obtained from the site data analysis.
- Validation of the CFD model by cross prediction of the onsite measured wind speed profiles.
- Comparison of the Alinot-and-Masson and the M.P. van der Laan et al RANS turbulence model modifications with respect to model results and validity.

1.4 Overview

The structure of this document is as follows: Chapter 2 is devoted to the necessary literature review. Chapter 2.1 presents a review of the ABL physics and theory along with the main ABL governing equations. In Chapter 2.2 the the current state of CFD models are presented along with RANS turbulence models and wall functions. Chapter 2.3 is devoted to describing Monin-Obukhov Similarity Theory. The theory and physics of MOST are presented along with its adaptation into the $k-\epsilon$ RANS turbulence model through constants and profiles. The incompatibility of MOST with the standard $k-\epsilon$ turbulence model is presented along with two solution methods. The incompatibility of the standard wall functions with ABL simulations are presented and two ABL specific wall function models are introduced.

Chapter 3 presents the selected wind farm location and the main features present at the location. The wind data from the site are analysed for stability effects and profiles are generated for wind speed, temperature and turbulence. Chapter 4 provides the description of the CFD model including the physical models along with the boundary conditions and the modified equations. The numerical implementation of the modifications is presented and tested for validity through the models ability to maintain the inlet profiles in an empty domain. Chapter 5 presents the results of the model's application to the complex terrain of the proposed wind farm location. Finally the model is validated using onsite measured data. Chapter 6 provides the conclusions to the dissertation and also highlights possible future work to be performed.

Chapter 2

Literature Review

2.1 The Atmospheric Boundary Layer

The Atmospheric boundary layer (ABL) is defined in literature as the lower portion of the atmosphere that is influenced by the earth's surface and typically occupies the lower 10-20 % of the troposphere with a height range of less than 100 m up to 3000 m or more [6]. The height is typically defined as the region where turbulence drops to values in the range of 5 % of the surface value. Above the ABL temperature starts to increase causing a temperature inversion that separates the ABL from the rest of the troposphere [3]. The ABL is shown graphically in Figure 2.1. The varying conditions of the earth's surface influence the ABL wind field by means of turbulence communicating the drag from the ground surface throughout the ABL.

The ABL consists of two regions with the atmospheric surface layer occupying the lower 10 % of the ABL. The surface layer is categorized by steep vertical gradients of wind speed and temperature with their structures governed mainly by surface friction and the vertical temperature gradient. In this region, the heat and momentum fluxes are approximately constant with negligible impact from the Coriolis force. Above the surface layer, the gradients of wind speed and temperature decrease and the Coriolis effect influence becomes more apparent causing a turning of the wind with height [3].

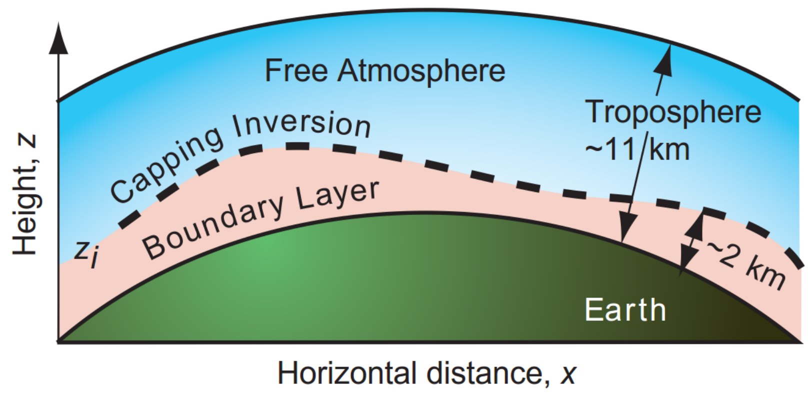


Figure 2.1: The ABL as shown in a vertical cross section of the troposphere [6].

The ABL undergoes continuous change during the day (24 Hour diurnal cycle) during typical fair weather conditions. The changes are induced by alternating heating and cooling of the earth's surface caused by incoming solar radiation that heats the ground during daytime and cooling in night time caused by emitted long-wave radiation [6]. The windfield changes rapidly in response to these changing conditions and Figure 2.2 shows this graphically in a vertical cross section [6]. The three distinct states of the ABL can be defined (Neutral, Stable and Unstable). Neutral conditions occur when a constant potential temperature with height is present. Stable occurs when the ground surface is cooler than the air, typically in night time, and unstable occurs when the ground surface is warmer than the air, typically during day time.

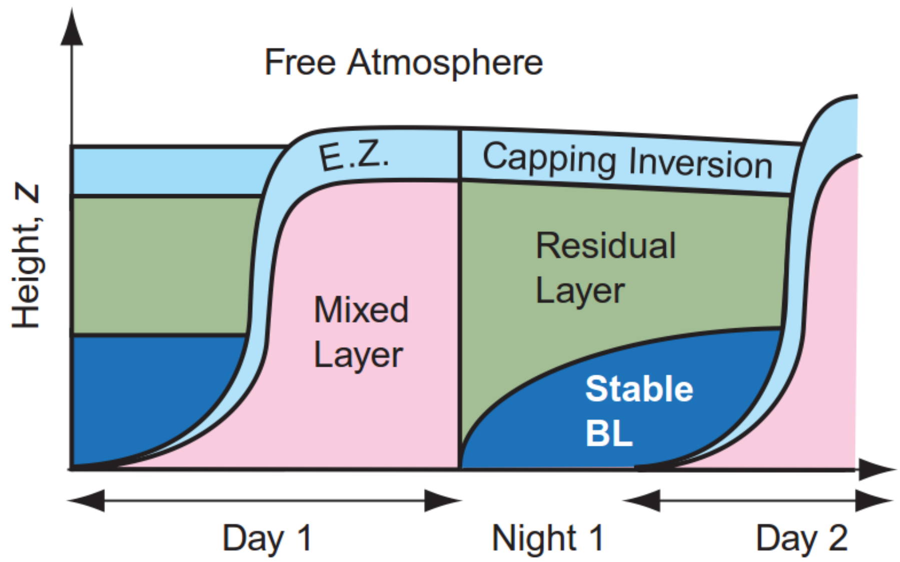


Figure 2.2: The ABL evolution of a typical summer day [6].

A typical diurnal cycle is described below and will repeat daily in fair weather conditions [6]. During unstable conditions air heated from the ground surface rises due to buoyancy forces, this effect enhances turbulence production causing a mixing layer to form. This layer is also called the convective boundary layer. The ABL continues to grow throughout the day caused by strengthening of the buoyancy forces and turbulent mixing. This process leads to entrainment in which rising thermals overshoot a small distance into the inversion layer. Strong convective turbulence causes air parcels from above the inversion layer to be mixed into the mixing layer, this is called the entrainment zone. The ABL continues to grow until typically late afternoon when the maximum height of the ABL is reached and the ABL is then in a neutral condition. After sunset the decrease in ground surface temperature causes a small stable layer to form close to the ground (Nocturnal Boundary Layer), the near neutral layer remains on top of the stable layer and is called the residual layer and retains the capping inversion. In the nocturnal layer cold air sinks to the ground due to buoyancy forces and also causes the turbulence to be suppressed. The nocturnal layer continues to grow during night time. Upon sunrise unstable conditions will start to occur close to ground level which will erode the stable conditions.

The windfields close to ground react quickly to any changes on the ground surface that occur diurnally. Typical ABL fluctuations appear with a time scale of around one hour or less [6]. Generally the wind profiles are assumed to be accurately approximated with a logarithmic profile that reduces to zero at ground level. However, when stability is taken into account, the profile can deviate significantly from the standard neutral condition logarithmic profile [1] [6]. Stable conditions are characterized by low turbulence levels due to drag from the surface layer not being effectively communicated from the ground surface level. This leads to less mixing effects and an increase in shear (higher increase for wind speed as a function of height above ground). Unstable conditions are characterized by high turbulence levels that mix momentum towards to the ground which leads to a high wind speed increase with height close to the ground, however, further away the well-mixed flow results in much smaller vertical gradients for wind speed. This means that the wind profile for unstable conditions have a much lower wind shear value. Figure 2.3 shows the wind speed profiles for the three conditions as described above [6]. Non-neutral stratification can also cause lifting/blocking effects when the windfields encounter a terrain feature like a hill [7] [1]. In neutral conditions the wind profiles would go smoothly over the hill, in stable conditions they are more likely to flow around the hill rather than over. This is due to the buoyancy effects in stable condition that counteract lifting. In unstable conditions the profile rises over the hill and is more prone to continue to rise after the hill due the buoyancy effect caused by the displaced profiles which is warmer than the surrounding air [7].

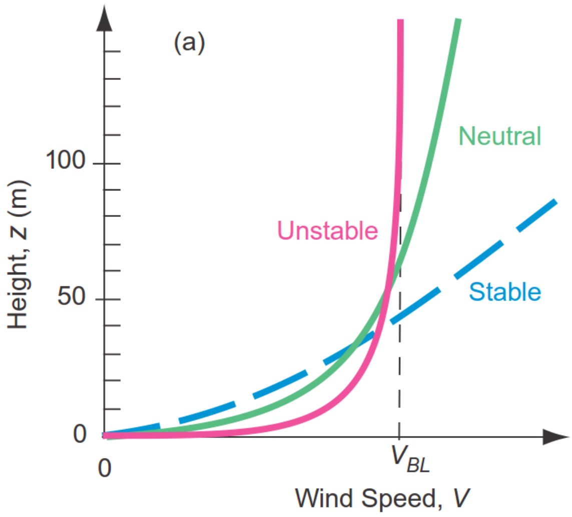


Figure 2.3: Typical wind speed profiles for the various stability conditions [6].

The atmosphere consists mainly of oxygen and nitrogen with trace amounts of other gases including water vapour, hydrogen, carbon dioxide and helium. However, the atmosphere can be regarded as homogeneous gas of uniform composition [6]. Consider a parcel of air lifted upwards in the atmosphere. The pressure of the parcel will decrease in response to the atmospheric pressure field under the influence of gravity if there is no heat transfer to the parcel from conduction or radiation (Adiabatic). Due to the rapid nature of the vertical turbulence motions of the ABL, the adiabatic assumption is considered to be accurate [6].

Determining the wind speed profiles of the ABL is not easily assessed due numerous parameters that influence the ABL. A valid assumption for the ABL can be made by modelling the ABL as a Newtonian fluid with the Navier-Stokes equations. The most common relation is based on the turbulence kinetic energy (TKE) [6]. TKE is given as the sum of the average square fluctuations of the wind speed.

$$\text{TKE} = 0.5 \times \left(\overline{u''^2} + \overline{v'^2} + \overline{w'^2}\right) \tag{2.1}$$

There are several contributors such as buoyancy and dissipation that influence the definition of TKE, dividing these terms by $u_*^3/(kz)$ to make them non dimensional, the relation in Equation 2.2 is obtained. The left hand side of the equation represents the mechanical production/loss due to shear. K is the von Karman constant and z is the vertical height above ground [6]. All of the other contributing terms are included in the right-hand side that is written as a function of a non-dimensional universal function ψ_m. This function is proportional to the frictional velocity u_* as given in Equation 2.3. The wind speed fluctuations parallel and perpendicular to the average of the main wind speed u are given by u' and w', respectively.

$$\frac{Kz}{u_*}\frac{\partial \overline{u}}{\partial z} = \psi_m \tag{2.2}$$

$$u_*^2 = \mid u'w' \mid \tag{2.3}$$

If the mechanical wind shear term is in equilibrium with the other contributing factors ψ_m equals to 1 [6]. This occurs during neutral conditions and allows Equation 2.2 to be rewritten into the format in Equation 2.4. Taking the integral of this relation between the ground roughness length z_0 and a reference height z_{ref} as is done in Equation 2.5 the logarithmic wind speed profile above ground is obtained with Equation 2.6. The roughness length is defined as the height above ground up to which the wind speed is zero, Appendix A shows a table of typically assumed roughness lengths [8].

$$\partial u = \frac{u_*}{Kz}\partial z \tag{2.4}$$

$$\int_0^{u_{ref}} du = \int_{z_0}^{z_{ref}} \frac{u_*}{Kz} dz \tag{2.5}$$

$$u\left(z\right) = \frac{u_*}{K}\ln\left(\frac{z}{z_0}\right) \tag{2.6}$$

Equations 2.4 to 2.6 are only valid in neutral conditions, for general conditions ψ_m does not equal 1 and the derivation of the wind speed profile is more complex and a new specific universal function taking into account the other terms from buoyancy and dissipation in the TKE relation is needed.

The problem is overcome by using the Monin-Obukhov Similarity Theory (MOST) [9]. The theory assumes that by using a nondimensionalization scheme (Buckingham's π-theorem) the parameters g/T_0 (T_0 is the ground surface temperature and g the gravitational acceleration) along with u_* and $Q_H/(C_p\rho)$ (where Q_H is the ground kinematic heat flux, C_p the specific heat and ρ the air density) successfully describe the atmospheric turbulence. Only one parameter is then needed to describe the process, the Monin-Obukhov Length (MOL) as given in Equation 2.7 [10] with θ_* indicating the temperature scale and L the symbol used for Monin-Obukhov Length.

$$L = \frac{u_*^2 T_0}{K g \theta_*} \tag{2.7}$$

From Buckingham's π-theorem the universal functions for wind speed and temperature should only be a function of the dimensionless parameter z/L. This allows the following equations to be obtained.

$$\frac{Kz}{u_*}\frac{\partial u}{\partial z} = \psi_m\left(\frac{z}{L}\right) \tag{2.8}$$

$$\frac{Kz}{T_*}\frac{\partial T}{\partial z} = \psi_t\left(\frac{z}{L}\right) \tag{2.9}$$

The most used universally accepted functions for $\psi_m$and ψ_t and the relations used throughout this study, are the Dyer relations [11]:

Stable:

$$\psi_m\left(\frac{z}{L}\right) = \psi_t\left(\frac{z}{L}\right) = 1 + 5\frac{z}{L} \tag{2.10}$$

Neutral:

$$\psi_m\left(\frac{z}{L}\right) = \psi_t\left(\frac{z}{L}\right) = 0 \tag{2.11}$$

Unstable:

$$\psi_m\left(\frac{z}{L}\right) = \left(1 - 16\frac{z}{L}\right)^{-1/4} \tag{2.12}$$

$$\psi_t\left(\frac{z}{L}\right) = \left(1 - 16\frac{z}{L}\right)^{-1/2} \tag{2.13}$$

A useful conversion is made from temperature to potential temperature θ using Equation 2.14 [6]. R is the universal gas constant. Potential temperature is defined as the temperature a parcel of air will attain if it were brought adiabatically to the standard pressure p_0 of the earth's surface [6]. Potential temperature is used for the intrinsic property of being conserved with height and undergoes no change during vertical movements of the air parcel in the adiabatic atmosphere. This removes the typical rate of change of temperature with height known as dry adiabatic lapse rate (ALR). At ground level the temperature and potential temperature are equal.

$$\theta = T\left(\frac{p_0}{p}\right)^{R/C_p} \tag{2.14}$$

Using potential temperature, the stability conditions are easily recognizable using Equations 2.15 to 2.17 [3].

$$\frac{\partial\theta}{\partial z} > 0 \quad \text{Unstable} \tag{2.15}$$
$$\frac{\partial\theta}{\partial z} = 0 \quad \text{Neutral} \tag{2.16}$$
$$\frac{\partial\theta}{\partial z} < 0 \quad \text{Stable} \tag{2.17}$$

After converting and integrating Equations 2.10 to 2.13 as before, the expressions for wind speed and potential temperature are obtained. Ψ_m and Ψ_t are the universal functions for wind speed and temperature which are the integrals of ψ_m and ψ_t respectively.

$$u(z) = \frac{u_*}{K}\left[\ln\left(\frac{z}{z_0}\right) - \Psi_m\left(\frac{z}{L}\right)\right] \tag{2.18}$$
$$\theta(z) = \theta(z_0) + \frac{\theta_*}{K}\left[\ln\left(\frac{z}{z_0}\right) - \Psi_t\left(\frac{z}{L}\right)\right] \tag{2.19}$$

With the following obtained by integrating Equations 2.10-2.13:

Stable:

$$\Psi_m\left(\frac{z}{L}\right) = \Psi_t\left(\frac{z}{L}\right) = \left(\frac{-5z}{L}\right) \tag{2.20}$$

Neutral:

$$\Psi_m\left(\frac{z}{L}\right) = \Psi_t\left(\frac{z}{L}\right) = 0 \tag{2.21}$$

Unstable:

$$\Psi_m\left(\frac{z}{L}\right) = 2\ln\left[\frac{1+x}{2}\right] + \ln\left[\frac{1+x^2}{2}\right] - 2\arctan(x) + \frac{\pi}{2} \tag{2.22}$$
$$\Psi_t\left(\frac{z}{L}\right) = 2\ln\left[\frac{1+x^2}{2}\right] \tag{2.23}$$
$$\tag{2.24}$$

Where

$$x = \left[1 - \left(16\frac{z}{L}\right)\right]^{1/4} \tag{2.25}$$

This indicates the variation of wind speed and temperature profiles and highlights the importance of using stability-based profiles instead of the standard logarithmic profiles.

The Monin-Obukhov Length is used as the parameter to define atmospheric stability. The stability is defined in five classes as reported in Table 2.1 [12]. Up to seven classes exist including slightly unstable and slightly stable, for this work these cases are absorbed into the unstable and stable regions respectively. Using classes to bin the wind field allows the correct profiles to be obtained for each onsite stability condition.

Table 2.1: Monin-Obukhov Length classification for atmospheric stability [12]

Condition	Monin-Obukhov Length [m]
Extremely Unstable	$-100 \leq L < 0$
Unstable	$-500 \leq L < -100$
Neutral	$\mid L \mid > 500$
Stable	$50 \leq L < 500$
Extremely Stable	$0 \leq L < 50$

The calculation of MOL is paramount in the definitions of the stability classes and profiles. Determining MOL, however, is not straightforward and various techniques are presented in literature. The following methods are presented hereafter: Gradient Richardson $Ri_{gradient}$, Bulk Richardson Ri_{bulk} and a profile method using different levels of wind speed and temperature.

2.1.1 Gradient Richardson number

The Gradient Richardson method is based on wind speed and temperature gradients [11]. It is shown in Equation 2.26. Negative values of Gradient Richardson indicate unstable conditions and positive values stable.

$$Ri_{gradient} = \frac{g}{\theta}\frac{\frac{\partial \theta}{\partial z}}{\left(\frac{\partial u}{\partial z}\right)^2} \tag{2.26}$$

A typical implementation involves taking a finite difference from two relevant heights with $\Delta z = z_1 - z_2$.

$$Ri_{\Delta} = \frac{g}{\theta}\frac{\dfrac{\Delta \theta}{\Delta z}}{\left(\dfrac{\Delta u}{\Delta z}\right)^2} \tag{2.27}$$

Using the Gradient Richardson number and Equation 2.29 the MOL can be calculated using Equation 2.29 using the following length scale.

$$z_s = \frac{\Delta z}{\ln \dfrac{z_1}{z_2}} \tag{2.28}$$

The method is only valid for $Ri_{\Delta} < 0.2$ [13].

$$L = \begin{cases} \frac{z'}{Ri_{\Delta}} & , Ri_{\Delta} \leq 0 \\ \rightarrow \infty & , Ri_{\Delta} = 0 \\ \frac{z'(1-5Ri_{\Delta})}{Ri_{\Delta}} & , 0 < Ri_{\Delta} \leq 0.2 \end{cases} \tag{2.29}$$

2.1.2 Bulk Richardson number

The Bulk Richardson number is based on the wind speed at only the upper level [13]. Due to only using wind speed at one level (z_2), using Ri_{bulk} to determine L leads to inaccuracies in the method when extrapolating the profiles [13].

$$Ri_{bulk} = \frac{\frac{z_2 g \Delta\theta}{\theta_2}}{u_2^2} \tag{2.30}$$

2.1.3 Profile method

Using the profiles obtained in Equations 2.18-2.19 along with the relations from Equations 2.20-2.25 and rewriting them using two different levels, one can explicitly solve for u_* and θ_* using the equations below:

$$u_* = \frac{k\left(u_2 - u_1\right)}{\ln\left(\frac{z_2}{z_1}\right) - \Psi_m\left(\frac{z_2}{L}\right) + \Psi_m\left(\frac{z_1}{L}\right)} \tag{2.31}$$

$$\theta_* = \frac{k\left(\theta_2 - \theta_1\right)}{\ln\left(\frac{z_2}{z_1}\right) - \Psi_t\left(\frac{z_2}{L}\right) + \Psi_t\left(\frac{z_1}{L}\right)} \tag{2.32}$$

These values can then be used in Equation 2.7 to obtain L. The method is presented here using 2 heights to explicitly solve for u_* and θ_*. If more height data points are available a non-linear least squares fitting using Equations 2.18-2.19 can be used to obtain the profiles that best fit the measurements by solving for u_* and θ_*.

2.1.4 Heat flux

The ground heat flux Q_H can be calculated using Equation 2.33 [6]. A positive heat flux is associated with warm air moving up, cold air moving down and is experienced during unstable conditions. During stable conditions the warm air starts to move downward and negative heat flux is experienced [6].

$$Q_H = -\rho C_p u_* \theta_* \tag{2.33}$$

2.1.5 Air density

Atmospheric gases can be considered to exactly obey the ideal gas law. Taking into account moist air the density of air, can be determined using:

$$\rho = \frac{pM}{ZRT}\left[1 - x_v\left(1 - \frac{M_v}{M}\right)\right] \tag{2.34}$$

with p pressure, T temperature, R the universal gas constant, Z the compressibility factor, M and M_v the molar mass for dry air and water respectively. x_v is the mole fraction of water vapour derived from the relative humidity [14].

2.2 Governing Equations

In this section the governing equations that describe the dynamics and physics of the turbulent ABL are presented. The focus is on micro-scale phenomena in the ABL and processes on greater scales are omitted along with atmospheric processes such as radiation, heat transfer between soil and air, clouds and precipitation. Firstly the basic set of governing equations for neutral incompressible atmospheric flows are presented along with the standard $k-\epsilon$ turbulence model. In the following section the necessary adaptations of the governing equations are presented in order to describe the non-neutral ABL including the Coriolis force.

There are three main expressions governing fluid mechanics: the conservation of mass, momentum and energy. The ABL can be treated as an incompressible Newtonian fluid obeying the perfect gas law [6] and the governing equations reduce to the incompressible Navier-Stokes equations for mass, momentum and energy presented in Equations 2.35 to 2.37. $x_i(x_1 = x, x_2 = y, x_3 = z)$ are the longitudinal, lateral and vertical directions and u_i is the velocity component along x_i labelled (u, v, w) respectively. μ is the dynamic viscosity, ρ the fluid density and F_i the body forces. e_{tot} represents the total energy, q_j the heat flux and τ the viscous stresses.

Continuity:

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{2.35}$$

Momentum:

$$\rho\left(\frac{\partial u_i}{\partial t} + u_k\frac{\partial u_i}{\partial x_k}\right) = -\frac{\partial p}{\partial x_i} + \rho F_i + \frac{\partial}{\partial x_k}\left(\mu\frac{\partial u_i}{\partial x_k}\right) \tag{2.36}$$

Energy:

$$\rho\frac{\partial e_{\text{tot}}}{\partial t} = -\frac{\partial}{\partial x_j}\left[\rho u_j e_{\text{tot}} + u_j p + q_j - u_i\tau_{ij}\right] \tag{2.37}$$

The flow in this present study is treated as incompressible due to the fact that density changes are small and appear at low speeds, however, this does not imply constant density and by definition pressure changes due to density changes are negligible [3]. More information is presented in Section 2.3.1.

In most cases analytical solutions to the Navier Stokes equations equations do not exist and numerical solutions remain the only possible way. Solutions for the buoyancy forces, thermal effects, Coriolis forces and turbulence in the ABL are needed for full description of the ABL.

Full resolution of turbulence is only possible using Direct-Numerical-Simulations (DNS). This method is affordable only for very low Reynolds numbers. The typical Reynolds numbers for ABL flows are in the range of 10^5 to 10^{10} [3]. These kind of flows are affordable to Reynolds-Averaged Navier-Stokes equations (RANS) and in smaller domains with high fidelity models such as Large-Eddy-Simulation (LES) and hybrid methodologies that use a combination of RANS and LES such as the Detached-Eddy-Simulation (DES).

The most common method for studying turbulence uses Reynolds decomposition. This separates the fluctuation variables in turbulent flow into a mean term indicated with a overbar and a fluctuating term indicated by an apostrophe. In the case of a general variable, $\Upsilon = \overline{\Upsilon} + \Upsilon'$, respectively [15]. RANS applies Reynolds decomposition to the Navier-Stokes equations in order to time average them. RANS contains further unknowns called Reynolds stresses and these stresses need to be modelled in order to allow for the closure of turbulence. There are various RANS-based turbulence models and they are typically based upon the additional number of differential transport equations needed to close the original set of partial differential equations [15]. Some of the main models are:

- Zero equation algebraic model: mixing length
- One equation model: Spalart Allmaras
- Two equation model: $k - \epsilon$ (standard, RNG, realizable), $k - \omega$ (standard, SST)
- Seven equation model: Reynolds Stress Model (RSM)

A second approach to turbulence modelling is LES, which is based on the space-filtered Navier Stokes equations. This method resolves the large eddies whose dimensions are larger than that of the filter width. The smaller eddies are modelled using Sub-Filter-Scale (SFS) turbulence models [15]. Due to fact that the large eddies are resolved, the LES methods needs to be three-dimensional and transient. In general using LES over RANS alleviates the issue of needing to tune model constants to the given problem. Large eddies are strongly anisotropic and are thus heavily dependant on the flow and boundary conditions. The smaller eddies lose information about these conditions and are more homogeneously spread and isotropic. This means that if the correct filter is applied, the small eddies can be accurately modelled for all turbulence conditions. LES requires fine grids to discretize the near-wall region in wall-bounded flows which increases the expense of running the model.

DES uses a RANS approach in the near-wall region and an LES model for the zones distant from the walls. It originated for external aerodynamic simulations. DES modifies the usual RANS model to act in its standard way close to the wall and in a modified method far from the wall using an SFS model. Eddy solving methods are computationally expensive, however, with modern computational capabilities it is possible to use these methods for ABL simulations typically using Wall Modelled LES or DES [16]. However these results are transient and barriers remain for the use of these results in wind turbine loading calculations, as such RANS remains the most widely used approach for ABL modelling and it is the focus method of this study.

2.2.1 The RANS equations

Turbulent flows can be treated as statistically steady if the statistics of the flow remain constant over a certain time period. Time averaging the Navier-Stokes using Reynolds decomposition over a time period long enough to reach this state results in the RANS Equations 2.38 and 2.39 for continuity and momentum. The high Reynolds number ABL flows in this study are based on solutions to these equations.

$$\frac{\partial \rho U_i}{\partial x_i} = 0 \tag{2.38}$$

$$\frac{\partial \rho U_i}{\partial t} + \frac{\partial \rho U_i U_j}{\partial x_j} - \frac{\partial}{\partial x_j}\left[(\mu + \mu_t)\left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i}\right)\right] + \frac{\partial \hat{p}}{\partial x_i} = S_M \tag{2.39}$$

When time averaging the Navier-Stokes equations new unknowns are introduced for turbulent eddy viscosity μ_t and Reynolds stresses $\overline{\rho u_i' u_j'}$. The Boussinesq hypothesis is used to relate the Reynolds stresses to the mean velocity gradients using Equation 2.40 [15], with δ_{ij} the Kronecker symbol.

$$\overline{\rho u_i' u_j'} = \mu_t \left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i}\right) - \frac{2}{3}\left(\rho K + \mu \frac{\partial U_i}{\partial x_i}\right)\delta_{ij} \tag{2.40}$$

The Reynolds stresses originate from time averaging the convective term of the Navier-Stokes equations [15]. They are typically grouped in the diffusive term of the RANS momentum and are responsible for turbulent diffusion of momentum which in highly turbulent flows is several orders of magnitude greater than molecular diffusion due to viscosity. The hydrostatic pressure $\hat{p} = \rho_0 g_i$ is absorbed into the pressure formulation [3].

In order to close the equations the turbulent eddy-viscosity needs to be modelled. Various models for μ_t exist and are listed in Section 2.2 [15]. Zero-equation models assume a constant turbulent eddy-viscosity or calculates a direct solution for turbulent eddy-viscosity using the flow variables. One-equation models use a single transport equation for the turbulent eddy-viscosity. The most common is to use two transport equations, one for the length scale and one for the velocity scale of the turbulence. A way to close the RANS without the Boussinesq hypothesis is to apply a transport equation for each of the seven Reynolds stresses, however, this leads to a high computational cost and it is more common to perform a simulation using LES or DES instead.

In the following section the standard two-equation $k - \epsilon$ turbulence model is described as it is the primary model uses in this study. The model is presented in the formulation it is included in Fluent 18.1. [5] [17].

2.2.2 $k - \epsilon$ model

In the $k - \epsilon$ turbulence model the turbulent eddy viscosity is defined using a velocity scale V_s and a length scale l_s with use of Equations 2.41 and 2.42 [15].

$$\mu_t = \rho C_\mu V_s l_s \tag{2.41}$$

$$V_s = k^{1/2} \quad l_s = \frac{k^{3/2}}{\epsilon} \tag{2.42}$$

The standard model is based on two transport equations for turbulent kinetic energy (k) and its dissipation rate (ϵ) as shown respectively in Equations 2.43 and 2.44 [17].

$$\frac{\partial \rho k}{\partial t} + \frac{\partial \rho k u_i}{\partial x_i} = \frac{\partial}{\partial x_j}\left[\left(\mu + \frac{\mu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] + G_k + G_b - \rho\epsilon - Y_m + S_k \tag{2.43}$$

$$\frac{\partial \rho \epsilon}{\partial t} + \frac{\partial \rho \epsilon u_i}{\partial x_i} = \frac{\partial}{\partial x_j}\left[\left(\mu + \frac{\mu_t}{\sigma_\epsilon}\right)\frac{\partial \epsilon}{\partial x_j}\right] + C_{\epsilon 1}\frac{\epsilon}{k}(G_k + C_{\epsilon 3}G_b) - C_{\epsilon 2}\rho\frac{\epsilon^2}{k} + S_\epsilon \tag{2.44}$$

σ_k and σ_ϵ are the turbulent Prandtl numbers for k and ϵ respectively with S_k and S_ϵ user defined source terms. $C_{\epsilon 1}, C_{\epsilon 2}$ and $C_{\epsilon 3}$ are model constants. The model constants are not universal, although certain values are typically used as they produce the correct levels of turbulence in common industrial flows. The default values adopted in the $k - \epsilon$ model are shown in Table 2.2 [15] [17].

Table 2.2: Default $k - \epsilon$ model constants

C_μ	$C_{1\epsilon}$	$C_{2\epsilon}$	σ_k	σ_ϵ
0.09	1.44	1.92	1	1.3

G_k represents turbulence production due to the mean velocity gradients and is determined using Equation 2.45.

$$G_k = -\rho\overline{u_i' u_j'}\frac{\partial \rho u_j}{\partial x_i} \tag{2.45}$$

G_b represents turbulence production due to buoyancy and is determined with Equation 2.46 with β the coefficient of thermal expansion given by Equation 2.47. G_b is included in Fluent only if a non-zero gravity field and temperature gradient are present. σ_θ is the turbulent Prandtl number for energy and has a Fluent default value of 0.85 [17].

$$G_b = \beta g_i \frac{\mu_t}{\sigma_\theta}\frac{\partial T}{\partial x_i} \tag{2.46}$$

$$\beta = -\frac{1}{\rho}\left(\frac{\partial \rho}{\partial T}\right) \tag{2.47}$$

When G_b is positive, turbulence is augmented, while a negative G_b suppresses turbulence. These conditions are in alignment with the unstable and stable conditions respectively. The buoyancy effects on k is well understood, not so however, for ϵ [17]. By default G_b is set to zero in the ϵ transport equation, however, it can be included by advanced settings in Fluent [17]. For this study it is not activated in Fluent but reintroduced via the source term. The degree to which ϵ is influenced by G_b is determined using the $C_{\epsilon 3}$ constant. Fluent does not allow the specification of $C_{\epsilon 3}$ and it is instead calculated using Equation 2.48 where v is the velocity component parallel to the gravitational vector and u perpendicular to the gravitational vector [17].

$$C_{\epsilon 3} = \tanh\left(\left|\frac{v}{u}\right|\right) \tag{2.48}$$

2.2.3 Wall functions

The presence of walls has a significant impact on turbulent flows. The velocity field is affected by the wall no-slip condition. The turbulence is affected very close to the wall by viscous damping which reduces the tangential velocity fluctuations and the normal fluctuations are reduced by kinematic blocking [17]. Due the large gradients in mean velocity towards the outer part of the near-wall region there is a rapid augmentation in turbulence. In the near-wall region the solution variables have large gradients and the transport equations occur more vigorously than in other regions [17]. Walls are typically the main source of turbulence and mean vorticity. Solution fidelity and accurate predictions of wall-bounded turbulent flows therefore require accurate representation of the flow in the near-wall region [17].

There are two main approaches for near-wall modelling. The first, called the wall function approach, relies on the use of semi-empirical formulas that bridge the region between the wall and the fully turbulent region [17]. This method does not require the modification of the turbulence models in the near-wall region. The second approach relies on the modification of the turbulence models near the wall to allow resolution using a fine a mesh all the way down to the wall [17]. This method is called the near-wall modelling approach.

To obtain high quality numerical results using near-wall modelling the resolution of the wall boundary needs to be sufficiently fine [17]. Due to large size of ABL simulations the required resolution for this method would not be feasible and wall-function methods are predominately used in ABL simulations.

The standard wall function used in Fluent is based on the work of Launder and Spalding [18] [17]. The wall function is applied in the wall adjacent cells only. The wall function modified for roughness has the following form [19] [20].

$$\frac{U_p u_* \rho}{\tau_w} = \frac{1}{K} \ln\left(E \frac{\rho u_* z_p}{\mu}\right) - \Delta_B \tag{2.49}$$

with

$u_* =$ Frictional velocity $= (C_\mu^{1/4} k^{1/2})$
$E =$ Empirical constant $(= 9.793)$
$U_p =$ Fluid mean velocity at the wall adjacent cell centroid
$k_p =$ Fluid turbulent kinetic energy at the wall adjacent cell centroid
$z_p =$ Distance from wall to cell centroid of the wall adjacent cell
$\tau_w =$ Wall shear stress

Δ_B is the additive constant which quantifies the shift of the standard log-law intercept due to roughness effects and depends on the type and size of roughness [19]. The constant has been correlated with the non-dimensional roughness height K_s^+ based on the physical roughness height K_s [19] [21].

$$K_s^+ = \frac{\rho K_s u_*}{\mu} \tag{2.50}$$

There are three distinctive forms for K_s^+ namely hydrodynamically smooth $(K_s^+ \leq 2.25)$, transitional $(2.25 \geq K_s^+ \leq 90)$ and fully rough $(K_s^+ \geq 90)$ [19]. The condition for ABL flow is generally rough and the non-dimensional roughness height then becomes Equation 2.51 where C_s is the roughness constant which has a Fluent default value of 0.5 [21] [19].

$$\Delta_B = \frac{1}{K} \ln\bigl(1 + C_s K_s^+\bigr) \tag{2.51}$$

The k transport equation is solved in the whole domain including the wall-adjacent cell with the following boundary condition imposed at the wall where n is the local coordinate normal to the wall [17]

$$\frac{\partial k}{\partial n} = 0 \tag{2.52}$$

The production of turbulent kinetic energy, G_k and the dissipation rate are computed in the wall adjacent cells under the local equilibrium hypothesis which assumes the production of k and dissipation are equal [17]. The production of k at the wall adjacent cell then becomes

$$G_k = \frac{\tau_w^2}{K \rho C_\mu^{1/4} k_p^{1/2} z_p} \tag{2.53}$$

The dissipation transport equation is not solved at the wall adjacent cells, instead ϵ is determined using the following equation.

$$\epsilon_p = \frac{C_\mu^{3/4} k_p^{3/2}}{K z_p} \tag{2.54}$$

2.3 Adaptation of Governing Equations for the ABL

When modelling the ABL at full scale there are additional dynamics that need to be added to the RANS momentum Equation 2.39. These include buoyancy forces caused by thermal stratification and the Coriolis force due to the earth's rotation [3]. These effects can be introduced into the RANS momentum Equation 2.39 as an external force via an additional source term. Their effects are summed up here using a source term S_M defined in Equation 2.55 with ρ_0 the reference density, g_i and ι_i is defined in Equation 2.56 [3].

$$S_M = g_i(\rho - \rho_0) + \iota_i f_c \rho U_i \tag{2.55}$$

$$g_i^T = (0, 0, -g) \quad , \quad \iota_i^T = (-1, 1, 0) \tag{2.56}$$

f_c is defined as the Coriolis parameter using Equation 2.57 with the earth's rotation rate Θ_E and latitude Λ in geographical radians. The earth's rotation rate equals 7.292×10^{-5} rad s^{-1} [6].When viewed from a rotating reference frame only the component that acts perpendicular to the direction of the wind is considered. The Coriolis force causes the air to deflect from its original path of motion and causes increasing wind veer as a function of height. Only the horizontal components are considered as the vertical component is negligible due to the gravitational acceleration.

$$f_c = 2\Theta_E \sin(\Lambda) \tag{2.57}$$

2.3.1 Boussinesq approximation for buoyancy

The buoyancy term $g_i(\rho - \rho_0)$ in Equation 2.55 accounts for temperature based density variations in the ABL. According to the Boussinesq approximation for buoyancy density variations are small enough to be considered negligible except when appearing together with gravitational acceleration and is based on a combination of the ideal gas law, hydrostatic relation and potential temperature. The Fluent model treats density as a constant value in all solved equations except for the buoyancy term [17].

ABL temperature, pressure and density are linked over a wide range of conditions with the use of the ideal gas law [6]. With the assumption of incompressible flow this law can be simplified with the molar form of the ideal gas law approximated by:

$$\rho = \frac{Mp}{RT} \approx \frac{Mp_0}{RT} \tag{2.58}$$

This relation indicates that relative changes in temperature are now inversely proportional to changes in density; coupled with gravitational acceleration this results in vertical buoyancy forces [3]. The Boussinesq approximation accounts for density changes only in the vertical component of the momentum equation via a buoyancy term. Along with the continuity and momentum equations, the energy equation is then solved to model the temperature changes. With the Boussinesq approximation the buoyancy term then becomes the following based on a reference temperature T_0 and density ρ_0 [19].

$$g_i(\rho - \rho_0) = -\rho_0 \beta \left(T - T_0\right) g_i \tag{2.59}$$

Typical ABL density variations are small and the Boussinesq approximation is considered accurate [3] [6]. The Boussinesq approximation is used in various models other than those presented here and its use is incorporated into the Fluent RANS models by default as was discussed in Section 2.2.2 [17].

2.3.2 Monin-Obukhov similarity theory

The standard $k - \epsilon$ turbulence model as presented in Section 2.2.2 is not capable of accurately representing non-neutral conditions and modifications are needed to take stability effects into account [10] [1]. This is due to the fact that turbulence profiles generated using MOST are unbalanced with the turbulent transport equations to the standard $k - \epsilon$ turbulence model [3] [10]. This means that the profiles for velocity, temperature and turbulence will not demonstrate horizontal homogeneity in an empty domain. Several authors have presented methods to introduce modifications of the turbulent transport equations to overcome the inconsistencies. The changes are generally in the form of parametrizations of one or two model constants as listed in Table 2.3. Freedman and Jacobson [22] argued that the k-equation is in near equilibrium in stable atmospheric conditions and changes only need to be made in the ϵ-equation and introduced $C_{\epsilon 1}$ as a function of Richardson number to overcome the inconsistency. Alinot and Masson [10] proposed modifications to the transport equations by introducing $C_{\epsilon 3}$ as a function of the stability parameter z/L. The ϵ inlet profile was also modified to account for the k-equation imbalance. This method has been shown to work well for small domains [10] [1] but it can face issues in large domains due to the fact that the transport equation for k is still not in equilibrium with MOST. Parente et al. [23] proposed adding a new source term into the k-equation to allow the model to sustain the k profile. M.P. van der Laan et al. [1] have most recently proposed a new k-epsilon model consistent with MOST. The model is based on a combination of ideas from Parente et al. and Alinot and Masson where an additional analytical source term is added to the k-equation and a variable $C_{\epsilon 3}$ is used. This ensures both stable and unstable MOST profiles to be maintained, this model is referred to as the DTU model.

Turbulence modelling with MOST is described in the following section followed by descriptions of the Alinot and Masson (AM) and DTU models.

2.3.3 MOST turbulence modelling

MOST assumes that the ABL is steady and horizontally homogeneous and that the turbulent stresses $\overline{u'w'}$ and vertical turbulent heat flux $\overline{w'\theta}$ are constant with height [1]. This is coupled with the assumption that normalized velocity and potential temperature gradients can be described with analytical functions ψ_m and ψ_t as was described in Section 2.1.

The kinematic turbulent eddy viscosity:

$$\upsilon_t = \frac{-\overline{u'w'}}{\frac{\partial U}{\partial z}} \tag{2.60}$$

can then be represented as υ_{tMO} conforming to MOST as

$$\upsilon_{tMO} = \frac{K u_* z}{\psi_m \left(\frac{z}{L}\right)} \tag{2.61}$$

If one then writes the TKE rate equation in non-dimensionalized form it is possible to relate ψ_m to the MOST functions of the TKE components. Expressed mathematically this implies normalizing the TKE budget in Equation 2.62 by the surface-layer dissipation rate $u_*^3/(Kz)$ to obtain Equation 2.63 with the normalized dissipation ψ_ϵ defined in Equation 2.64 [1]..

$$O + P + B = \epsilon \tag{2.62}$$

$$\frac{Kz}{u_*^3}(O + P + B) = \psi_T + \psi_m + \psi_B = \psi_\epsilon \tag{2.63}$$

$$\psi_\epsilon = \frac{\epsilon K z}{u_*^3} \tag{2.64}$$

O , *P* and *B* respectively represents TKE transport, turbulence production due to shear and rate of turbulent production or destruction of TKE due to buoyancy. A typically used relation for ψ_ϵ is that of Panofsky and Dutton [24]:

$$\psi_\epsilon = \begin{cases} 1 - \frac{z}{L} & , L < 0 \\ \psi_\epsilon - \frac{z}{L} & , L > 0 \end{cases} \tag{2.65}$$

Due to homogeneity requirements it is required that the transport equations solved by the CFD code must be in balance with the formulae used to specify the boundary conditions of the turbulence quantities of the ABL. Richards and Hoxey [25] proposed one of the most widely used methods for the neutrally stratified ABL under the assumption of constant properties in the direction of flow and that the flow is driven by a shear stress applied at the top of the layer. This shear term is given by Equation 2.66.

$$\tau = \frac{Kz}{u_*}\frac{\partial u}{\partial z} \tag{2.66}$$

Applying these assumptions with the logarithmic wind speed profile of Equation 2.6 and using the relations from the $k - \epsilon$ model results in Equations 2.67 and 2.68 [1]. These have gained widespread use as boundary conditions for the neutral ABL and they are used in this study along with Equation 2.6 for velocity.

$$k = \frac{u_*^2}{\sqrt{C_\mu}} \tag{2.67}$$

$$\epsilon(z) = \frac{u_*^3}{kz} \tag{2.68}$$

A similar approach can be followed for the boundary conditions of MOST under the same assumptions. Using the standard eddy viscosity from the $k-\epsilon$ model, the MOST profile from Equation 2.61 and the dissipation from Equation 2.63 imply a vertical turbulent kinetic energy profile as follows:

$$k(z) = \left(\frac{\upsilon_{tMO}\epsilon}{C_\mu}\right)^{1/2} = \frac{u_*^2}{\sqrt{C_\mu}}\left(\frac{\psi_\epsilon}{\psi_m}\right)^{1/2} \tag{2.69}$$

The transport equations for k and ϵ can then be written as:

$$\frac{Dk}{Dt} = D_k + P - \epsilon + B \quad , \quad \frac{D\epsilon}{Dt} = D_\epsilon + (C_{\epsilon 1}P - C_{\epsilon 2}\epsilon + C_{\epsilon 3}B)\frac{\epsilon}{k} \tag{2.70}$$

where D_k and D_ϵ represent the diffusion-based transport of k and ϵ. The above relations are known to be inconsistent with the standard $k-\epsilon$ model [10] [1] and various methods have been proposed to deal with the inconsistency. The methods of Alinot and Masson (AM) and the DTU model is used in this study. Table 2.3 indicates the various models available and the adaptation needed for the models.

Table 2.3: Model constants for various $k-\epsilon$ models for ABL flows [1] [10] [15]

$k-\epsilon$ Method	C_μ	K	$C_{\epsilon 1}$	$C_{\epsilon 2}$	$C_{\epsilon 3}$	σ_k	σ_ϵ	σ_θ	k-eq.
Launder and Spalding	0.09	0.4	1.44	1.92	0	1	1.3	0.71	-
ABL neutral Sorensen	0.03	0.4	1.21	1.92	0	1	1.3	-	-
MOST Alinot & Masson	0.033	0.42	1.176	1.92	eq.2.73	1	1.3	1	-
MOST DTU model	0.03	0.4	1.21	1.92	eq.2.79	1	1.3	1	eq.2.75

2.3.4 Method I: Alinot and Masson

Based on measurements of the surface turbulent kinetic energy budget terms Alinot and Masson [10] obtained the following for ϵ

$$\epsilon(z) = \frac{u_*^3}{Kz}\psi_\epsilon\left(\frac{z}{L}\right) \tag{2.71}$$

To ensure the velocity, temperature and turbulence profiles for MOST represent exact solutions to the $k-\epsilon$ model, the values of C_μ, K, $C_{\epsilon 1}$ and $C_{\epsilon 3}$ are updated to those listed in Table 2.3. Using Equations 2.71 and 2.68 combined with Equation 2.61 obtains the value for $C_\mu = 5.48^{-2}$. $C_{\epsilon 1}$ is obtained from the ϵ transport equations by introducing MOST:

$$C_{\epsilon 1} = C_{\epsilon 2} - \frac{k^2}{\sigma_\epsilon\sqrt{C\mu}} = 1.176 \tag{2.72}$$

Finally $C_{\epsilon 3}$ is obtained using a fifth order polynomial:

$$C_{\epsilon 3}\left(\frac{z}{L}\right) = \sum_{n=0}^{5} a_n\left(\frac{z}{L}\right)^n \tag{2.73}$$

with the coefficients listed in Table 2.4. The polynomial in Equation 2.73 is not a complete analytical solution but instead an approximation and is only valid for $-2.3 < z/L < 2.0$ [1].

Table 2.4: Alinot and Masson $C_{\epsilon 3}$ model constants [10]

	$L > 0$		$L < 0$	
	$\left(\frac{z}{L}\right)$ < 0.33	$\left(\frac{z}{L}\right)$ > 0.33	$\left(\frac{z}{L}\right)$ < -0.25	$\left(\frac{z}{L}\right)$ >-0.25
a_0	4.181	5.225	-0.0609	1.765
a_1	33.994	-5.269	-33.672	17.1346
a_2	-442.398	5.115	-546.88	19.165
a_3	2368.12	-2.406	-3234.06	11.912
a_4	-6043.544	0.435	-9490.792	3.821
a_5	5970.776	0	-11163.202	0.492

2.3.5 Method II: DTU solution

The DTU method involves an additional source S_{kMO} in the k-equation [1].

$$\frac{Dk}{Dt} = D_k + P - \epsilon + B - S_{kMO} \tag{2.74}$$

with

$$S_{kMO} = \frac{u_*^3}{kz} \times \begin{cases} \left(\frac{L}{z}\right)(\psi_m - \psi_\epsilon) - \frac{\psi_h}{\sigma_\theta \psi_m} - \frac{C_{kD}}{4}\psi_m^{13/2}\psi_\epsilon^{-3/2} f_{us}\left(\frac{z}{L}\right) & , L < 0 \\ 1 - \frac{\psi_h}{\sigma_\theta \psi_m} - \frac{C_{kD}}{4}\psi_m^{7/2}\psi_\epsilon^{-3/2} f_{st}\left(\frac{z}{L}\right) & , L > 0 \end{cases} \tag{2.75}$$

employing the following stability functions:

$$C_{kD} = \frac{k^2}{\sigma_k \sqrt{C_\mu}} \tag{2.76}$$

$$f_{us}\left(\frac{z}{L}\right) = \left(2 - \frac{z}{L}\right) + \frac{16}{2}\left(1 - 12\frac{z}{L} + 7\left(\frac{z}{L}\right)^2\right) - 16\left(3 - 54\frac{z}{L} + 35\left(\frac{z}{L}\right)^2\right) \tag{2.77}$$

$$f_{st}\left(\frac{z}{L}\right) = \left(2 - \frac{z}{L}\right) - 10\frac{z}{L}\left(1 - 2\frac{z}{L} + 2\left(\frac{z}{L}\right)^2\right) \tag{2.78}$$

Finally $C_{\epsilon 3}$ is determined using Equation 2.79.

$$C_{\epsilon 3} = \frac{\sigma_\theta L}{z}\frac{\psi_m}{\psi_h}\left(C_{\epsilon 1}\psi_m - C_{\epsilon 2}\psi_\epsilon + [C_{\epsilon 2} - C_{\epsilon 1}]\,\psi_\epsilon^{-1/2} f_\epsilon\left(\frac{z}{L}\right)\right) \tag{2.79}$$

with:

$$f_\epsilon\left(\frac{z}{L}\right) = \begin{cases} \psi_m^{5/2}\left(1 - 12\frac{z}{L}\right) & , L < 0 \\ \psi_m^{-5/2}(2\psi_m - 1) & , L > 0 \end{cases} \tag{2.80}$$

For the DTU model S_{kMO} and $C_{\epsilon 3}$ are complete analytical solutions to MOST and is valid for the entire range of z/L [1]. This is important as the domain in ABL CFD models extend multiple kilometres above ground and using typical values for MOL the region in which the Alinot and Masson method is valid is quickly overcome.

Following the MOST assumptions, G_b from Equation 2.46 can be rewritten to yield Equation 2.81, shown here in its potential temperature form [1]. This expression for G_b is commonly used in literature [26] [27]. It can be considered as the ABL modeller's choice because it does not require $\frac{\partial T}{\partial x_i}$ which allows MOST to be used without solving the energy equation and also removes the issue where accurate steady simulations are difficult to obtain with buoyancy forces [1] [7]. Using this method yields steady-state results that can be implemented into typical wind turbine loading simulations. In this study the standard and MOST formulation of G_b are investigated. The MOST formulation is referred to as G_{bMO} and is presented in Equation 2.81.

$$G_{bMO} = \frac{g\upsilon_t}{\theta_0 \sigma_\epsilon \theta}\frac{\partial \theta}{\partial z} = -\upsilon_t \left(\frac{\partial U}{\partial z}\right) \frac{z\psi_t}{L\sigma_\theta \psi_m^2} \tag{2.81}$$

The MOST profiles for velocity and turbulence from Equations 2.18, 2.69 and 2.71 are used for the Alinot and Masson and the DTU model boundary conditions [1] [10].

2.3.6 ABL wall functions

The accuracy of ABL simulations can be severely comprised when wall-function roughness modifications used in the standard wall functions employed in Fluent as discussed in Section 2.2.3 are applied at the bottom of the computational domain [20]. These functions are developed based on experimental data for sand grain roughened pipes and channels [20]. The effect of improper wall functions cause unintended stream-wise gradients in the vertical mean wind speed and turbulence profiles [20]. The typical implication is unwanted acceleration of the flow near the surface which causes changes in velocity and especially turbulent kinetic energy, which leads to simulations that are not horizontally homogeneous[20]. The requirements for ABL wall functions can be described using the following four criteria [20] [19].

- A sufficiently fine mesh resolution close to ground, typically < 1m
- Horizontally homogeneous ABL flow in the empty domain
- The wall adjacent cell centre distance z_p should be greater than the physical roughness height K_s
- The correct relationship between ground roughness length z_0 and physical roughness height K_s can be derived

The relationship for point 4 can be derived by first order matching the wall function velocity profile and the neutral ABL velocity profile. Applying the K_s^+ relation for a fully rough equilibrium boundary layer $\tau_w = \rho u_*^2$, $C_s K_s^+ >> 1$ and combining Equations 2.49, 2.50 and 2.51 yield the wall function velocity [20].

$$\frac{U_p}{u_*} = \frac{1}{K}\ln\left(E\frac{z_p}{C_s K_s}\right) \tag{2.82}$$

The neutral wind velocity profile from Equation 2.6 can be rewritten with the same left side argument

$$\frac{U_p}{u_*} = \frac{1}{K}\ln\left(\frac{z}{z_0}\right) \tag{2.83}$$

These two equations must be equivalent in the first cell where $z = z_p$ which yields Equation 2.84 and recovers the standard neutral wind speed profile from Equation 2.6 [20].

$$K_s = z_0\frac{E}{C_s} \tag{2.84}$$

The above indicates the relation between ground roughness length z_0 and physical roughness height K_s. Fluent takes the input to its wall functions as physical roughness height and this equation must be adhered to for accurate ABL simulations. In this study this method is referred to as the modified roughness approach.

This method has gained widespread use [20], however, it faces some issues. Using a typical roughness length of 0.1 m and the Fluent default values for $C_s = 0.5$ and $E = 9.793$ the physical roughness height in Fluent would then become 1.9586 m. With the restriction that the cell centre of the wall adjacent cell should be greater than the physical roughness height this would result in a unacceptably course mesh at ground level[20], with a first cell height greater than 4 m. Also the standard wall function does not consider any direct effect of roughness on the turbulence quantities at the wall [2]. For these reasons there have been ABL specific wall function developments. The method used in this study is based on the work of Parente et al [2] which uses the boundary conditions of Richards and Hoxey [25]. The proposed model uses the following for wall velocity, turbulent kinetic energy and dissipation.

$$U_p = \frac{u_*}{K}\ln\left(\frac{z_p + z_0}{z_0}\right) \tag{2.85}$$

$$G_k = \frac{\tau_w^2}{K\rho C_\mu^{1/4} k_p^{1/2}\left(z_p + z_0\right)} \tag{2.86}$$

$$\epsilon_p = \frac{C_\mu^{3/4} k_p^{3/2}}{K\left(z_p + z_0\right)} \tag{2.87}$$

Comparing these relations with the standard wall functions the direct use and addition of roughness length z_0 is noted. There is also now a direct influence of roughness on the wall properties and also adds more freedom in mesh generation as the wall function does not impose any additional limitations on first cell height. In this study this method is referred to as the modified wall function approach.

MOST profiles as discussed in Section 2.3.3 approach neutral conditions at the wall and wall functions developed for neutral flow can be used [1].

2.4 Summary

Following the reviewed literature the following conclusions can be drawn:

The ABL can be in three main stability conditions namely stable, neutral and unstable. The neutral condition neglects thermal stratification. During a diurnal cycle stable conditions typically occur at night with cooler land temperatures while unstable conditions appear in day times with elevated temperatures. Stable conditions are characterized by lower ambient turbulence and vertical fluxes are suppressed by buoyancy forces. In unstable conditions the increase in vertical motion increases the boundary layer height and is also categorized by higher ambient turbulence.

MOST is used to describe the non-neutral wind profiles. The theory describes wind speed, temperature and turbulence profiles as a function of MOL, using the universal Dyer functions. MOL is used to categorize the various stability classes. Converting temperature to potential-temperature allows neutral and non-neutral stratification to be easily recognized. Various MOL calculation methods are presented, including: Gradient Richardson, bulk Richardson and a profile method that can be extended to a least-squares fit implementation.

In order to have an accurate ABL CFD model that accounts for the large scale physical mechanisms of the ABL, modifications to the standard RANS CFD model equations are required. The rotation of the earth causes a Coriolis force which acts on the momentum equation. The thermal stratification causes a buoyancy force, due to the small density variations in the ABL the Boussinesq approximation for buoyancy is typically employed. The standard wall function methods are not applicable to ABL models. A modified roughness and modified wall-function approach, based on the work of Parente et al, were reviewed.

The standard $k - \epsilon$ turbulence model is not capable of accurately representing non-neutral conditions. This is due to the fact that turbulence profiles generated using MOST are unbalanced with the turbulent transport equations of the standard $k - \epsilon$ turbulence model. Several authors have presented methods to introduce modifications to the turbulent transport equations to overcome the inconsistencies. Two primary methods were reviewed, the first is the Alinot and Masson model, the model uses a fifth order polynomial for $C_{\epsilon 3}$. The second model is based on the work of M.P. van der Laan et al. which uses an additional source term in the k-equation and an analytical solution for the $C_{\epsilon 3}$ variable.

Following the MOST assumptions the standard turbulence production due to buoyancy can be rewritten as a function of MOL and velocity gradient. Since this term does not require the temperature gradient using this method, it is not necessary to include the energy equation, which is known to cause solution fidelity problems in steady-state simulations.

Chapter 3

Data Acquisition and Analysis

This chapter focuses on applying the analytical equations from MOST as presented in Sections 2.1 and 2.3.2. The theory is applied to measured time series data from onsite meteorological masts located on a proposed wind farm location in the Eastern Cape in South Africa. The masts are used to gather representative windfield information about the onsite conditions. Three heights were measured for wind speed and direction, two for temperature and one for pressure and relative humidity. Mesoscale data obtained from a WRF (Weather Research and Forecasting) model were also downloaded at the same location. The analyses apply MOST to obtain the influence of atmospheric stability on the wind farm and determine the profiles for wind speed, temperature and turbulence. The study area and results are used in the complex terrain CFD analysis and validation study in Chapter 5. The analysis was conducted using Matlab 2016a (code included in Appendix C).

3.1 Study Area

A main overview of the study area is shown in Figure 3.1

Figure 3.1: Study area location. Map data: Google, 2017 DigitalGlobe, 2017 AfriGIS

The study area is characterized by a hill of 950 m above sea level (ASL) that drops down to 550 m ASL via steep and undulating terrain. Two meteorological masts are located on the hill where turbines would then be erected in between the two locations. The masts are located East-West approximately 7200 m apart.

A Northern view of the study area can be seen in Figure 3.2 with both mast locations shown. Mast 1 is the primary mast and used for the current data analysis study.

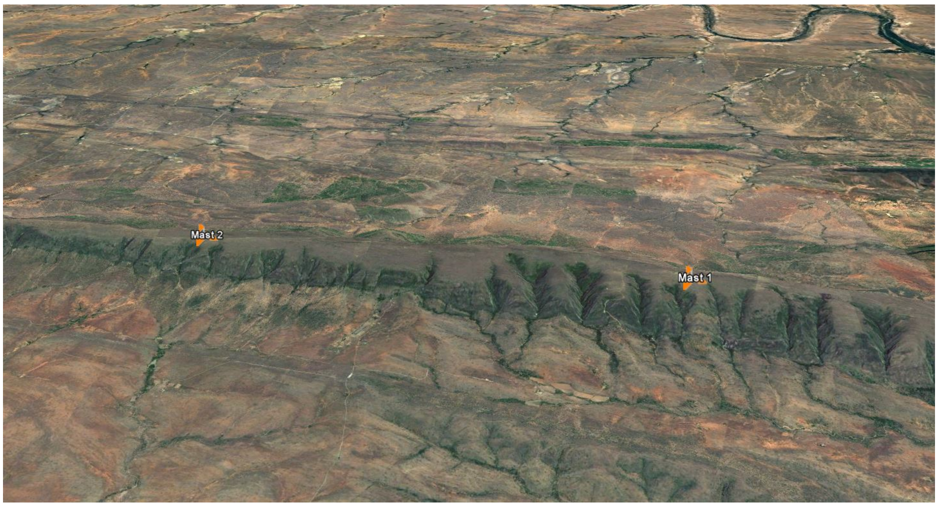


Figure 3.2: Northern view of study area. Map data: Google, 2017 DigitalGlobe, 2017 AfriGIS

An Easterly view along the hill is shown in Figure 3.3. The ground cover is typical open farmland with no major obstacles, this corresponds to a roughness height z_0 of 0.030 m.

Figure 3.3: Eastern view of study area. Map data: Google, 2017 DigitalGlobe, 2017 AfriGIS

The digital terrain model of the site is constructed from surveyed 5 m contour data over and around the main hill and then extended with 30 m shuttle radar topography mission data [28] to obtain a site model of 35 km × 25 km. The x and y axes are aligned with East and North respectively. The model indicates high topographical direction changes of up to 70° of inclination on the hill. Due to the steep terrain features linear flow models such as WAsP Engineering from DTU Wind Energy, which are specifically designed to work in flat terrain, are not suitable and CFD modelling is required [29]. The topographical angle of inclination is shown in Figure 3.4 below.

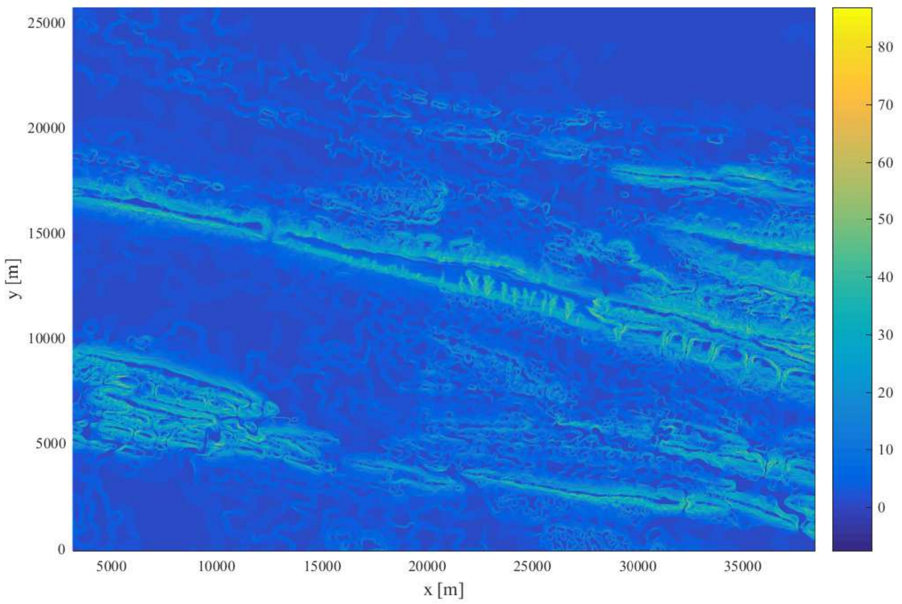


Figure 3.4: Angle of topographical inclination from the wind farm digital terrain model

3.1.1 Meteorological mast

In order to accurately represent the conditions onsite the data measurement campaign has to be of a certain standard. For the meteorological mast this can be summarised as follows [8]: Observations must be made at heights no lower than 0.75 of the proposed wind turbine hub on a lattice mast tower. The instruments must be located on slender booms extending from the mast much further than the diameter of the mast or the anemometer. Multiple readings along the mast are required with sufficient spacing to avoid interference. Experimentally calibrated first class anemometers and wind vanes must be used. Measurements are averaged over 10 minute periods concurrently for all sensors.

The masts used in this study are 82 m tall mast with cup anemometers located at 82 m, 60 m, and 40 m. Wind vanes are installed at 80 m and 40 m with temperature sensors at 80 m and 5 m. Pressure and relative humidity are measured at 5 m. Measnet Sensor calibration [30] has been successfully completed on all anemometers and wind vanes. The anemometers measure mean and standard deviation. A sampled 1 hour data set is shown in Appendix B.

3.1.2 Mesoscale data

The WRF model is a mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting and generates atmospheric simulations based on real data obtained from observations and analyses. EMD [31] uses its own in-house WRF model to allow users to download data sets at any location in the world based on this model. The data are available at any location with a temporal resolution of 1 hour. Typical correlation coefficients for the data sets to onsite data sets are in the range of 0.7-0.9 [31]. For this study the EMD model is utilized. The returned data for the set include: wind speed and direction at 10 m, 25 m, 50 m, 75 m, 100 m, 150 m and 200 m as well as temperature at 2 m and 100 m and pressure at 2 m. The data can be acquired for any time period with a 3 month delay to the current date and up to 20 years in the past. This allows simultaneous data sets to be downloaded to that of the measured data onsite. For this study WRF data sets were downloaded at the inlet location of the CFD model and also at the same location of Mast 1. The inlet location WRF data are used to describe the inlet boundary conditions and the WRF data at Mast 1 is used to understand the ability of WRF to predict stability on the site.

3.2 Calculation of Prevalence of Stability from Data

Two years worth of data were extracted for 2015-2017, full years are used to not introduce any seasonal bias in the data. The measured data have a recovery in excess of 90 % and the mesoscale data has 100 % recovery. The data were cleaned for faulty readings and outliers using standard wind industry data cleaning procedures. For this study Mast 1 is the primary mast and is used for the results displayed in this section. To determine the frequency of each stability condition the temperature was converted to potential temperature using Equation 2.14 for each reading. Using the potential temperature gradient and Equations 2.15-2.17 the reading is then classified as neutral, stable or unstable. The Monin-Obukhov Length of each reading is calculated using the three measurement heights for velocity to perform a non-linear least squares fit with the corresponding stability velocity profile from Equation 2.18. Using the conditions in Table 2.1 the data are then binned into the various classes. The results for the data from Mast 1 set can be seen in Figure 3.5. The results show that only 11 % is spent in the neutral condition, this shows that using the standard ABL CFD model for this site would be applicable to a very small portion of the actual onsite conditions. 36 % of the time is spent in the extremely unstable condition. This is typical in countries in the Southern hemisphere due to the high daytime temperatures. 40 % of the time the site was in the stable condition.

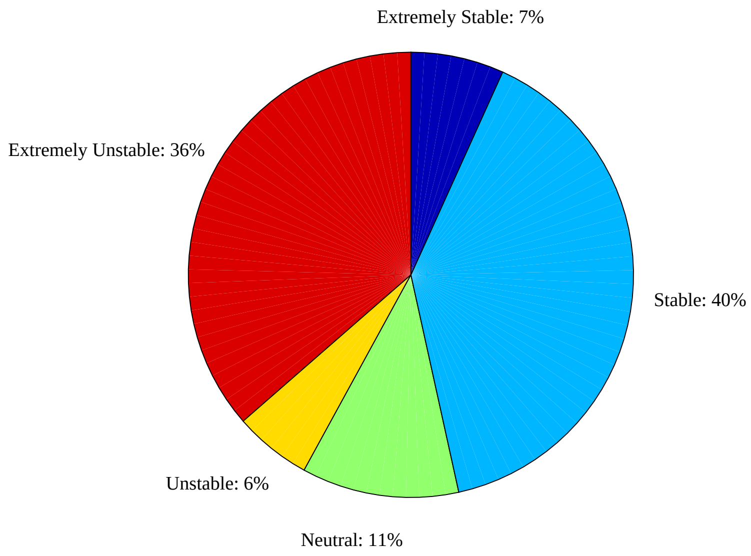


Figure 3.5: Stability frequency classification for Mast 1

Windfarm CFD models simulate the wind flow from 12 different directions, this means the data need to be split into 30° bins. The stability rose in Figure 3.6 shows the sector wise distribution of stability from the mast obtained by using the top wind vane for directional binning. The prevailing wind directions can be identified as sectors 120-180°. This shows that the wind mainly approaches the hill from the South Eastern direction. It can also be noted that the stability percentage remains mainly unchanged within each sector and indicates that stability is independent of the direction for this location and time span.

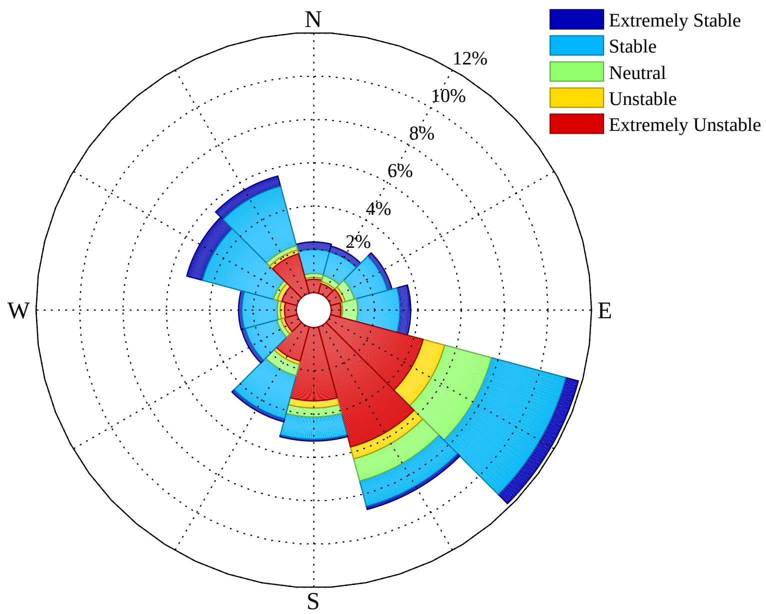


Figure 3.6: Stability rose for Mast 1

The 10 minute diurnal evolution of stability can be seen in Figure 3.7. The main trend is identifiable with strong (90%) extremely unstable and unstable prevalence during day-time with stable conditions dominating the night time. This matches with the typical diurnal ABL evolution presented in Section 2.1. This diurnal cycle is used to average the data for all of the following calculations and any mean determined is weighted against the time it occurs in the diurnal cycle. This means when determining statistics for the extremely stable region the effects of the conditions occurring in night time is weighted more heavily than the few times it occurs during day time. This is done to alleviate the effects of stratification occurring outside of its normal conditions, for example a day time rain storm with high cloud cover can cause the extremely stable condition during daytime.

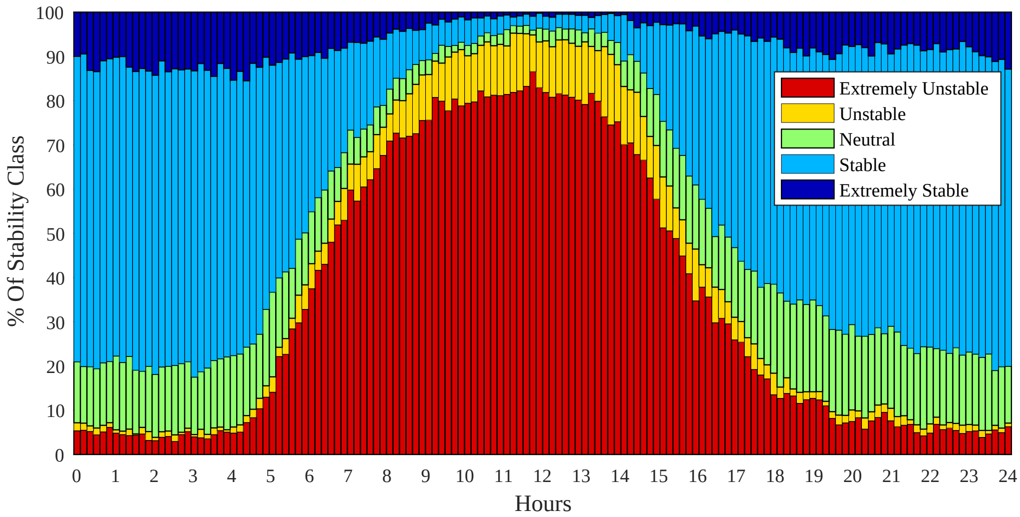


Figure 3.7: Diurnal stability classification for Mast 1

Three of the main conditions effecting turbine power performance and suitability are: wind speed, turbulence and wind shear. Turbulence Intensity (TI) can be determined from anemometer data using the fraction of standard deviation ζ_U to mean wind speed U using Equation 3.1 [8].

$$\text{TI} = \frac{U}{\zeta_U} \tag{3.1}$$

Wind shear is defined in terms of a shear exponent α as shown in Equation 3.2 using a power law for wind speed as a function of height $u(z)$ based on a reference wind speed from a fixed height $u(z_{ref})$. A larger shear exponent indicates a faster growth of wind speed with height than a lower shear exponent. This equation is solved for α using a least squares fit with the three measurement heights. This was completed for every reading to obtain the instantaneous shear exponent.

$$\frac{u(z)}{u(z_{ref})} = \left(\frac{z}{z_{ref}}\right)^{\alpha} \tag{3.2}$$

Rewriting this equation in a linear form results in:

$$\ln(u(z)) = \ln(u(z_{ref})) + \alpha \ln\left(\frac{z}{z_{ref}}\right) \tag{3.3}$$

The diurnal conditions are analysed by assuming the central limit theorem allowing the mean to be taken at each 10 minute bin of the measured data by fitting a normal distribution at each time step. The results for turbulence intensity and shear exponent are shown in Figure 3.8. It can be seen that in the extremely unstable condition the turbulence intensity is much higher than in any other condition. The daytime extremely unstable turbulence exceeds 0.16. This is an important factor as wind turbines are designed within certain turbulence classes and above 0.16 a class-A turbine is required [8]. Meaning that if stability is neglected and not modelled an unsuitable turbine could be used onsite. The shear exponent also indicates how in the extremely unstable and unstable region the shear exponent is very low due to the vertical motion of the air that limits wind profile growth. While the extremely stable and stable conditions both show very high wind shear values. Understanding the time spent at these high shear conditions is important for turbine suitability as high shear leads to uneven turbine loading. The diurnally averaged results for the shear exponent are shown in Table 3.1. Figure 3.9 shows the diurnal MOL and illustrates how the diurnal cycle starts stable during night time and changes to extremely unstable as the temperature starts to rise in day time before reverting back to stable as the cooler night time starts.

Table 3.1: Wind shear exponent results from Mast 1 - Sector 180°

	Extremely Unstable	Unstable	Neutral	Stable	Extremely Stable
Shear Exponent α	0.001	0.059	0.079	0.246	0.680

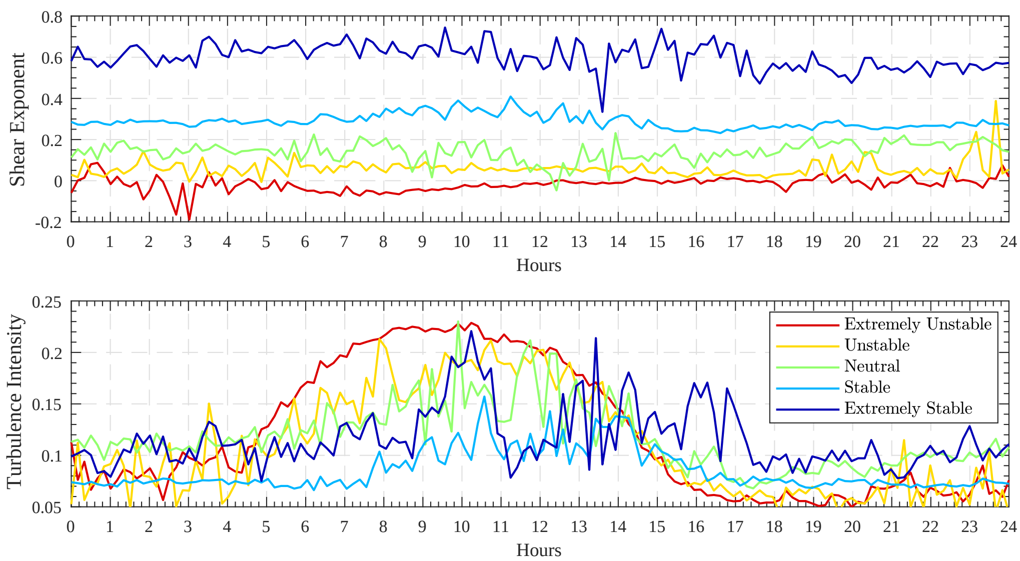


Figure 3.8: Diurnal turbulence intensity and wind shear exponent

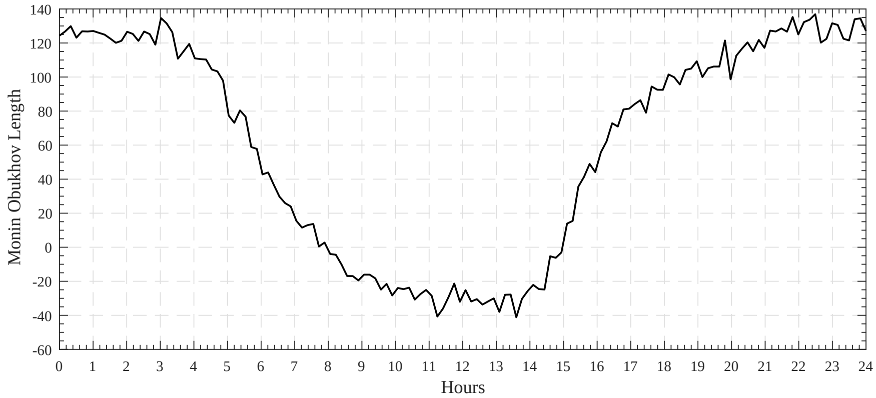


Figure 3.9: Diurnal Monin-Obukhov Length

Based on the stability prevalence results it is clear that non-neutral stratification is present on the site and that it influences the conditions to such an extent that using only the standard neutral CFD model the necessary effects would not be captured on-site.

Table 3.2 compares the stability distribution obtained using the measured and mesoscale data. There is a negligible difference, expect for the unstable and neutral conditions. The difference can be attributed to the fact that these conditions are non-dominating and statistically larger variations are present during the condition due to their less frequent occurrence. In the two dominating conditions (extremely unstable and stable) only a 1% difference is present, this shows the mesoscale data are able to capture stability for the site location.

Table 3.2: Stability classification difference between measured and mesoscale data

	Extremely Unstable	Unstable	Neutral	Stable	Extremely Stable
Mast [%]	36	6	11	40	7
Mesoscale [%]	37	13	3	39	8
Difference [%]	1	7	8	1	1

3.3 Calculation of Vertical Profiles from Data

Turbulent fluxes of momentum and heat near the surface are of primary concern to the design of wind farms as they determine the shape of the velocity, temperature and turbulence profiles. These profiles are calculated using the measured data. Sector 180° (wind direction from 165-195°) is used as the test sector for this study as it is one of the prevailing wind directions as well as being located directly south of the main hill. Using this direction as an inlet for the CFD model allows a suitable upwind and downwind fetch along the flat terrain. All of the results presented further are based on this sector only.

First only the data from the relevant sector are extracted. Using the diurnally weighted average of the data at each height a fixed data point for velocity and potential temperature is then calculated for each stability condition. The same is done for the MOL. This process yields the results in Table 3.3.

Table 3.3: Average measured velocity, potential temperature and MOL - Sector 180°

	u_{82} [$\mathrm{ms^{-1}}$]	u_{60} [$\mathrm{ms^{-1}}$]	u_{40} [$\mathrm{ms^{-1}}$]	θ_{80} [K]	θ_5 [K]	MOL [m]
Extremely Unstable	7.00	6.95	6.97	299.6	298.4	-5.8
Unstable	8.25	8.13	7.86	299.4	298.5	-230.0
Neutral	8.10	7.90	7.71	294.1	294.2	N/A
Stable	5.68	5.29	4.87	295.3	296.0	221.8
Extremely Stable	2.65	2.19	1.76	294.1	294.8	26.3

Using Equation 2.18 with the corresponding stability functions in Equation 2.20 to 2.25 and the data from Table 3.3 in a non-linear least squares fit allows the solution of the frictional velocity u_* to be obtained such that the velocity profile is the best fit to the data. z_0 is set to the roughness height on-site of 0.030 m. The initial guess for u_* is obtained using the profile method from Equations 2.31 along with the top and bottom height. The results for frictional velocity are shown in Table 3.4 and the velocity profile results can be seen in Figure 3.10. The crosses indicate the averaged data points to which the profiles are fitted. In the extremely stable and stable condition the velocity profiles are flat, indicating a high increase in windspeed as a function of height. The opposite is true for the unstable and extremely unstable conditions where there is hardly any change of velocity with height. It can also be seen that the extremely stable condition is much more prevalent a lower wind speeds.

The procedure is repeated for potential temperature using Equation 2.19 with the corresponding stability functions from Equations 2.20 to 2.25 and the data from Table 3.3. This time, however, there are two unknowns, potential temperature length scale and also ground potential temperature. Once again the profile method in Equation 2.32 is used as initial guess for θ_*. Solving for θ_* and $\theta(z_0)$ using a non-linear regression yields a direct solution since there are 2 unknowns and 2 data points. The resulting profiles are shown in Figure 3.11. The crosses indicate the averaged data points to which the profiles are fitted. The profiles are located along the temperature axis in their expected positions with the unstable conditions occurring during the higher daytime temperatures and stable during the cooler night-time temperatures. The shape of the profiles also corresponds with Stable $\frac{\partial\theta}{\partial z} < 0$ and unstable $\frac{\partial\theta}{\partial z} > 0$. The neutral condition appears vertical since during this condition the potential temperature gradient matches that of the dry adiabatic lapse rate. The results for potential temperature length scale and ground potential temperature are shown in Table 3.4.

Using Equation 2.33 and the determined frictional velocity and potential temperature length scale the ground heat flux is calculated. The stable condition is characterized by negative heat flux due to the heat transfer from the air to the ground, the heated ground in unstable conditions causes a positive heat flux and the neutral condition has a heat flux close to zero. The density at the mast location is determined using Equation 2.34 and the diurnally averaged pressure, relative humidity and temperature data at 5 m. The results for heat flux and density are shown in Table 3.4.

Table 3.4: Results from Mast 1 data analysis - Sector 180°

	Extremely Unstable	Unstable	Neutral	Stable	Extremely Stable
Frictional Velocity u_* [m s^{-1}]	0.361	0.332	0.308	0.181	0.040
Temperature Length Scale θ_* [K]	-1.126	-0.217	0.000	0.064	0.016
Ground Temperature $\theta(z_0)$ [K]	316.6	303.6	294.2	294.0	293.8
Density ρ [kg m^{-3}]	1.082	1.082	1.101	1.097	1.103
Heat Flux Q_H [W m^{-2}]	441.7	78.4	0.00	-12.8	-0.8

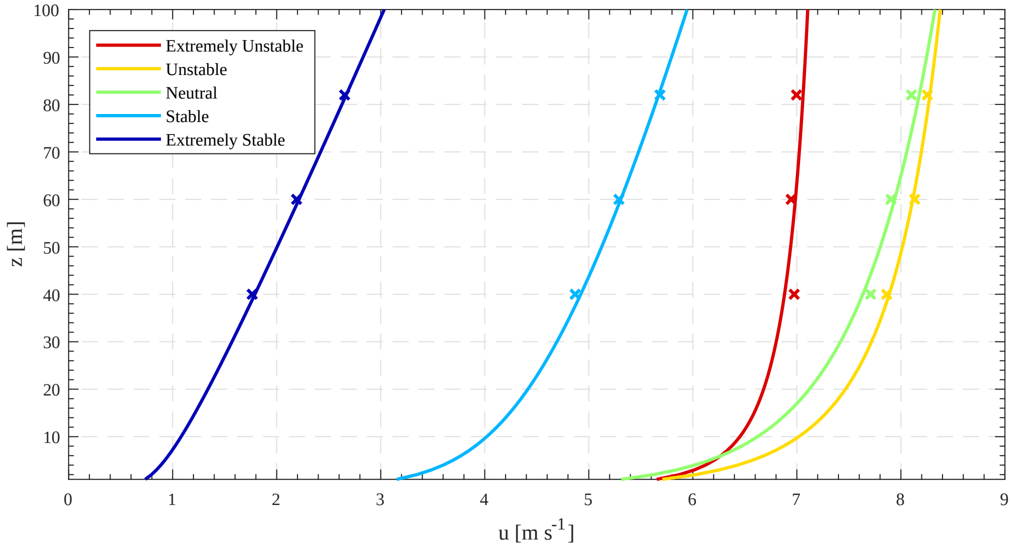


Figure 3.10: Measured velocity profiles - Sector 180°

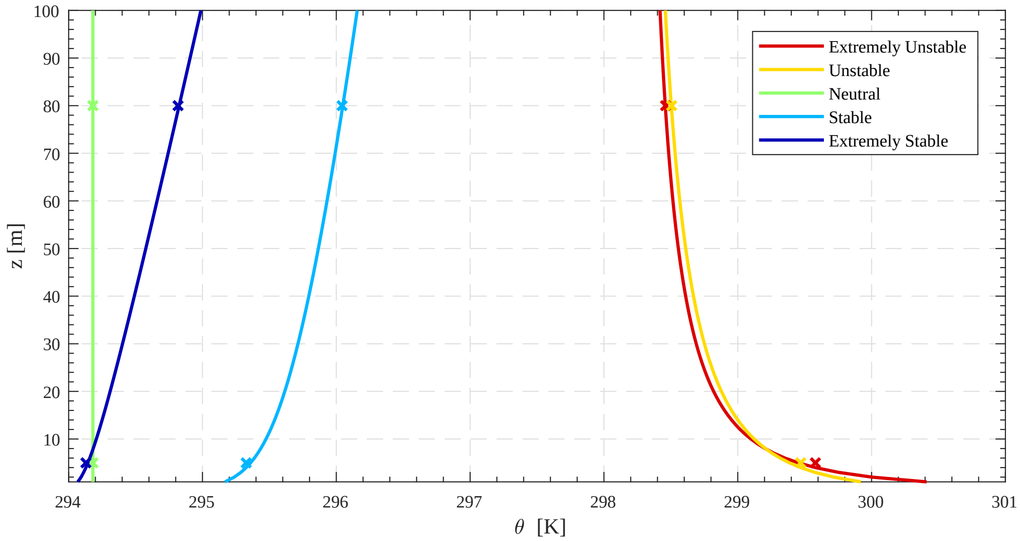


Figure 3.11: Measured potential temperature profiles - Sector 180°

The turbulence profiles for k and ϵ are determined using Equation 2.69 and 2.71 with the frictional velocity calculated above. The resulting profiles are presented in Figures 3.12 and 3.13. The turbulent kinetic energy k has a much higher value in the unstable conditions than that of the stable regions. This is to be expected due to fluctuations present in this state. In the stable regions the fluctuations are suppressed and yields the vertical profiles with a much lower value than that of the other conditions. The turbulent dissipation rate ϵ profiles highlight how the dissipation is increased close to ground level.

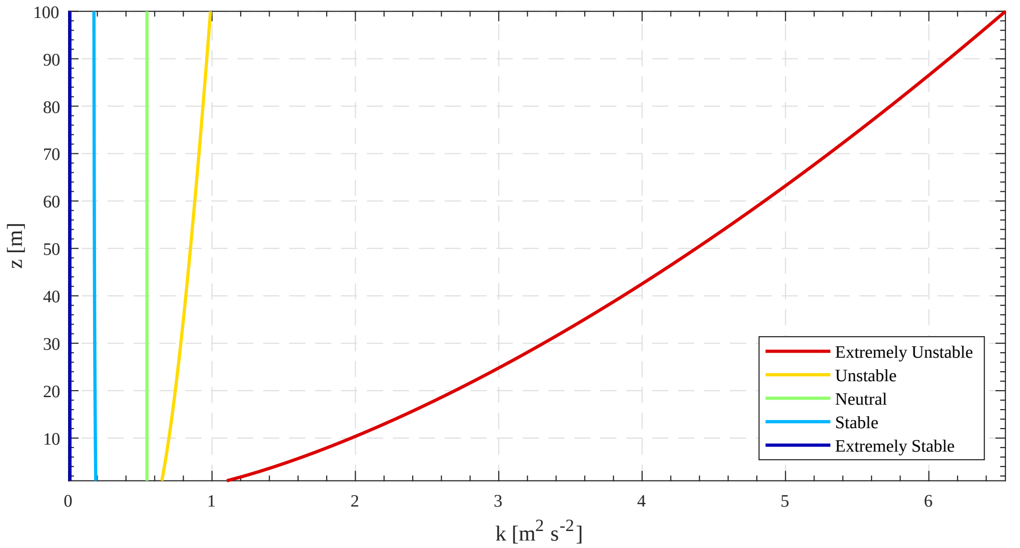


Figure 3.12: Turbulent kinetic energy from measurements - Sector 180°

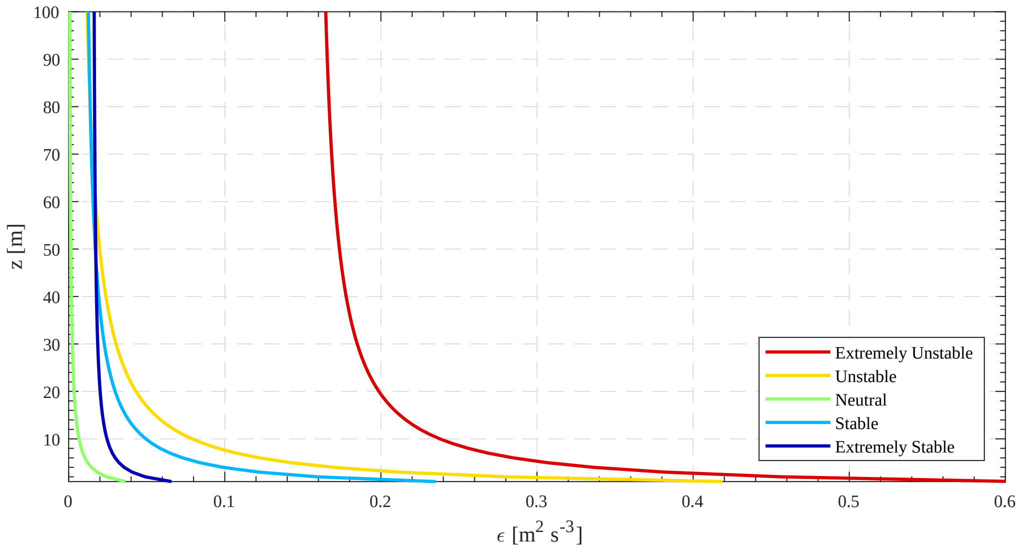


Figure 3.13: Turbulent dissipation rate from measurements - Sector 180°

3.4 Summary

From the data analysis it can be concluded that non-neutral stratification is present at the site location and assuming the standard neutral conditions would not result in a accurate description of the site conditions. The two most dominating conditions are the extremely unstable and stable conditions which account for more than 75% of the stratification onsite. The results from the mesoscale data stability prediction showed that the mesoscale data are able to capture the various stability conditions.

The analysis of the time series data successfully showed that the method can be used to determine accurate profiles of velocity and potential temperature and the calculation of MOL based on a non-linear least squares profile fit.

The site conditions used in the validation study of the complex terrain ABL CFD model in Chapter 5 is obtained from the analysis performed on the data from Mast 1 with the results listed in Tables 3.3 and 3.4. Sector 180° is used as the test sector for the validation study as it is one of the prevailing sectors as well as being located upstream perpendicular to the main hill.

The data analysis procedure is repeated using the data from Mast 2. It is used in the validation study by comparing the measured velocity profiles from Mast 2 with the ability of the CFD model to predict the velocity profiles using the data from Mast 1. The procedure is also applied to a mesoscale data set obtained at the inlet location to create the inlet vertical profiles needed for the CFD model.

Chapter 4

ABL CFD Model

The MOST modifications presented in Section 2.3 are applied to the Fluent 18.1 RANS model equations by user defined functions (UDF). The numerical implementation of these functions are presented along with a description of the CFD model. The implementations are tested by their ability to maintain inlet profiles in an empty computational domain. Three main cases are tested. A comparison is made of the modified roughness and wall function approaches. The AM and DTU models are tested using the MOST G_b formulation. Finally the standard and MOST G_b formulations are both tested using the AM model.

4.1 Numerical Implementation

User defined functions (UDFs) are additional functions that can be loaded into the ANSYS Fluent Solver to enhance the standard features. UDFs are defined by various *DEFINE* macros provided in Fluent. The UDFs are coded using the C language. They use additional functions and macros that can access Fluent solver data and perform numerous tasks [32]. Each UDF is hooked into the Fluent solver prior to performing a simulation. The following UDFs are used in this study:

- *DEFINE_PROFILE* Specification of the velocity, temperature, turbulence and wall roughness profiles at the boundary conditions.
- *DEFINE_SOURCE* Specification of the source terms in the momentum and turbulence transport equations.
- *DEFINE_WALL_FUNCTIONS* Implementation of the modified wall function
- *DEFINE_INIT* Initialization of the solution
- *DEFINE_EXECUTE_AT_END* Custom function that executes at the end of each iteration to compute the height above ground.
- Data access macros allow access to stored variables at each cell centroid location. These include velocity components, turbulence values and gradients.

Three main UDF sets have been developed, one each for neutral, unstable and stable conditions. They are included in Appendix D. The UDFs are compiled inside Fluent using Microsoft Visual Studios on Windows and the internal TUI commands on Linux. Each UDF is controlled by specifying values in the *#define* section of the code. The same UDF is used for the extreme and normal cases only with different values in the *#define* section.

4.1.1 Momentum source terms

The Coriolis force is included in the source term S_m in Equation 2.39. Equation 2.57 is applied in both the X and Y momentum equations in Fluent. u and v is respectively set to fluid x and y velocity obtained at each cell centroid using the appropriate data access macro. The local latitude of each cell is used by adding the difference between the latitude at the inlet and the latitude at the cell of interest.

The buoyancy momentum source $g_i(\rho - \rho_0)$ is included by activating the energy equation and the Boussinesq Approximation. This is an included feature in Fluent and no UDF code is needed to control the source. For the MOST G_b formulation the energy equation is not activated and the buoyancy momentum source is neglected.

4.1.2 Turbulence source terms

For the DTU method the S_k source term in the turbulent kinetic energy transport equation, Equation 2.43, includes S_{kMO} and G_{bMO}. It is described in the UDF using the source term in Equation 4.1. G_{bMO} is included since the DTU model does not activate the energy equation and Fluent therefore neglects G_b from the turbulent kinetic energy transport equation. S_{kMO} and G_{bMO} are given by Equations 2.75 and 2.81 respectively. Fluent stores the gradients required to describe $\frac{\partial U}{\partial z}$ in G_{bMO} and it is extracted at each cell centroid using the appropriate data access macro. The velocity has two horizontal components (u and v) and $\frac{\partial U}{\partial z}$ is therefore evaluated using the Euclidean norm shown in Equation 4.2.

$$S_k = -\rho S_{kMO} + \mu_t G_{bMO} \tag{4.1}$$

$$\frac{\partial U}{\partial z} = \sqrt{\left(\frac{\partial u}{\partial z}\right)^2 + \left(\frac{\partial v}{\partial z}\right)^2} \tag{4.2}$$

The frictional velocity in the S_k source term is not kept constant but instead calculated by rewriting Equation 2.69 to obtain

$$u_* = C_\mu^{1/4} k^{1/2} \left(\frac{\psi_\epsilon}{\psi_m}\right)^{-1/4} \tag{4.3}$$

The ϵ source terms included in the turbulence energy dissipation rate transport equation, Equation 2.44, are based on modifications to the $C_{\epsilon 3}$ constant. Fluent by default sets $C_{\epsilon 3}$ to zero. In order to reintroduce $C_{\epsilon 3}$ in a manner consistent with Equation 2.44, S_ϵ takes the following form.

$$S_\epsilon = C_{\epsilon 1} \frac{\epsilon}{k} C_{\epsilon 3} G_b \tag{4.4}$$

For this study there are three versions of the S_ϵ source term: The DTU method, AM with the energy equation and AM without the energy equation. The versions are presented in Equations 4.5-4.7 with $C_{\epsilon 3}$ obtained from Equations 2.79 and 2.73 for the DTU and AM methods respectively. The $C_{\epsilon 1}$ constant for each method is listed in Table 2.3.

DTU with G_{bMO}:

$$S_\epsilon = C_{\epsilon 1} \frac{\epsilon}{k} C_{\epsilon 3} \mu_t G_{bMO} \tag{4.5}$$

AM with G_{bMO}:

$$S_\epsilon = C_{\epsilon 1} \frac{\epsilon}{k} C_{\epsilon 3} \mu_t \left(-G_{bMO}\right) \tag{4.6}$$

AM with energy and G_b:

$$S_\epsilon = C_{\epsilon 1} \frac{\epsilon}{k} C_{\epsilon 3} \mu_t G_b \tag{4.7}$$

The AM is model is only valid for $-2.3 < z/L < 2.0$ and outside this region S_ϵ is set to 0. The source terms are introduced only after 5 iterations so that divergence does not occur if ill-posed initializations exist that cause extreme gradients.

The height of the boundary layer must be taken into account, above this height the inlet profiles and sources are set to the fixed value they would attain at the boundary layer edge. The values used in this study follow typical ABL heights. The stable boundary layer is known to be more shallow and is set to 600 m AGL, while the vertical motions in the unstable condition cause an increased boundary layer height and is set to 800 m. The neutral boundary layer is set as 1000 m AGL. For the empty domain study all of the heights are, however, set to 1000 m AGL to not introduce any additional gradients into the solution.

4.1.3 Temperature variations

For the AM model including energy and the standard G_b, the temperature variations are included by activating the energy equation in Fluent. The potential temperature profiles obtained using Equation 2.19 are converted to standard temperature inlet profiles in Fluent using Equation 2.14. The method is employed only in the empty domain test and the pressure above ground is calculated using the standard barometric formula based on Fluent's operating pressure and temperature [6].

$$p = p_{oper} \left(\frac{T_{oper}}{T_{oper} + L_b z} \right)^{\frac{-gM}{RL_b}} \tag{4.8}$$

with

$$\begin{aligned}
p_{oper} &= \text{Operating pressure} = 101325 \text{ Pa} \\
T_{oper} &= \text{Operating temperature} = 288.16 \text{ K} \\
M &= \text{Molar mass dry dir} = 29 \text{ g mol}^{-1} \\
g &= \text{Gravitational acceleration} = -9.81 \text{ m s}^{-2} \\
R &= \text{Universal gas constant} = 8.314 \text{ J (K mol)}^{-1} \\
L_b &= \text{Standard temperature lapse rate} = -0.0065 \text{ K m}^{-1}
\end{aligned}$$

4.1.4 Wall function

Two versions of wall functions are investigated in this study. To implement the modified roughness approach, the physical roughness height input into Fluent is simply set equal to the roughness length relation in Equation 2.84.

The modified wall function approach is incorporated using a user-defined wall function. The wall function is designed according to the ABL wind velocity profile in Equation 2.83.

In laminar flow

$$u^{+} = z^{+} = \frac{u_{*} z_{p}}{\rho} \tag{4.9}$$

and in the fully turbulent region, written here to preserve the form of Equation 2.82

$$u^{+} = \frac{1}{K} \ln\left(\tilde{E}\tilde{z}^{+}\right) \tag{4.10}$$

with

$$\tilde{E} = \frac{\mu}{\rho z_{0} u_{*}} \quad , \quad \tilde{z}^{+} = \frac{\rho\left(z_{p} + z_{0}\right) u_{*}}{\mu} \tag{4.11}$$

where $u^{+} = U_{p}/u_{*}$ is the dimensionless wall tangential velocity. $\tilde{z}^{+}$ is the non-dimensional distance and is simply the standard z^{+} shifted by z_{0}. For Equation 4.11 u_{*} is not kept constant but instead calculated using Equation 4.12 obtained by rewriting the neutral k profile from Equation 2.67. MOST profiles approach neutral conditions at the wall and the neutral relation between u_{*} and k is used instead of the non-neutral relationship used in the sources.

$$u_{*} = C_{\mu}^{1/4} k^{1/2} \tag{4.12}$$

To incorporate the wall function UDF into fluent the function must compute and return u^{+} along with its first and second order derivatives taken with respect to z^{+} in both laminar and turbulent regions. Using Equations 4.9 and 4.10 for u^{+} laminar and turbulent respectively results in the following Equations.

Laminar	Turbulent	
$u^{+} = z^{+}$	$u^{+} = \frac{1}{K} \ln\left(\tilde{E}\tilde{z}^{+}\right)$	(4.13)
$\frac{\partial u^{+}}{\partial z^{+}} = 1$	$\frac{\partial u^{+}}{\partial z^{+}} = \frac{1}{K\tilde{z}^{+}}$	(4.14)
$\frac{\partial^{2} u^{+}}{\partial z^{+2}} = 0$	$\frac{\partial^{2} u^{+}}{\partial z^{+2}} = -\frac{1}{K\tilde{z}^{+2}}$	(4.15)

Fluent automatically uses u^{+} and the derivatives to calculate G_{k} and ϵ_{p} at the wall adjacent cell and thus recovers Equations 2.85-2.87 [32]. With the user-defined wall function the physical roughness specification in Fluent is not necessary as the roughness length is input directly into the UDF.

4.1.5 Height above ground

The source terms are function of height above ground and requires accurate information on the distance between the cell centroid and the bottom boundary. The z coordinate of the cell cannot be used due to terrain features on the bottom boundary. Fluent does not have a standard macro to access height above ground. This limitation is overcome by introducing a user defined scalar (UDS) that is solved inside Fluent. Fluent allows the specification of user defined scalars (χ) that are solved via

$$\frac{\partial \rho \chi_j}{\partial t} + \frac{\partial}{\partial x_i}\left(\rho u_i \chi_j - D_j \frac{\partial \chi_j}{\partial x_i}\right) = S_{\chi j} \quad j = 1, .., N \tag{4.16}$$

where D_j and $S_{\chi j}$ are the diffusion coefficients and source terms for each of the N scalar equation added [17]. The approach to calculate the wall distance involves solving an additional UDS using a diffusion only transport equation with a uniform unity source term through the entire domain [33]. The UDS value (χ) is set to zero on walls to which the distance is to be calculated, in this case the ground and the normal flux $\frac{\partial \chi}{\partial n}$ is set to zero for all other. The UDS is then used to reconstruct the wall distance d to the selected boundary using Equation 4.17[33] where $\boldsymbol{\nabla}$ represents the gradient operation. The UDS is incorporated in an 'Execute at end' UDF that calculated at the end of an iteration, for this reason the source terms are not activated at the first iteration as the height above ground would not yet have been calculated.

$$d = -|\boldsymbol{\nabla}\chi| = \sqrt{\boldsymbol{\nabla}\chi \cdot \boldsymbol{\nabla}\chi + 2\chi} \tag{4.17}$$

4.1.6 Initialization

The solution is initialized using the velocity and turbulence profiles from Equations 2.18, 2.69 and 2.71. This ensures that the gradients used to evaluate the first iteration do not cause divergence if the standard initialization was ill-posed and also helps speed up the solution procedure.

4.2 Model Settings

The general setup of the ABL CFD model is described here and is identically employed in the empty domain and wind farm simulation performed in Chapter 5.

The inflow is along the y axis, the x axis is horizontally perpendicular to the inlet and z is the height above ground. The inlet boundary is a x-z plane located upstream of the computational domain. The inlet profiles are set via the 'define profile' UDF based on the velocity and turbulence profiles and are imposed normal to the inlet boundary. The top boundary is a x-y plane and is also treated as an inlet using the same profiles as the inlet. The velocity is described in the y direction only. The sides of the domain are y-z planes and are set to symmetry boundary conditions. The outlet boundary is a x-z plane located downstream of the computational domain and uses an outflow condition that allows extrapolation of the relevant flow variables from inside the domain onto the outlet boundary. The bottom of the domain is set to a zero-slip wall.

The standard limit of 10^5 for turbulent viscosity ratio inside Fluent is based on common industrial internal flows and for the ABL simulation it is increased to 10^{10}.The solution algorithm adopted in Fluent uses the coupled method for pressure-velocity coupling. The Presto (PREssure STaggering Option) is used for pressure spatial discretization. A least squares cell based method is used for the gradients and all other properties adopt a second-order upwind scheme based on a multi-linear reconstruction approach. All simulations are performed under steady-state conditions.

4.2.1 Fluid properties

The fluid used is air with the properties listed in Table 4.1. These settings are retained throughout the study except in Chapter 5 where the site specific air density is used.

Table 4.1: Air Properties [6]

Density ρ [kg m^{-3}]	1.225
Specific Heat C_p [J (kg K)$^{-1}$]	1006.43
Thermal Expansion β [K^{-1}]	0.0032
Viscosity μ [kg (m s)$^{-1}$]	1.7894 $\times 10^{-5}$

4.3 Empty Domain Model

The first step towards the validation of the proposed approach and its numerical implementation is to demonstrate that the methods produce sustainable ABL profiles of velocity and turbulence. The first objective is thus to prove horizontal homogeneity of the fully developed inlet profiles in an empty domain. A schematic of the computational domain used for this study is shown in Figure 4.1.

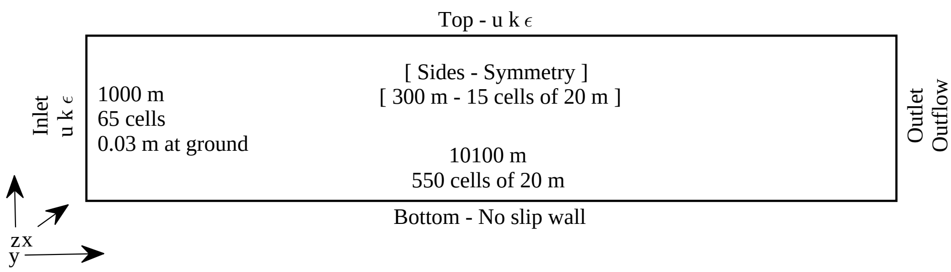


Figure 4.1: Computational domain - Empty domain
Square brackets indicate properties along the x dimension

The domain is rectangular cuboid with dimensions of 300 m, 10100 m and 1000 m in x, y and z respectively. The domain is discretized with a uniform grid in the x and y directions of 20 m. In the z direction the ground cell height equals 0.030 m and expands using geometric growth ratio of 1.14 with 65 cells. The complete mesh is comprised of 492375 cells. Figures 4.2 and 4.3 respectively show the cells close to ground and a full overview of the mesh. Typical upstream inlet locations in ABL CFD models are around 2000-5000 m from the main features and using this sized domain allows the model results to be checked at distances up to 10000 m. Only the stability-based source terms are included in the horizontal homogeneity tests and Coriolis force is neglected.

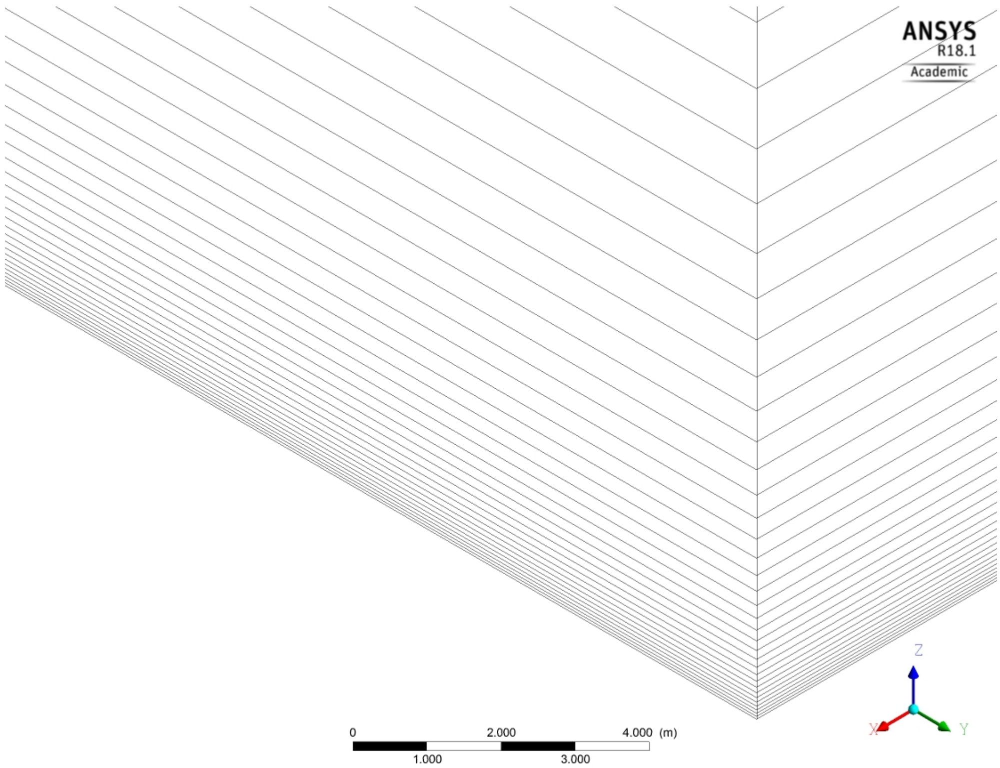


Figure 4.2: Close up of z refinement - Empty domain

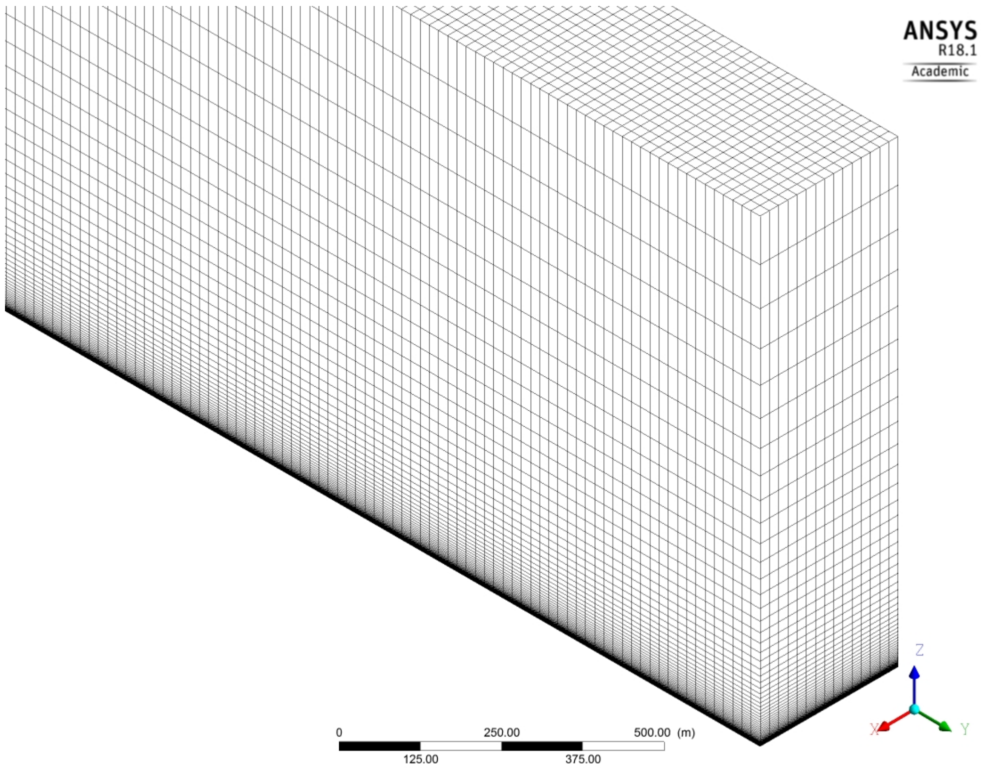


Figure 4.3: Mesh overview - Empty domain

4.3.1 Wall function test results

Three roughness length values were used to test the modified roughness and wall function approach. These are listed in Table 4.2 along with the modified roughness method's corresponding physical roughness height using Equation 2.84. For the modified wall function method the roughness length is directly used in the user-defined wall function. A normal, high and low roughness were tested under neutral conditions using a frictional velocity u_* of 0.612 m s^{-1}. The inlet profiles are created using the neutral profile Equations 2.6, 2.67 and 2.68 for velocity and turbulence.

Table 4.2: Roughness lengths - Wall function test

	Roughness length z_0 [m]	Physical roughness height K_s [m]
Normal	0.002	0.0392
High	0.5	9.793
Low	0.0002	0.0039

The resulting profiles at 1000 m, 5000 m and 10000 m downstream from the inlet for velocity, turbulent kinetic energy and dissipation are shown graphically in Figures 4.4, 4.5 and 4.6, respectively. In each figure the right-side plot is a zoomed-in view of the left plot. Table 4.3 gives the absolute percentage error from the inlet profile calculated at 96.8 m AGL. This height corresponds closely with the typical wind turbine hub heights used on commercial wind farms.

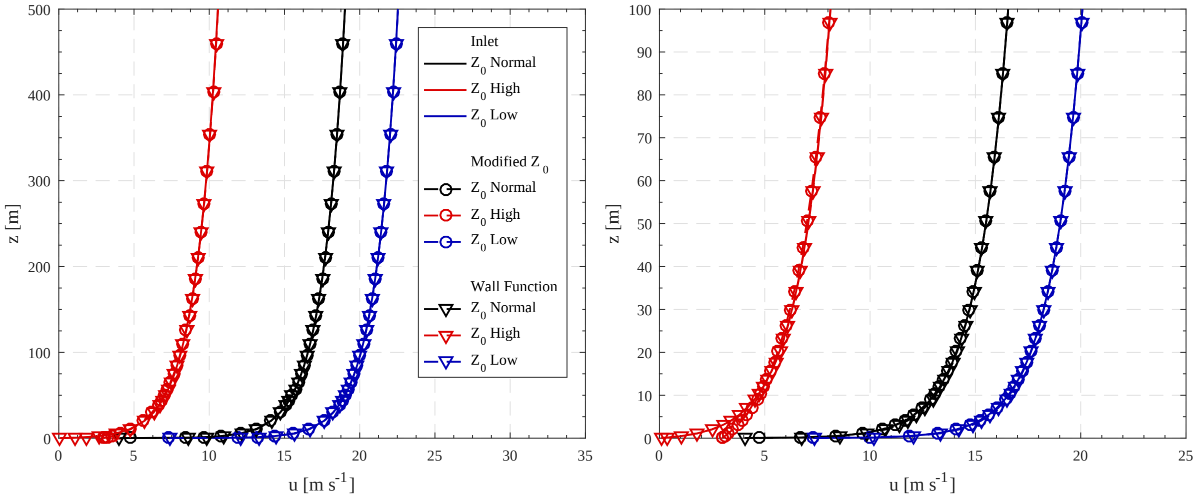
From the results it can be seen that for the normal and low roughness both methods perform very well with negligible errors even up to 10000 m. However, for the high roughness the modified approach breaks down completely with errors in excess of 10 % for turbulent kinetic energy and dissipation at 5000 m while the wall function method is less than 1 % from the inlet values. For velocity the modified roughness error is approximately double that of the modified wall function method.

The reason for the breakdown of the modified roughness approach in high roughness is due to the large physical roughness height that it requires which is larger than the first cell height. As described in Section 2.3.6 the first cell height should be greater than the roughness height to insure numerical fidelity. This breakdown in fidelity is evident in the k and ϵ high roughness profiles in Figures 4.5 and 4.6. It can be seen that close to ground level the modified roughness method's values are completely incorrect, either greatly over or under predicted. In the case of k the profile switches from large over to under prediction at 5000 m compared to 10000 m, this emphasizes the inability of the Fluent solver and mesh to deal with problem setup. The normal roughness length used equates to a physical roughness height in slight excess of the first cell height ($K_s = 0.0392 > 0.030$) however, the Fluent solver is able to deal with this inconsistency.

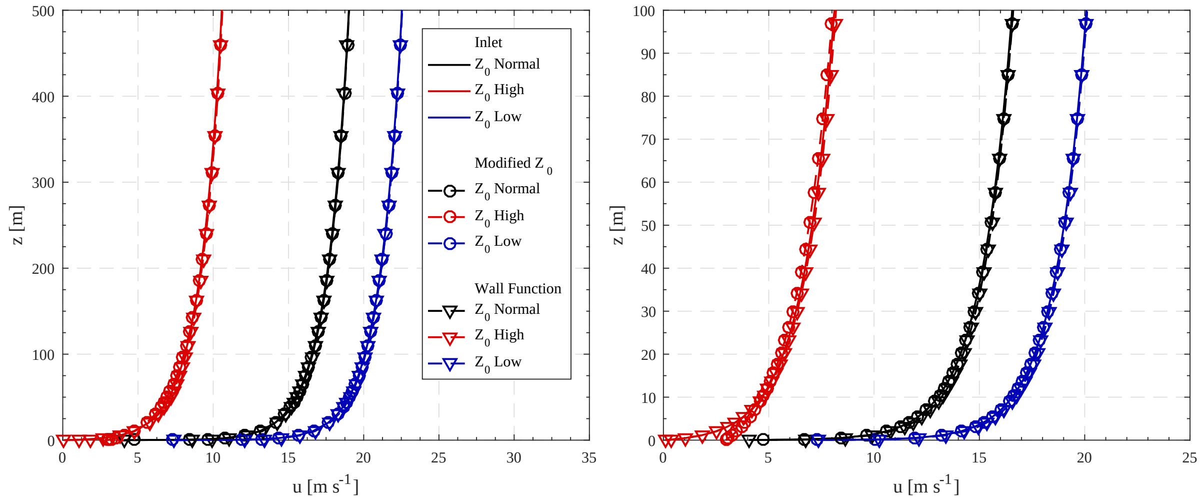
It can be concluded that if roughness lengths are present that would cause the physical roughness height to be sufficiently in excess of the first cell height the modified wall function approach should be used rather than the modified roughness approach.

Table 4.3: Percentage error at 96.8 m AGL - Wall function test

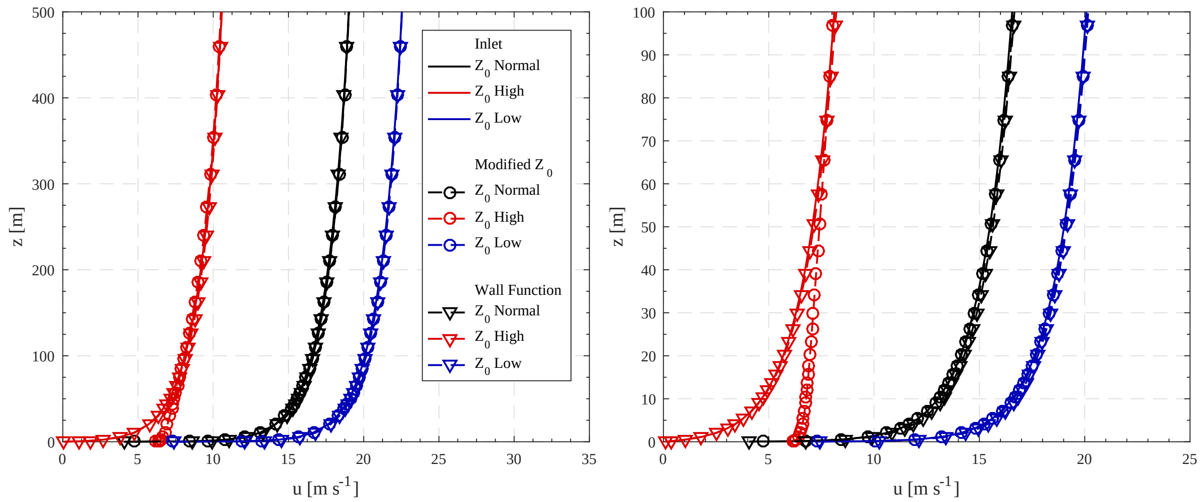
Velocity u [m s^{-1}]	1000 m	5000 m	10000 m
z_0 Normal - Mod Roughness	0.02	0.27	0.43
z_0 Normal - Wall Function	0.00	0.32	0.65
z_0 High - Mod Roughness	0.19	0.92	0.24
z_0 High - Wall Function	0.13	0.98	1.11
z_0 Low - Mod Roughness	0.01	0.20	0.36
z_0 Low - Wall Function	0.00	0.20	0.45
k [m^2 s^{-2}]	1000 m	5000 m	10000 m
z_0 Normal - Mod Roughness	0.05	0.44	0.10
z_0 Normal - Wall Function	0.08	0.73	2.44
z_0 High - Mod Roughness	2.04	10.45	2.63
z_0 High - Wall Function	0.07	0.19	3.68
z_0 Low - Mod Roughness	0.03	0.11	0.15
z_0 Low - Wall Function	0.05	0.41	1.61
ϵ [m^2 s^{-3}]	1000 m	5000 m	10000 m
z_0 Normal - Mod Roughness	0.48	1.65	2.12
z_0 Normal - Wall Function	0.38	2.24	4.29
z_0 High - Mod Roughness	4.74	19.56	13.92
z_0 High - Wall Function	0.41	0.99	1.21
z_0 Low - Mod Roughness	0.44	0.62	0.76
z_0 Low - Wall Function	0.41	1.36	3.25



(a) 1000 m

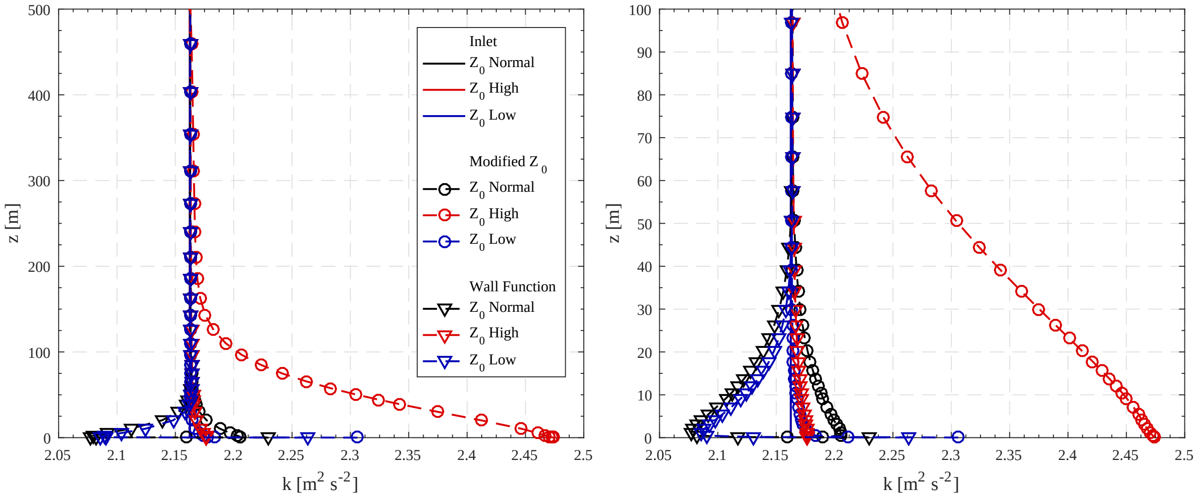


(b) 5000 m

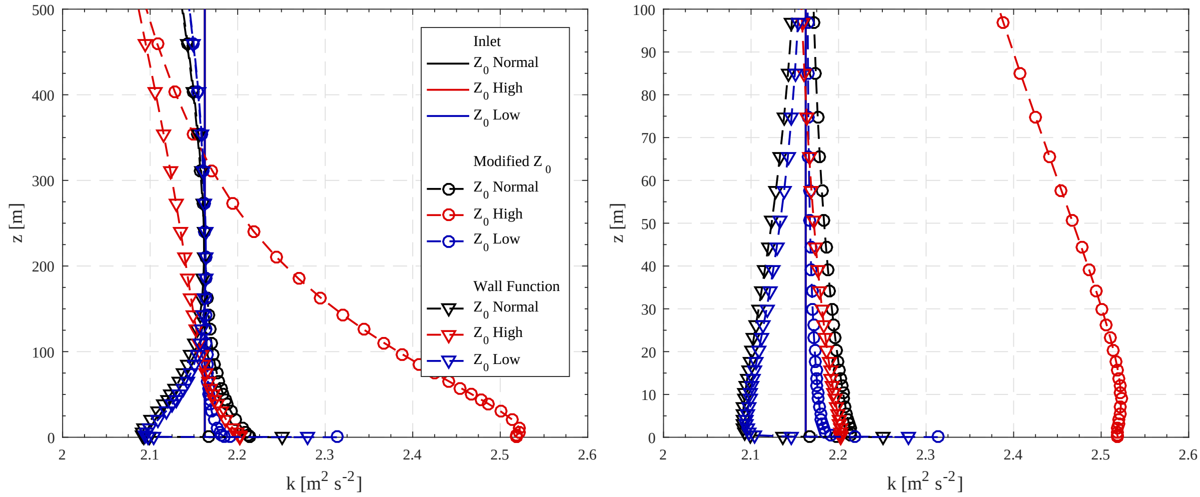


(c) 10000 m

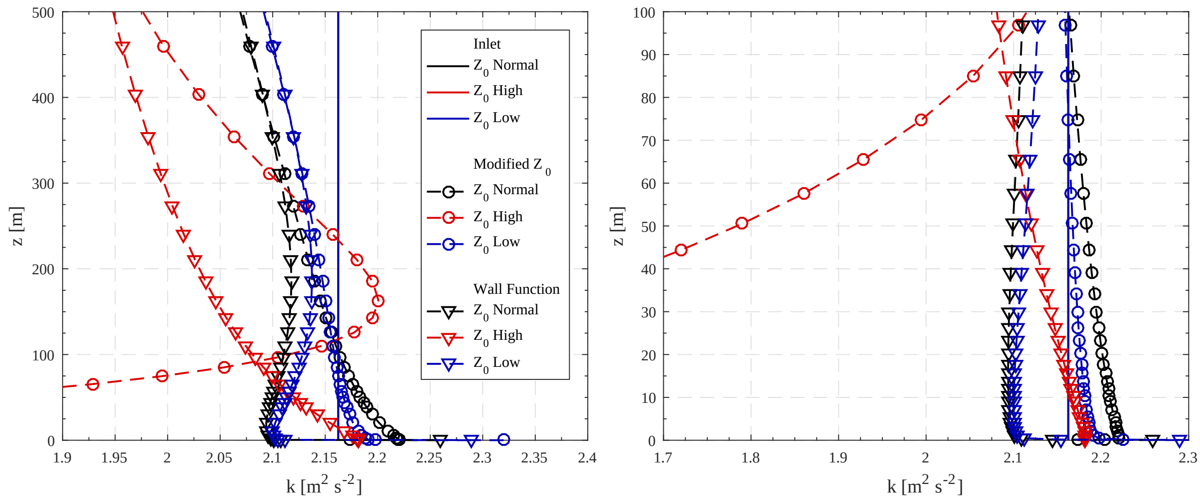
Figure 4.4: Wall function test results - Velocity



(a) 1000 m

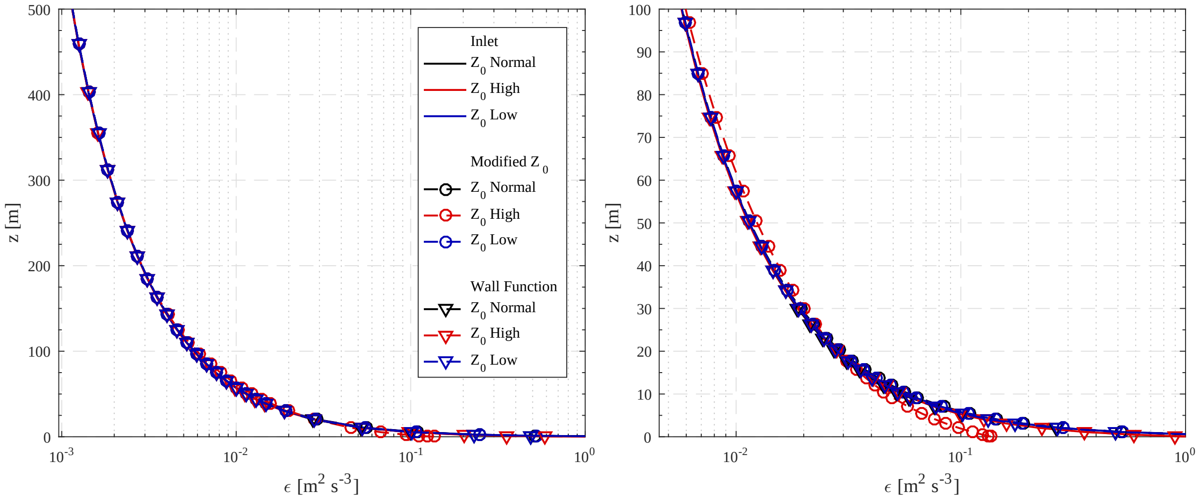


(b) 5000 m

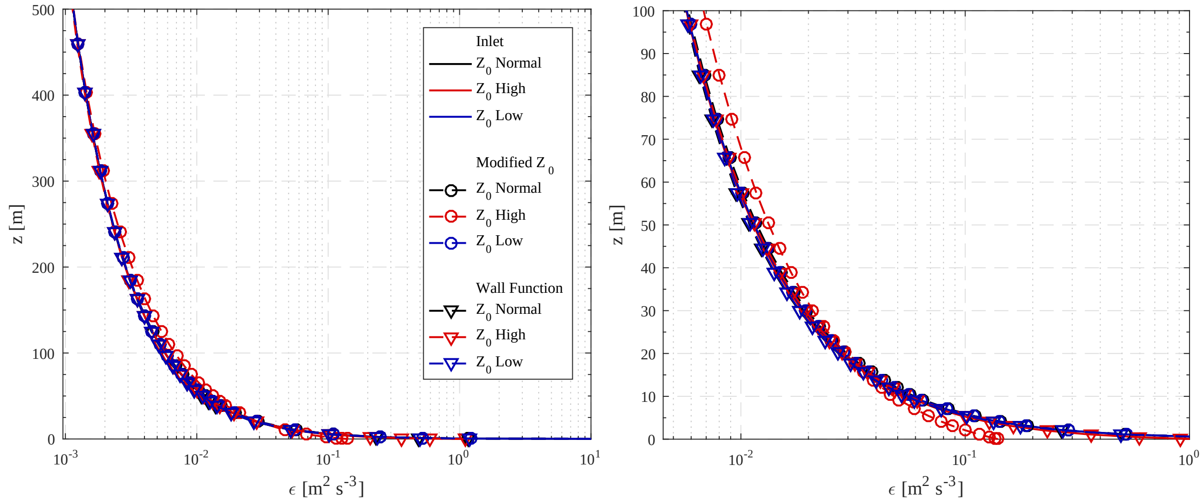


(c) 10000 m

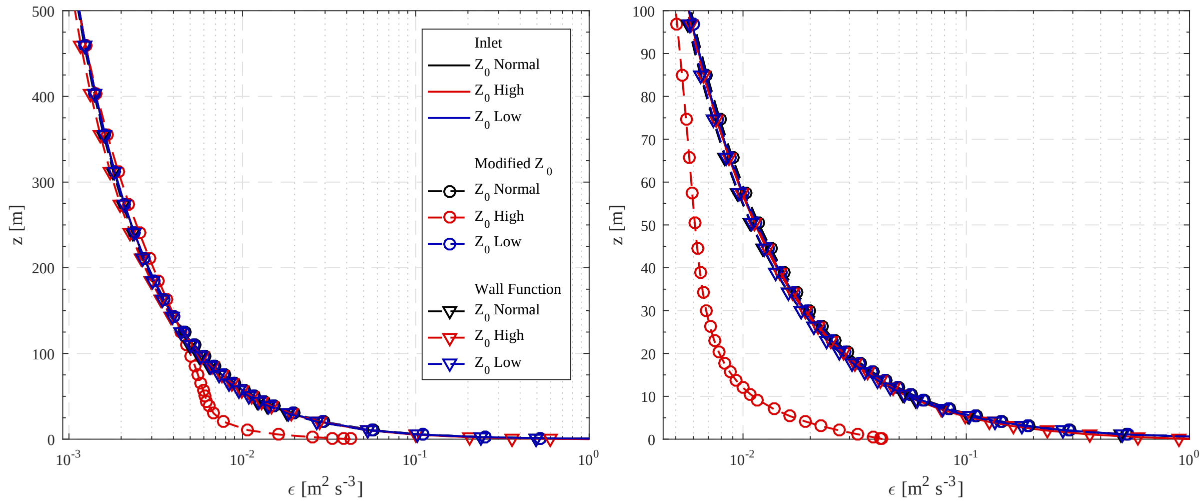
Figure 4.5: Wall function test results - k



(a) 1000 m



(b) 5000 m



(c) 10000 m

Figure 4.6: Wall function test results - ϵ

4.3.2 Stability model test results

The AM and DTU MOST stability models were tested using the four non-neutral stability conditions, the corresponding properties for MOL and frictional velocity are listed in Table 4.4. The AM and DTU models are introduced using the procedure described in Section 4.1.2. The results shown here are based on the G_{bMO} implementation and thus the energy equation is not included. A roughness length of 0.002 m is used. Based on the results of the wall function test and the fact that MOST profiles approach neutral conditions at the wall the modified wall function method is used for this section and the remainder of this study. The inlet profiles are created using the non-neutral profile Equations 2.18, 2.69 and 2.71 for velocity and turbulence.

Table 4.4: Model parameters - Stability model test

	MOL L [m]	Frictional Velocity u_* [m s^{-1}]
Extremely Unstable	-20.0	0.642
Unstable	-200.0	0.642
Stable	200.0	0.424
Extremely Stable	20.0	0.424

The resulting profiles at 1000 m, 5000 m and 10000 m downstream from the inlet for velocity, turbulent kinetic energy and dissipation are shown graphically in Figures 4.7, 4.8 and 4.9, respectively. In each figure the right-side plot is a zoomed-in view of the left plot. Table 4.5 gives the absolute percentage error from the inlet profile calculated at 96.8 m AGL.

The results show that for the velocity profiles the error induced at 5000 m is negligibly small ($< 1\%$). At 10000 m, however, there are increased errors for the two extreme conditions. For k and ϵ the same trend is seen: Analysing the DTU k error at 5000 m in unstable and stable conditions the error is 7.42% and 14.87% respectively. However, in the two extreme cases this error is increased in excess of 38%. The AM method shows close to double the percentage errors than the DTU method in stable and unstable conditions.

Comparing the turbulence values at 1000 m it is noted that both models have problems with the two extreme cases. The AM model shows difficulties in the extremely stable case as it has a 29.19% error. This can attributed to the fact that this model is only valid for $z/L < 2.0$ and using a MOL of 20 m this source is only valid up to 40 m AGL. The 10000 m velocity profiles in Figure 4.7c highlights the issues with both models in the extreme cases: In the extremely stable condition the AM velocity is artificially increasing close to ground and in the extremely unstable conditions the DTU profile has started to decelerate.

Graphically it can be seen that in extremely unstable conditions the profiles from both models lack the energy to sustain the high turbulence values and the profiles start to trail back compared to the inlet. In the extremely stable and stable case the AM model overshoots the $k - \epsilon$ profiles. Both methods suffer breakdowns at 10000 m.

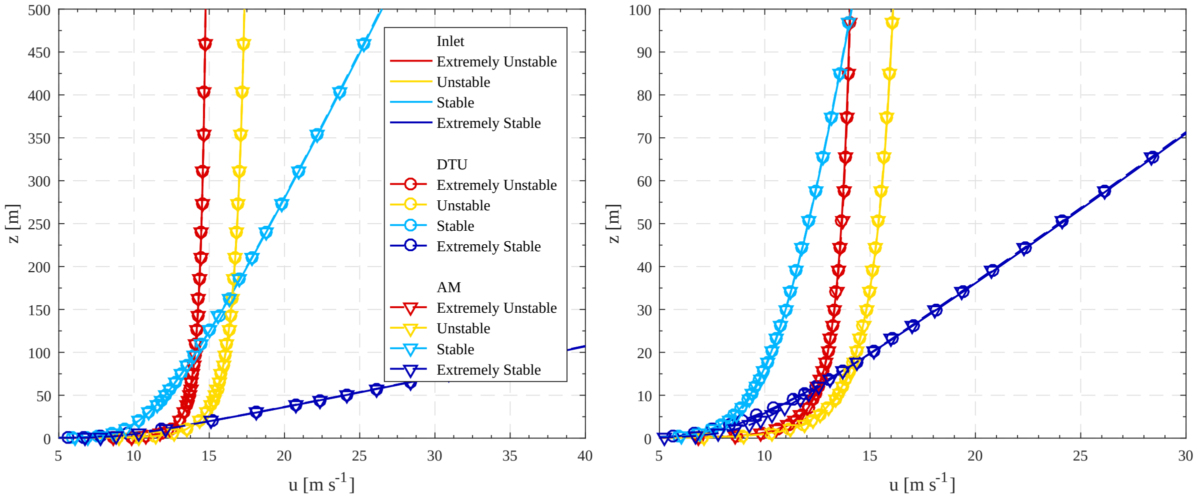
From the results it can be concluded care should be taken in the extreme cases, the models are presented in literature under standard non-neutral conditions (Omitting the extreme conditions) and their use in these cases are not well documented. The DTU model shows less error due the fact that the model is in balance for all values of z/L, including both extreme cases.

Under standard non-neutral conditions both models perform well with the DTU model showing less error. However, both models have trouble sustaining profiles over very large distances. ABL CFD models are known to be problematic in flat terrain [20]. For this reason care should be taken to not use excessively long upstream inlet distances. The low percentage error results at 1000 m and 5000 m indicates the models are suitable up to this range.

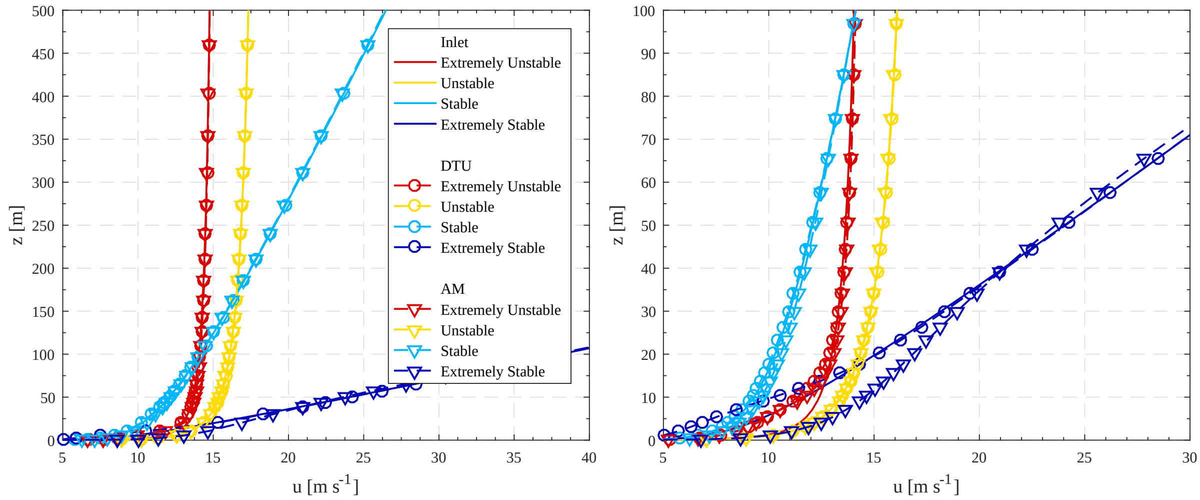
Using these results it can finally be concluded that the models can account for atmospheric stability and that horizontal homogeneity of the profiles can be obtained. However care should be taken in the two situations listed above.

Table 4.5: Percentage error at 96.8 m AGL - Stability model test

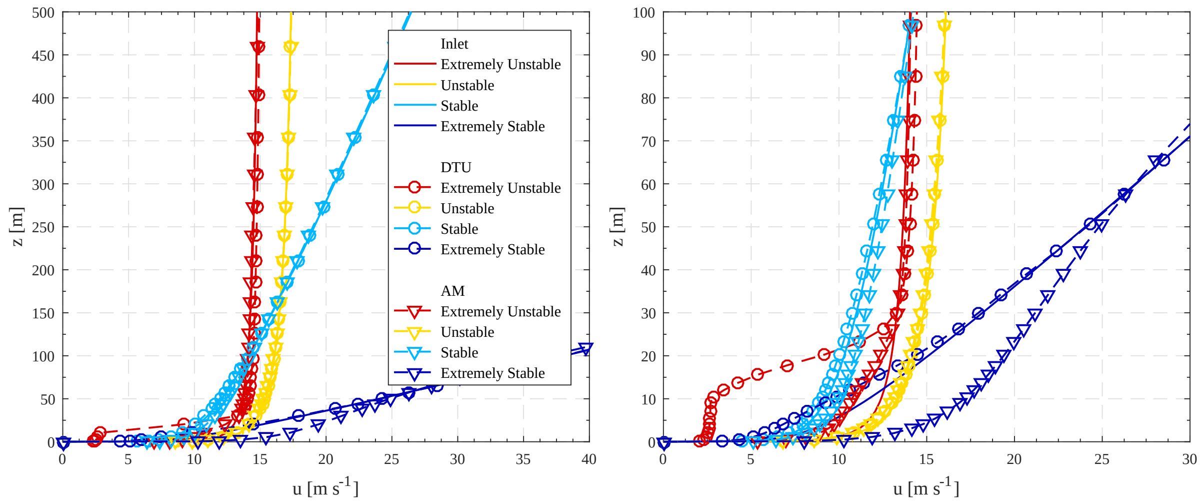
Velocity u [m s^{-1}]	1000 m	5000 m	10000 m
Extremely Unstable - DTU	0.03	0.48	2.83
Extremely Unstable - AM	0.12	0.58	0.30
Unstable - DTU	0.02	0.08	0.16
Unstable - AM	0.06	0.06	0.42
Stable - DTU	0.11	0.01	0.07
Stable - AM	0.14	0.01	0.98
Extremely Stable - DTU	0.30	0.19	0.25
Extremely Stable - AM	0.33	0.95	3.39
k [m^2 s^{-2}]	1000 m	5000 m	10000 m
Extremely Unstable - DTU	2.50	47.85	100.00
Extremely Unstable - AM	0.46	35.50	72.26
Unstable - DTU	0.47	7.42	27.78
Unstable - AM	1.91	12.76	34.16
Stable - DTU	1.42	14.87	26.97
Stable - AM	10.94	38.92	60.10
Extremely Stable - DTU	6.22	38.75	64.22
Extremely Stable - AM	29.19	159.21	520.18
ϵ [m^2 s^{-3}]	1000 m	5000 m	10000 m
Extremely Unstable - DTU	4.37	35.56	96.74
Extremely Unstable - AM	10.73	24.29	67.69
Unstable - DTU	2.02	2.37	17.05
Unstable - AM	1.76	6.22	21.88
Stable - DTU	1.63	10.38	20.63
Stable - AM	15.11	45.09	57.58
Extremely Stable - DTU	2.27	35.96	62.76
Extremely Stable - AM	41.20	192.82	591.95



(a) 1000 m

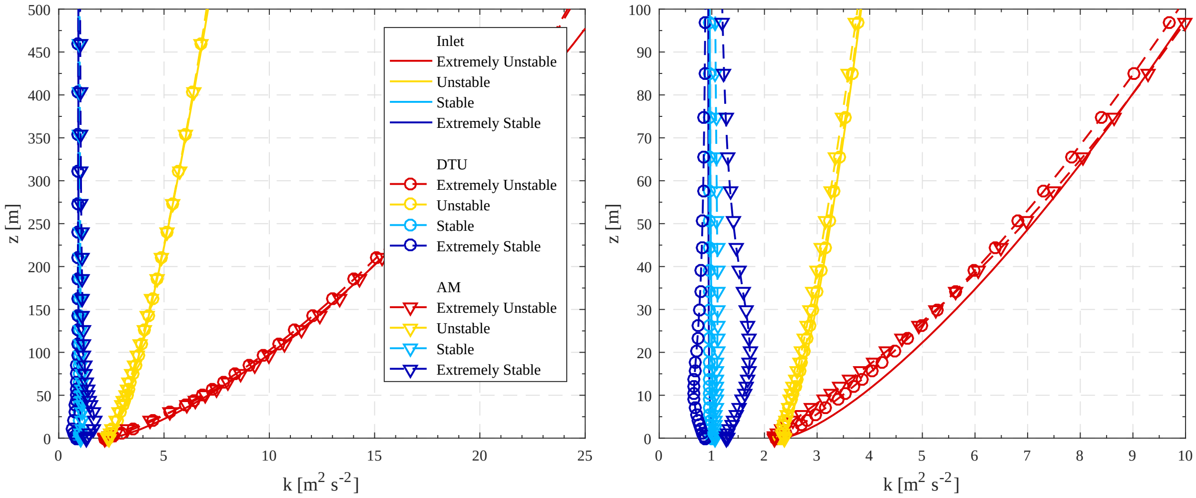


(b) 5000 m

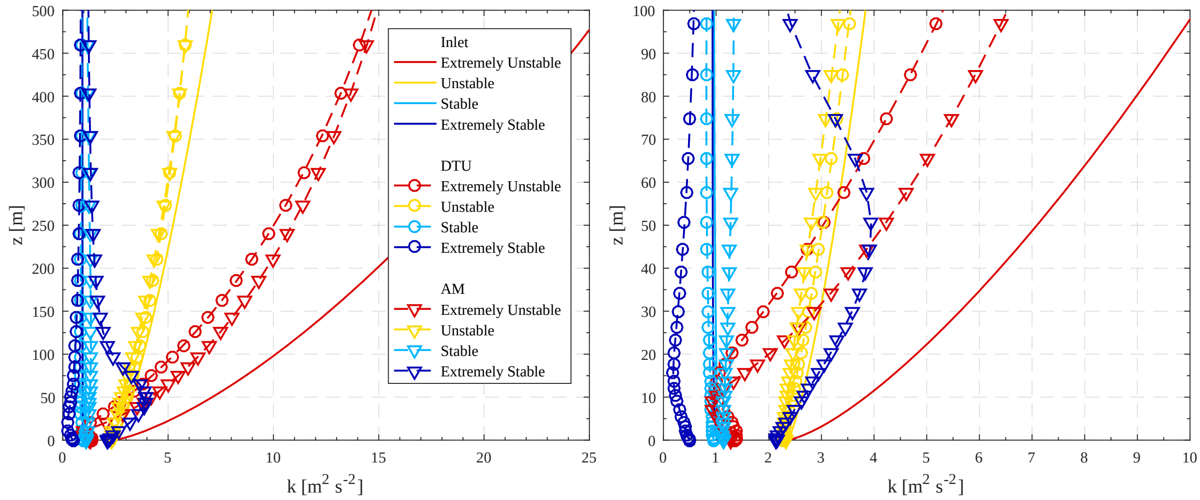


(c) 10000 m

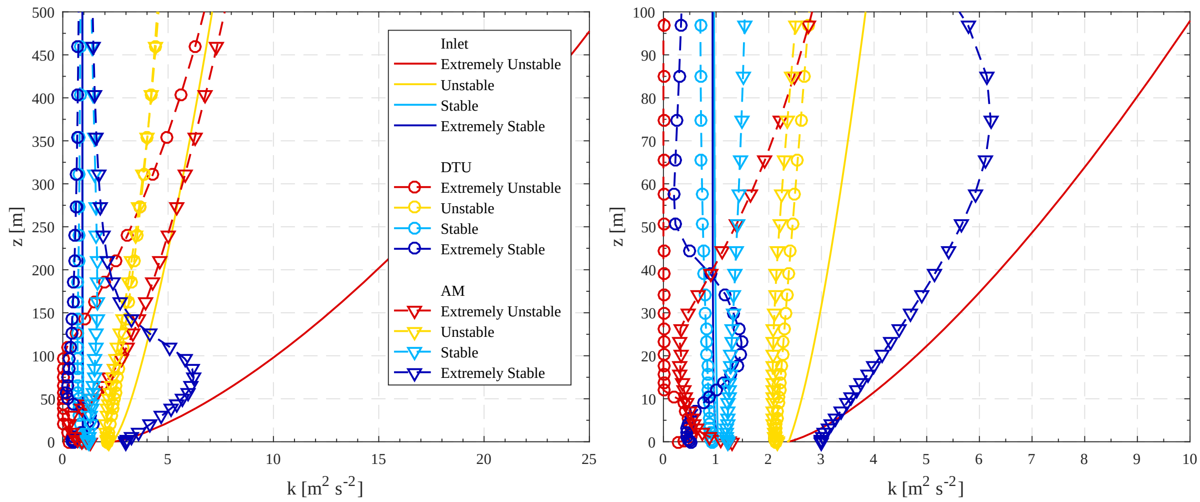
Figure 4.7: Stability model test results - Velocity



(a) 1000 m

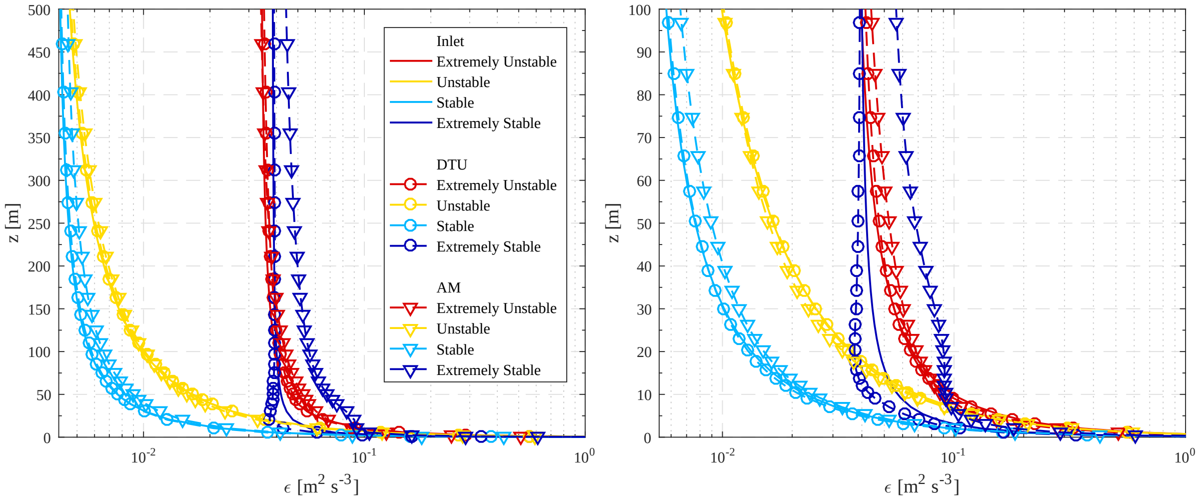


(b) 5000 m

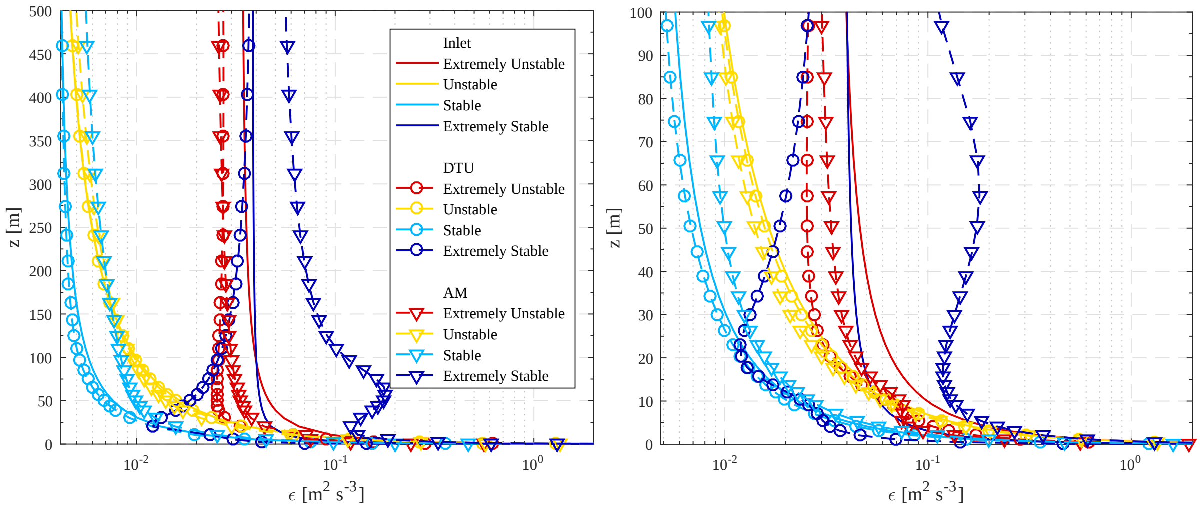


(c) 10000 m

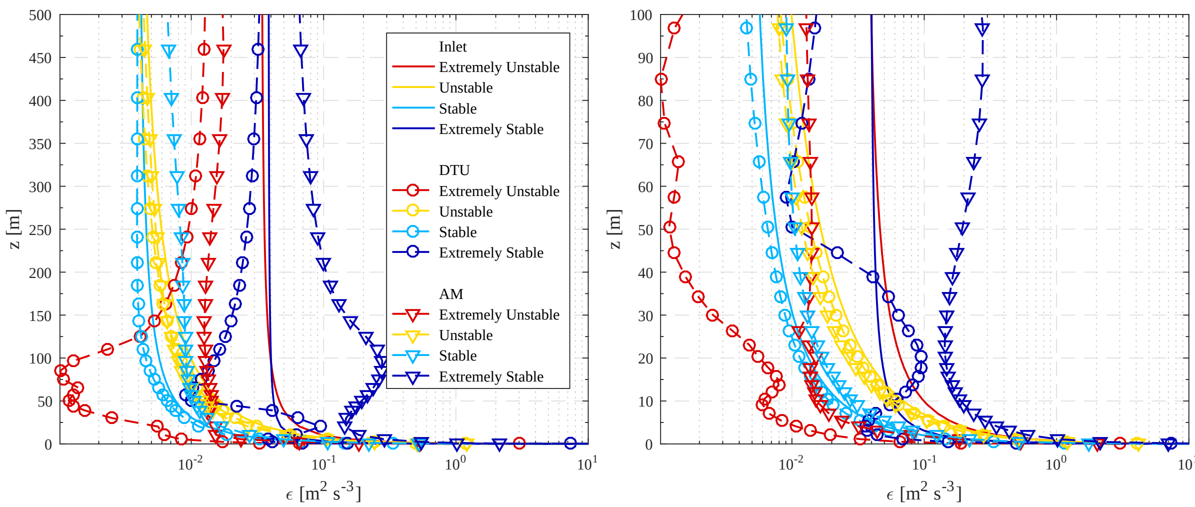
Figure 4.8: Stability model test results - k



(a) 1000 m



(b) 5000 m



(c) 10000 m

Figure 4.9: Stability model test results - ϵ

4.3.3 Buoyancy term test results

The standard G_b and MOST G_{bMO} were tested using the four non-neutral stability conditions, the corresponding properties for MOL, frictional velocity, ground temperature and potential temperature scale are listed in Table 4.6. The buoyancy term is tested with the AM model and introduced using the procedure described in Sections 4.1.2 and 4.1.3. The energy equation and the Boussinesq buoyancy approximation are used when evaluating the standard buoyancy term. A roughness length of 0.002 m is used with the modified wall-function method. The inlet profiles are created using the non-neutral profile Equations 2.18, 2.19, 2.69 and 2.71 for velocity, potential temperature and turbulence.

Table 4.6: Model parameters - Buoyancy term test

	MOL L [m]	Frictional Velocity u_* [m s^{-1}]	Ground Temp. T_0 [k]	Temp. Scale θ_* [k]
Extremely Unstable	-20.0	0.642	303.0	-0.108
Unstable	-200.0	0.642	303.0	-0.108
Stable	200.0	0.424	288.0	0.0232
Extremely Stable	20.0	0.424	288.0	0.0232

The resulting profiles at 1000 m, 5000 m and 10000 m downstream from the inlet for velocity, turbulent kinetic energy and dissipation are shown graphically in Figures 4.10, 4.11 and 4.12, respectively. In each figure the right-side plot is a zoomed-in view of the left plot. Table 4.7 gives the absolute percentage error from the inlet profile calculated at 96.8 m AGL.

From the results it is clear that under the extreme conditions the standard G_b formulation immediately breaks down with errors in excess of 100% for the turbulence quantities at 1000 m. This can be attributed to the fact that in these cases large heat fluxes and temperature gradients are present and obtaining an accurate steady-state solution is very difficult [7]. The values presented here are thus of little value and a transient simulation will be needed to deal with the unsteady convection physics that are at work. Under the standard not non-neutral conditions the effects are less intense however still present with the standard G_b formulation subject to excessively large errors.

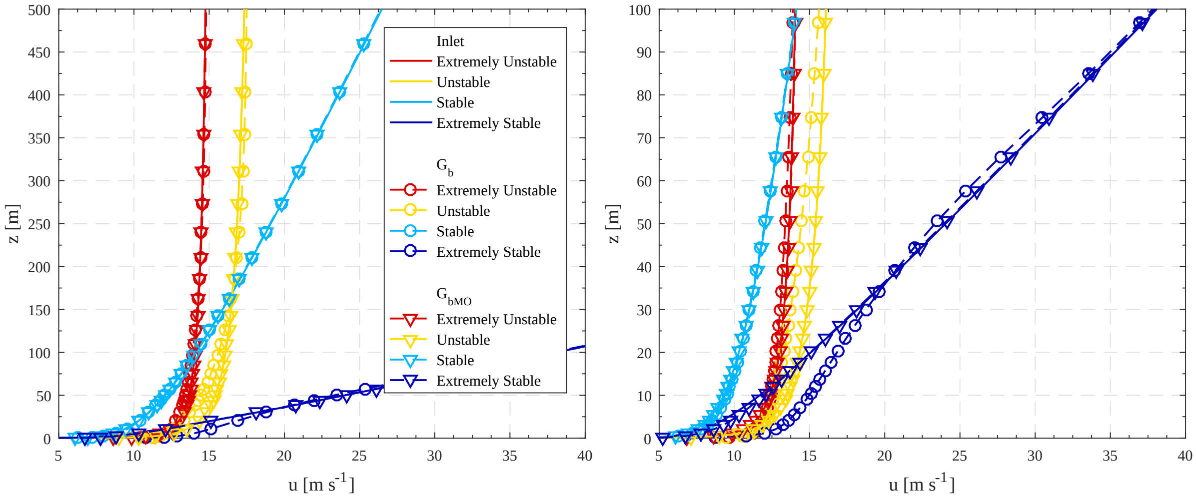
Graphically the issue can be seen in the velocity plots from Figure 4.10c where in the extremely stable case the velocity for G_b completely collapses and predicts overly large velocities. In the extremely unstable case Figure 4.11c the turbulent kinetic energy is also greatly over-predicted.

Using the MOST G_{bMO} formulation the stratification effects on the momentum equation is not directly present due the energy equation being neglected, however the low errors in the velocity profile results shows that the effects are negligible. This can be attributed to the fact that the turbulence source terms augment/suppress the turbulent quantities in the $k - \epsilon$ transport equations. Thus these effects are included in the turbulent eddy viscosity which is used in RANS momentum equation as explained in Sections 2.2.1 and 2.2.2.

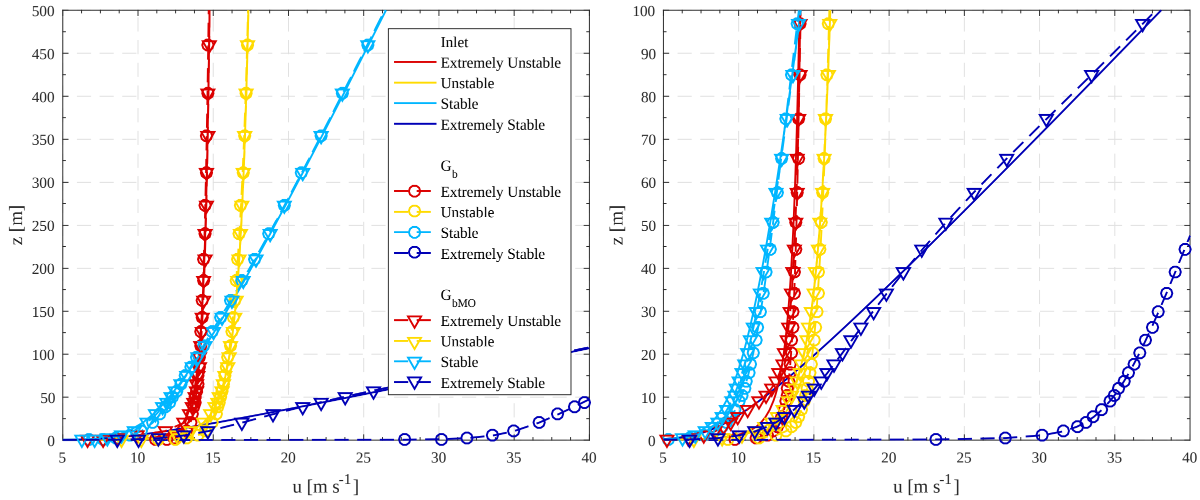
It can be concluded that the MOST G_{bMO} formulation produces more accurate results and for ABL CFD models the standard G_b formulation along with the energy equation and the Boussinesq buoyancy approximation is incompatible with steady-state simulations. Further research is therefore required into transient ABL CFD models.

Table 4.7: Percentage error at 96.8 m AGL - Buoyancy term test

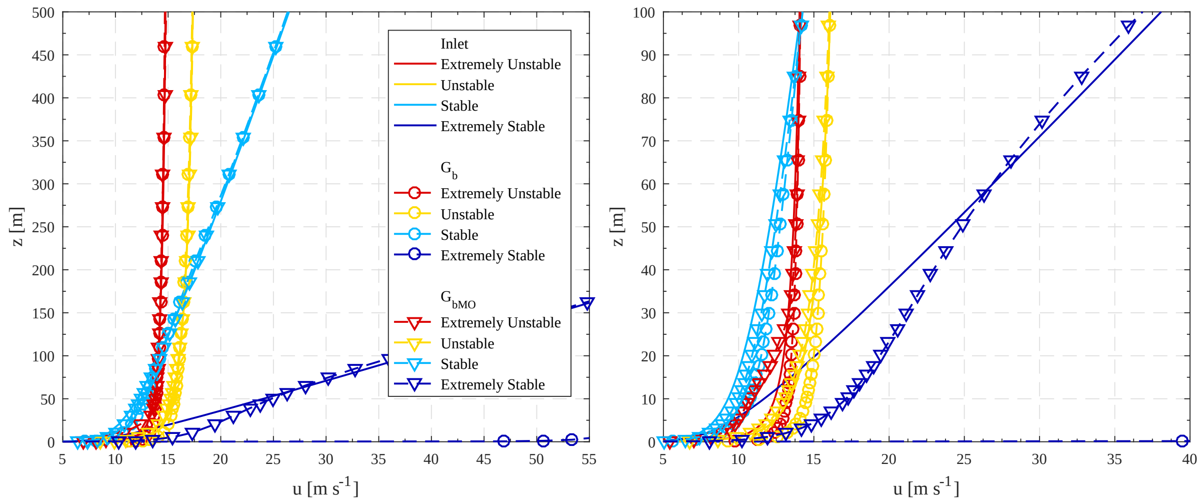
Velocity u [m s^{-1}]	1000 m	5000 m	10000 m
Extremely Unstable - G_b	1.15	0.39	0.48
Extremely Unstable - G_{bMO}	0.12	0.58	0.30
Unstable - G_b	3.08	0.37	0.04
Unstable - G_{bMO}	0.06	0.06	0.42
Stable - G_b	0.22	0.85	1.00
Stable - G_{bMO}	0.14	0.01	0.98
Extremely Stable - G_b	0.73	23.44	85.43
Extremely Stable - G_{bMO}	0.33	0.95	3.39
k [m^2 s^{-2}]	1000 m	5000 m	10000 m
Extremely Unstable - G_b	487.21	224.20	219.90
Extremely Unstable - G_{bMO}	0.46	35.50	72.26
Unstable - G_b	973.65	292.50	278.38
Unstable - G_{bMO}	1.91	12.76	34.16
Stable - G_b	20.25	91.36	124.90
Stable - G_{bMO}	10.94	38.92	60.10
Extremely Stable - G_b	245.81	16015.37	47797.47
Extremely Stable - G_{bMO}	29.19	159.21	520.18
ϵ [m^2 s^{-3}]	1000 m	5000 m	10000 m
Extremely Unstable - G_b	274.07	132.48	127.05
Extremely Unstable - G_{bMO}	10.73	24.29	67.69
Unstable - G_b	935.17	209.71	187.00
Unstable - G_{bMO}	1.76	6.22	21.88
Stable - G_b	28.05	93.15	100.35
Stable - G_{bMO}	15.11	45.09	57.58
Extremely Stable - G_b	117.04	6746.45	22905.36
Extremely Stable - G_{bMO}	41.20	192.82	591.95



(a) 1000 m

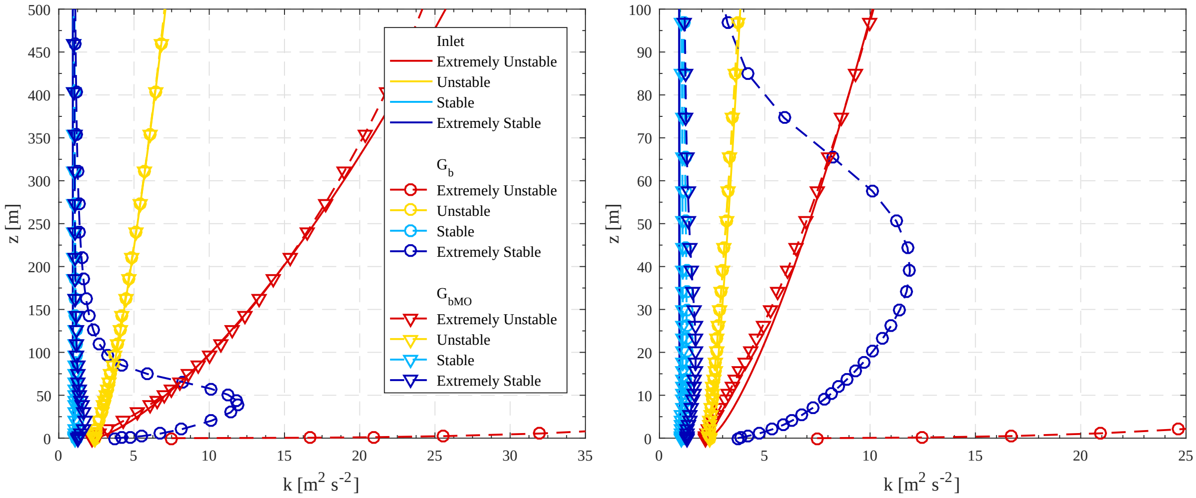


(b) 5000 m

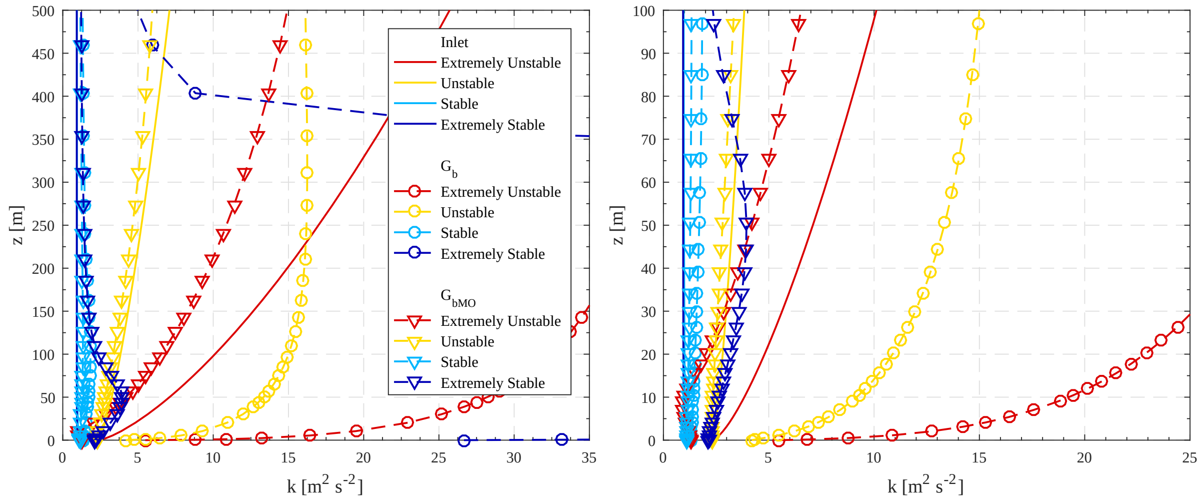


(c) 10000 m

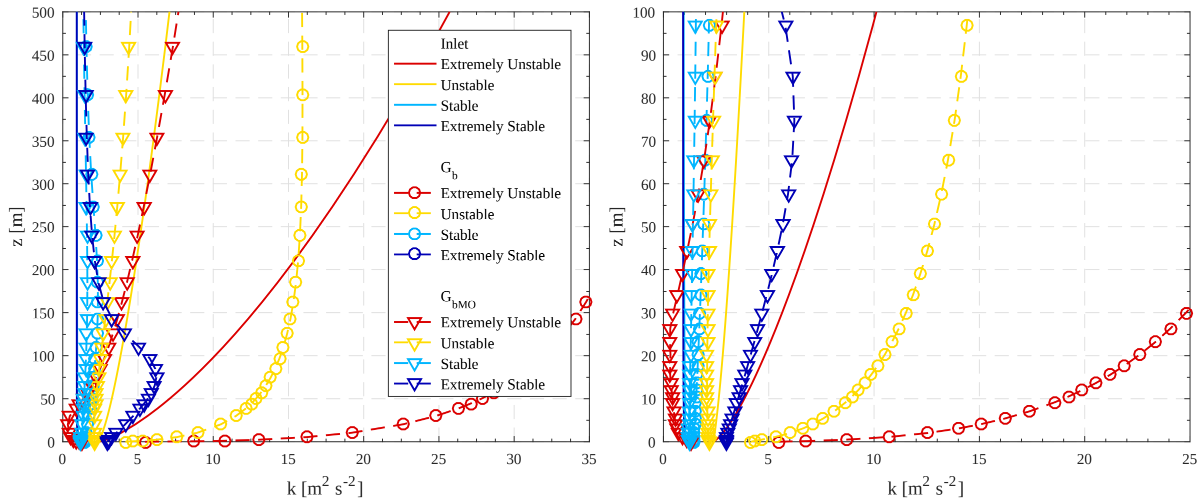
Figure 4.10: Buoyancy term results - Velocity



(a) 1000 m

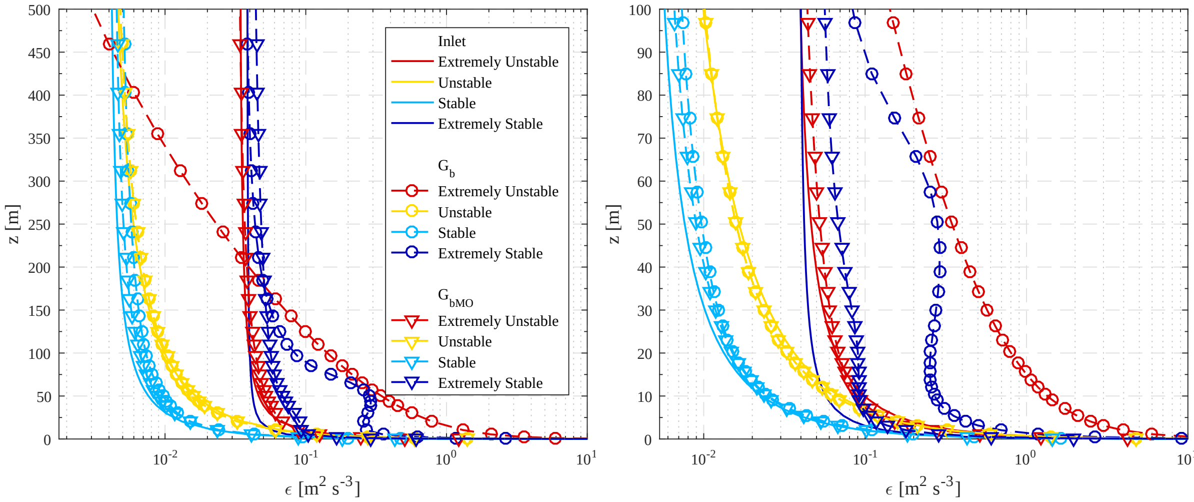


(b) 5000 m

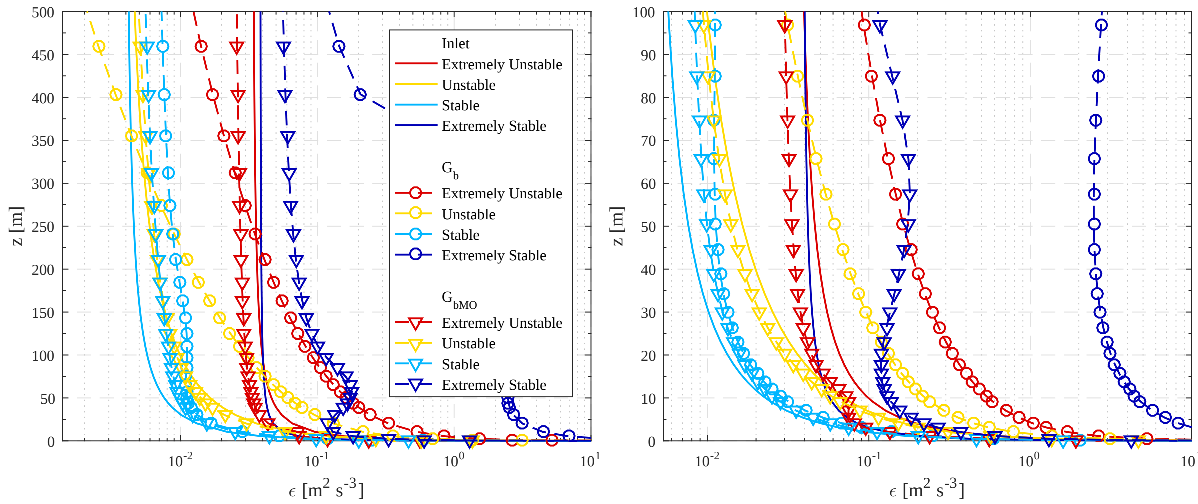


(c) 10000 m

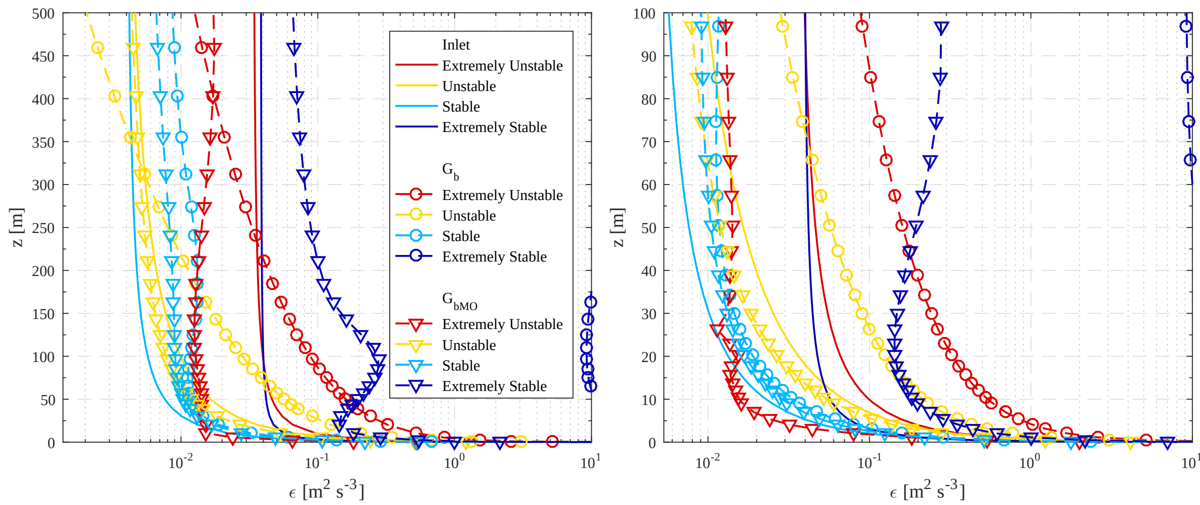
Figure 4.11: Buoyancy term results - k



(a) 1000 m



(b) 5000 m



(c) 10000 m

Figure 4.12: Buoyancy term results - ϵ

4.4 Summary

From the results it can be concluded that the inclusion of the required inlet profiles, sources and wall functions using UDFs are implemented correctly. They are identically implemented in the complex terrain CFD model in Chapter 5.

The results of the wall-function test showed that both methods can be accuratly used for neutral profiles. However, the modified roughness approach breaks down under high roughness. For this reason the modified wall-function method is preferred.

The results from the stability model test highlighted that both models have issues with the extreme conditions as well as maintaining profiles over distances greater than 5000 m. For this reason care should be taken for the extreme conditions and upstream inlet distances should be minimized. The DTU model also proved to be more accurate in maintaining the profiles, however the results prove non-definitive and both models are evaluated in the complex terrain CFD model to follow.

Finally, the buoyancy term test highlighted the issues of including thermal effects in steady CFD simulations and the MOST G_{bMO} formulation is more accurately able to account for the stratification effects in the turbulence equations. The MOST formulation also showed the ability to accurately include the stratification effects on the velocity profiles without the need for additional buoyant momentum sources. For these reasons the G_{bMO} formulation is preferred and used in the complex terrain CFD model.

Chapter 5

CFD Simulation of Complex terrain

The developed ABL CFD model from Chapter 4 along with the stability and site data from Chapter 3 are used to test and validate the model in a complex terrain. The model is validated by using two onsite masts to cross-predict the velocity profiles via transfer functions developed using the CFD results. Based on the conclusions from Chapter 4 the model uses the modified wall-function approach and the MOST buoyancy term formulation. The AM and DTU models are both evaluated.

5.1 Wind Farm Computational Domain

The complex terrain CFD model uses the same setup, settings and coordinate system as the empty domain model as described in Section 4.2. A schematic of the computational domain is shown in Figure 5.1.

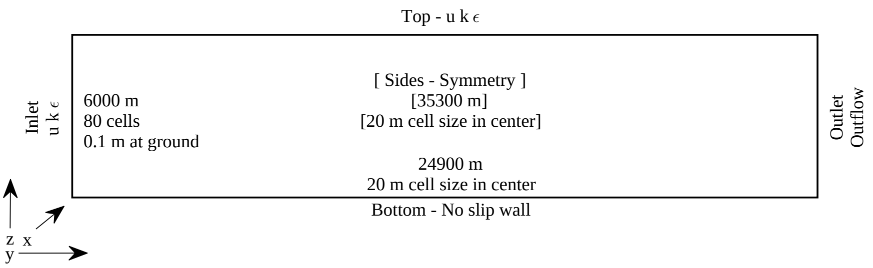


Figure 5.1: Computational Domain - Complex terrain
Square brackets indicate properties along the x dimension

The domain is rectangular cuboid with dimensions of 35300 m, 24900 m and 6000 m in x, y and z respectively. Using the wind farm contour data the domain is discretized via a block-structured double-O grid using the Ansys ICEM CFD mesher. The mesh block structure is shown in Figure 5.2. In the inner O grid the cell size is fixed to 20 m. This covers the entire hill feature plus a 500 m boundary. The next block is located 3000 m from this boundary. In this block the cells expand in size from 20 m to a maximum of 50 m using a geometric growth ratio of 1.05. In the outer O grid the cells expand in size from 50 m to a maximum of 100 m using a geometric growth ratio of 1.1. The cells at the edges of the domain thus have a size of 100 m. The z-direction is discretized using 80 vertical cells with a ground cell height of 0.1 m and a geometric growth ratio of 1.1. The complete mesh comprises of 24966291 cells. The meshing procedure and details are in accordance to generally accepted industry standards and are known to produce reliable and mesh independent results.

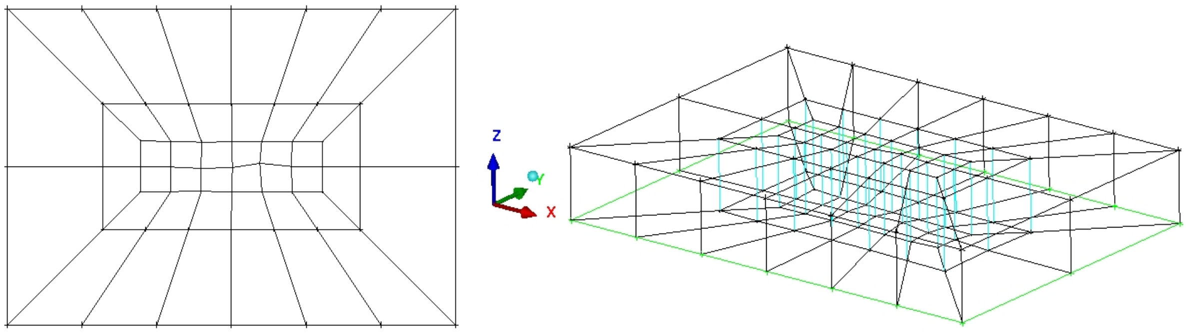


Figure 5.2: Mesh block structure used to discretized wind farm terrain model

The wind farm terrain model and the mesh on the South and West faces are shown in Figure 5.3. The pink and black spheres respectively shows the locations of Mast 1 and 2. It can be noted that an artificial smoothing is applied around the terrain model. This is used so that the inlet profiles can be applied on a completely flat terrain and removes the possibility of having terrain features present along any of the boundaries causing problems with the symmetry and outflow boundary conditions. The terrain is smoothed to the mean normal height ASL at the edge of the terrain model. The smoothed section thus serves as the upstream inlet and has a length of 2500 m. The stability models demonstrated the ability of maintaining profiles up to 5000 m in Section 4.3.2.

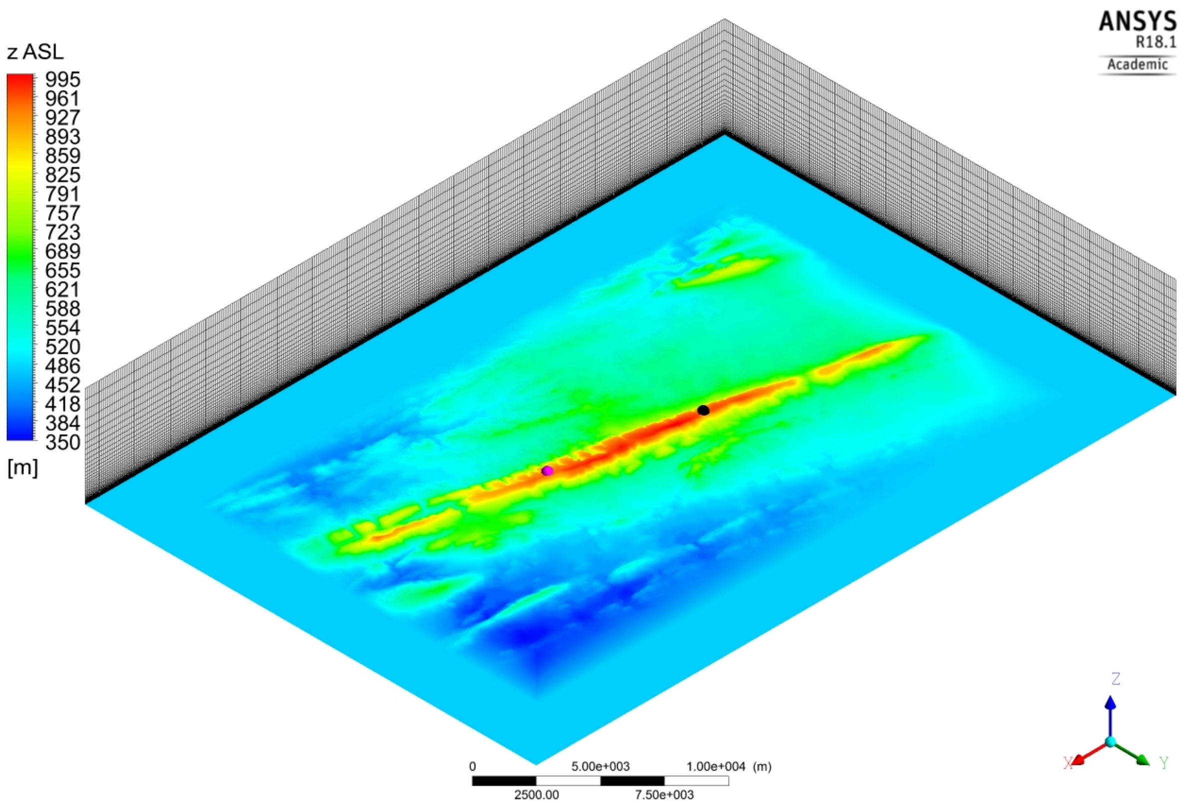


Figure 5.3: Wind farm terrain model coloured using height above sea level and indicating mesh density on South and West faces

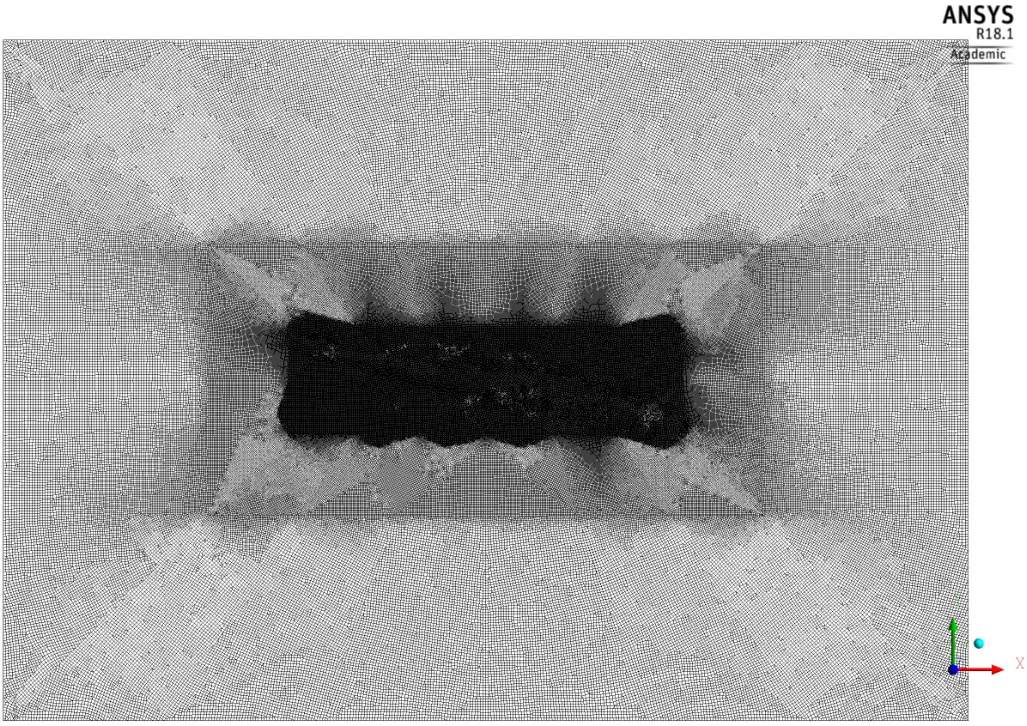


Figure 5.4: Top view - Wind farm mesh

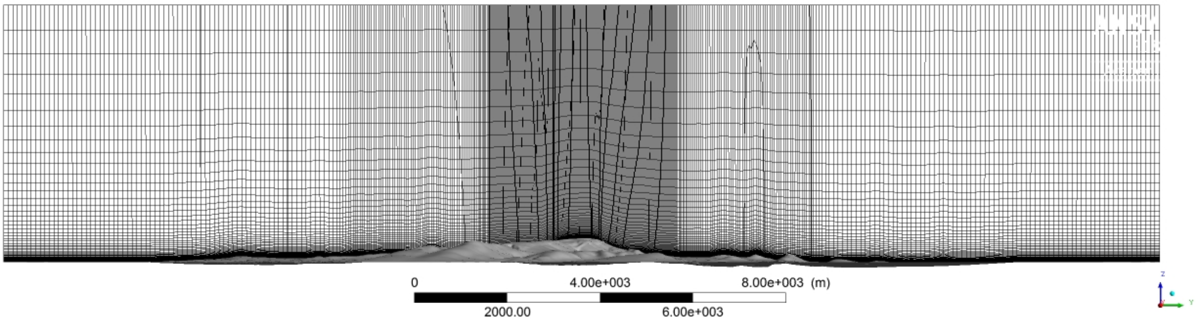


Figure 5.5: Easterly view - Wind farm mesh

The effect of the block-structured refinement is illustrated in Figure 5.4. The three distinct O grid regions can be seen with the 20 m cells shown in the dark central block. In Figure 5.5 the z refinement can be seen from the Easterly view that highlights the growth in cell size from the bottom to the top of the domain and the central refined grid.

The mesh's ability to accurately capture the small ravines and undulating terrain around the main hill is shown by the ground level mesh in Figures 5.6 and 5.7. Both masts are highlighted by the coloured spheres. It can be noted that at both mast locations the hill is not perfectly sinusoidal or smooth but instead there are ravines leading up the hill. These varying features cause differences in the measured profiles as well as the CFD results at the mast locations. These differences cause changes in the wind profiles experienced at the mast locations. Using these differences between Mast 1 and Mast 2 it is possible to construct a transfer function based on the CFD results that allows the wind profile prediction at Mast 2 using the measured data from Mast 1.

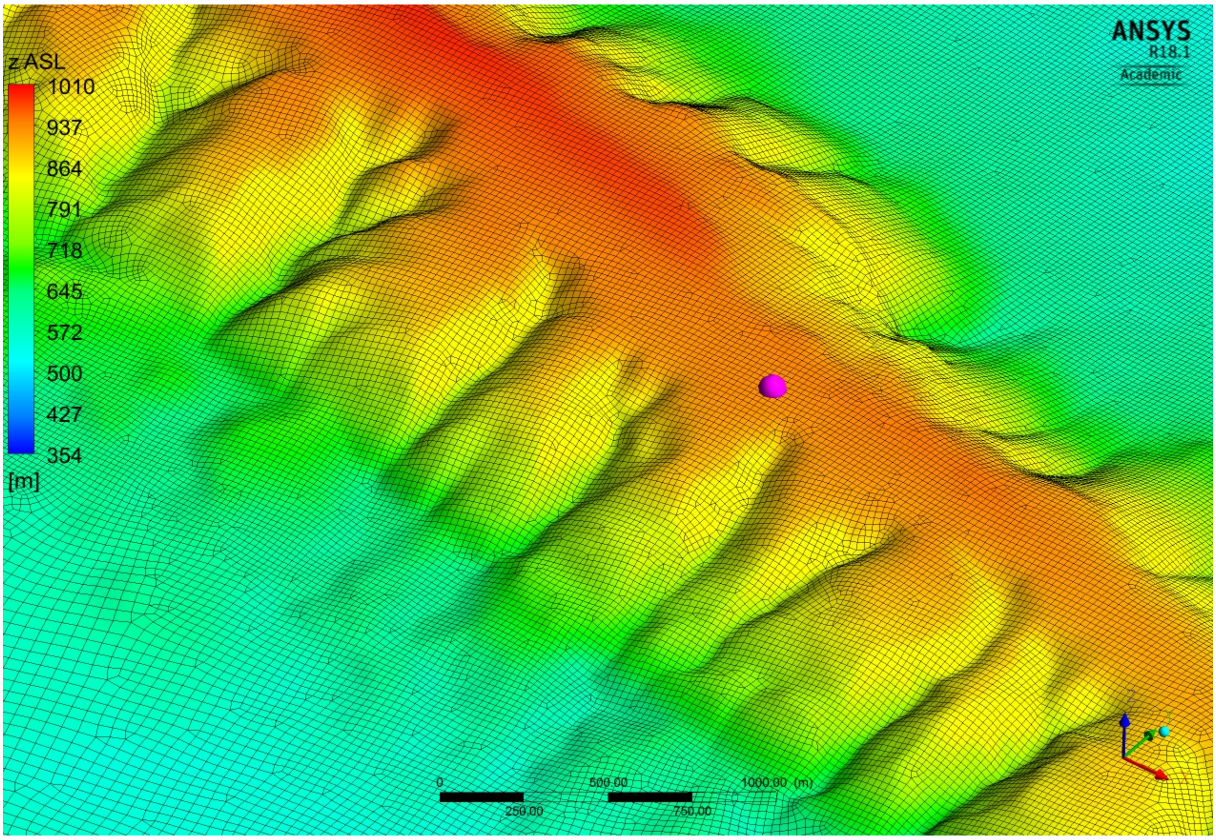


Figure 5.6: Terrain mesh at Mast 1 - Coloured using height above sea level

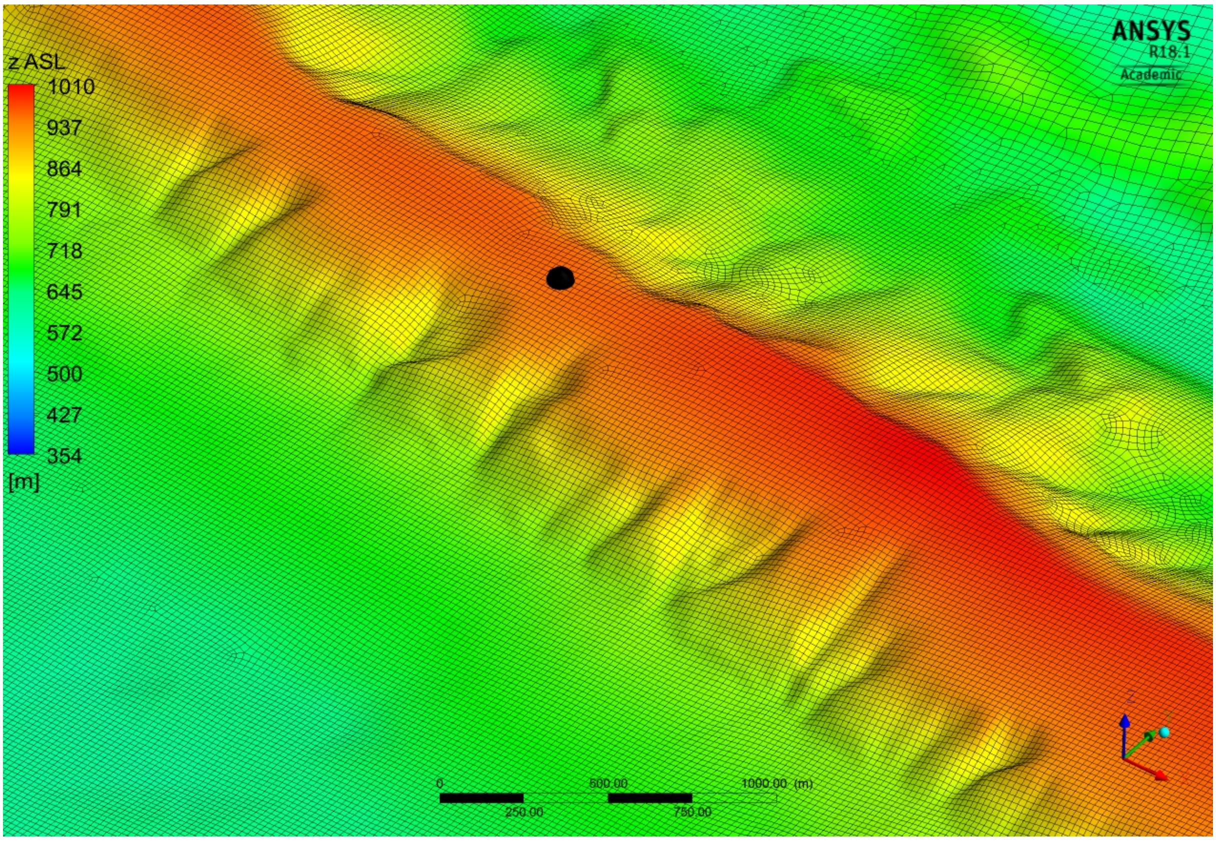


Figure 5.7: Terrain mesh at Mast 2 - Coloured using height above sea level

5.2 Windfarm Model Setup

The solver settings as well as the procedure for setting up the required inlet profiles, source terms and wall function using UDFs are repeated from Section 4.2. The x and y momentum source terms for the Coriolis force are now included because of the increased size of the domain and the need to capture all of the onsite physical processes. The inlet profiles are obtained by applying the data analysis procedure described in Chapter 3 to a WRF mesoscale data set obtained at the inlet location. The site MOL and site air density are obtained from the measured data at Mast 1. The main model-input data are given in Table 5.1. A linear interpolation function is employed to determine the MOL used in the source terms. The function interpolates from the MOL obtained using the WRF data at the inlet location to the MOL calculated from the measured data at Mast 1. This allows the inlet profiles to be maintained along the upwind fetch by their accompanying MOL and at the hill the actual measured onsite MOL is used. The simulations are considered converged when the residuals level out, resulting in a decrease of at least five orders of magnitude.

Table 5.1: Windfarm CFD model input data

	Inlet - Mesoscale		Mast 1 - Measured	
	MOL L [m]	Frictional Velocity u_* [m s^{-1}]	MOL L [m]	Air density ρ [kg m^{-3}]
Extremely Unstable	-9.0	0.373	-5.8	1.082
Unstable	-254.6	0.374	-231.0	1.082
Neutral	N/A	0.144	N/A	1.101
Stable	124.7	0.141	221.8	1.097
Extremely Stable	21.4	0.065	26.3	1.103

5.3 Mast Velocity Cross-Prediction Results

Using the CFD results three transfer functions are created from the velocity magnitude at 40 m, 60 m and 82 m AGL at both mast locations. These heights correspond to the measurement heights of the masts. The velocity transfer function Γ is defined as

$$\Gamma = \frac{u_{M2\ \mathrm{CFD}}}{u_{M1\ \mathrm{CFD}}} \tag{5.1}$$

where M1 and M2 denote Mast 1 and 2. Using the transfer function it is the possible to obtain the predicted velocity at Mast 2 using Equation 5.2.

$$u_{M2\ \mathrm{Predict}} = \Gamma\, u_{M1\ \mathrm{CFD}} \tag{5.2}$$

The percentage cross-prediction error is then calculated using Equation 5.3

$$\mathrm{Error} = 100 \times \frac{|u_{M2\ \mathrm{Measured}} - u_{M2\ \mathrm{Predict}}|}{u_{M2\ \mathrm{Measured}}} \tag{5.3}$$

The prediction results at 82 m AGL for both models are given in Table 5.2. The measured vs. predicted velocity profiles are shown in Figure 5.8. The crosses indicate the mean measured velocity from Mast 2 and the solid line is the velocity profile fit for these points. The circles and triangles are the predicted velocities using Equation 5.2.

Table 5.2: Mast 2 cross prediction results at 82 m

	Extremely Unstable	Unstable	Neutral	Stable	Extremely Stable
$u_{M2 \text{ Measured}}$ [m s^{-1}]	7.33	7.69	6.85	5.88	2.78
$u_{M2 \text{ Predict}}$ DTU [m s^{-1}]	7.34	8.92	8.80 [1]	5.32	2.53
$u_{M2 \text{ Predict}}$ AM [m s^{-1}]	7.28	9.09		5.49	2.24
Prediction Error DTU [%]	0.08	15.97	28.50 [2]	9.49	9.10
Prediction Error AM [%]	0.74	18.17		6.74	19.36

[1] Using the neutral model - $u_{M2 \text{ Predict}}$ Neutral [m s^{-1}]
[2] Using the neutral model - Error Neutral [%]

The cross-prediction results show that both models were able to accurately capture the two main stability conditions onsite. The model results give an error of less than 1% in the extremely unstable condition, as discussed in Chapter 3 this condition is present on-site 36 % of the time. The most dominating condition is the stable condition which is present 40% of the time. In this condition both models have errors of less than 10%. In the extremely stable condition at 82 m the DTU model outperformed the AM model by 10%. The profiles in Figure 5.8 illustrate this as one of the shortcomings of the AM model. As discussed previously in stable conditions this model is only valid for $z/L < 2$ and using the mast MOL of 21.4 m this model loses validity for heights greater than 42.8 m. This can be seen in the profiles by noting the small error at 40 m extremely stable compared to the increased error it exhibits at 82 m.

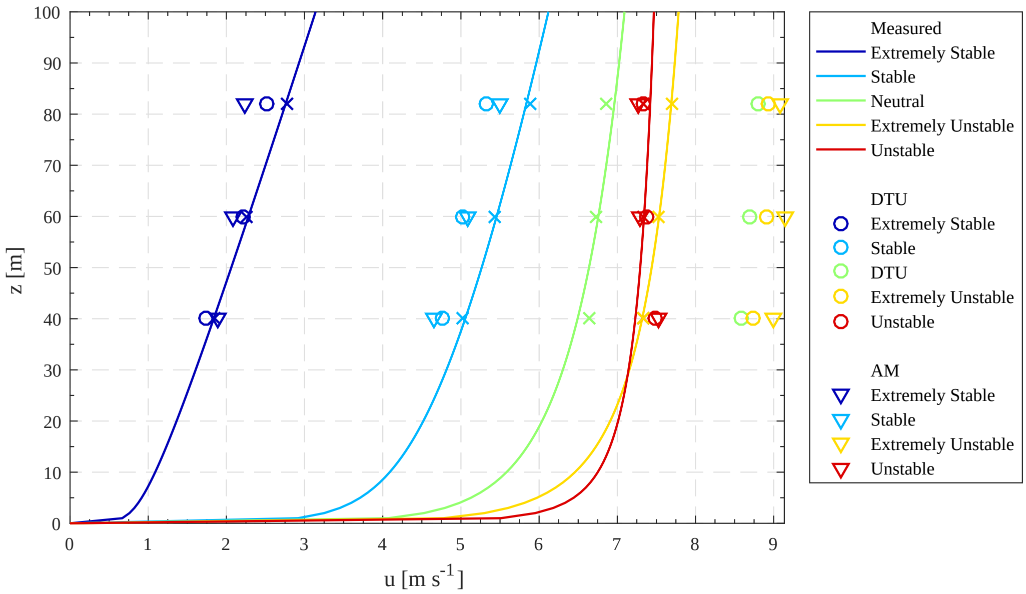


Figure 5.8: Predicted vs. measured wind speed profiles at Mast 2
The crosses indicate the mean measured velocity

Marginally increased errors are present for both models in the unstable condition. The worst performing model is the neutral model with a 28.5 % error. This high error can be attributed to the increased variance in the neutral data. The neutral condition is only present for 11% of the measurement campaign and by analysing the diurnal stability classification in Figure 3.7 the neutral condition does not have a fixed period in which it occurs, instead occurring at any time of day. There is thus higher variance in the neutral data which causes the increased error.

In order to understand the total error a frequency weighted error is calculated. This error is weighted according to the stability frequency classification and is determined as

$$\text{Total Error} = \frac{\sum_{j=1}^{5} \text{frequency}_j \times \text{Error}_j}{\sum_{j=1}^{5} \text{frequency}_j} \tag{5.4}$$

where j indicates the five stability classes, the error is obtained from Table 5.2 and the frequency is the stability frequency classification presented in Figure 3.5. The total error is calculated as 8.55% for the DTU model and 8.54% for the AM model. There is thus negligible difference between these two models in the total cross prediction error and both models have a error of less then 10% in cross prediction.

From the profiles in Figure 5.8 it can be seen that both models were able to accurately predict the shape of the wind profiles. In stable the high shear exponent causes the more flattened profiles and in unstable the profiles are closer to upright as there is very little change in velocity with height. The only condition that has an error in this regard is the extremely unstable condition in which both models have problems predicting the complete vertical profile, instead over-predicting the velocity at 42 m.

5.4 Stability Lifting/Blocking Effects

As described in Section 2.1 one the effects of non-neutral stratification is that of lifting and blocking the air flow. In stable conditions the wind profiles tend to flow around rather than over obstacles as it would in the neutral conditions and in unstable conditions the profiles keep rising after the obstacle.

This effect is present in the CFD results. In Figure 5.9 the neutral velocity streamlines over a specific hill section in the terrain are shown. The hill has a slight opening toward the Eastern part. The streamlines are released directly in front and perpendicular to the hill. In the neutral condition the streamlines flow over the hill completely straight and smooth with no turbulent mixing behind the hill.

In Figures 5.10 to 5.13 the streamlines released from the same location in unstable and stable conditions are shown for the DTU and AM models. Both models exhibit the same behaviour and were accurately able to capture the lifting and blocking effects.

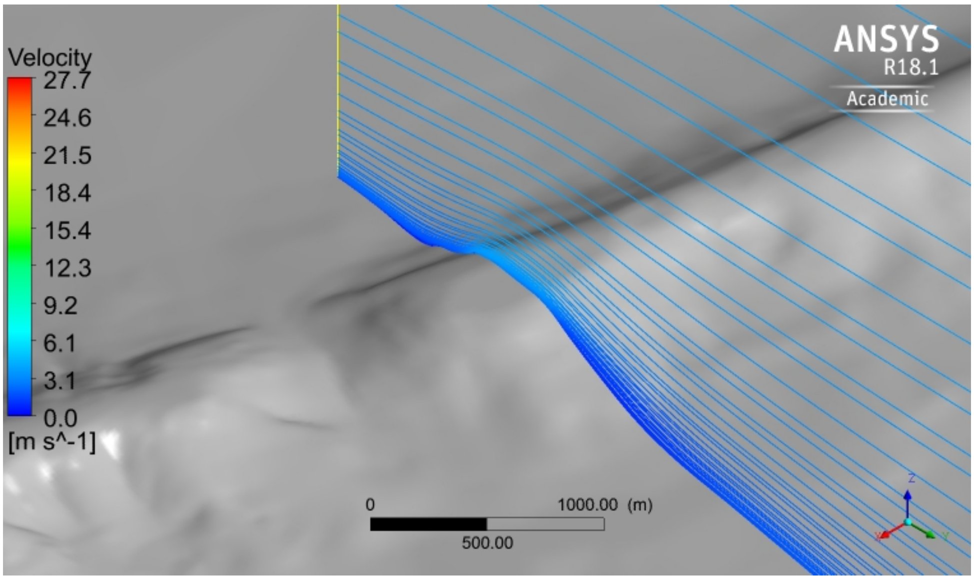


Figure 5.9: Velocity streamlines over terrain feature under neutral stratification

In stable conditions, Figures 5.10 and 5.12, the streamlines flow around the hill towards the opening instead of over. This effect causes the high wind shear values experienced in stable conditions. The streamlines close to ground flow around instead of up the hill. A slow moving parcel of air is thus experienced close to ground on top of the hill, the streamlines higher above ground do flow over the hill and where these two meet there is an increased change of velocity with height which leads to the high wind shear values.

In unstable conditions, Figures 5.11 and 5.13, the streamlines go over the hill and travel onwards after the hill instead of flowing smoothly down. This causes the turbulent mixing zone that is present behind the hill, this zone was captured by both models. This increased turbulence is the reason why in unstable conditions the turbulence intensity is increased from the neutral and stable conditions.

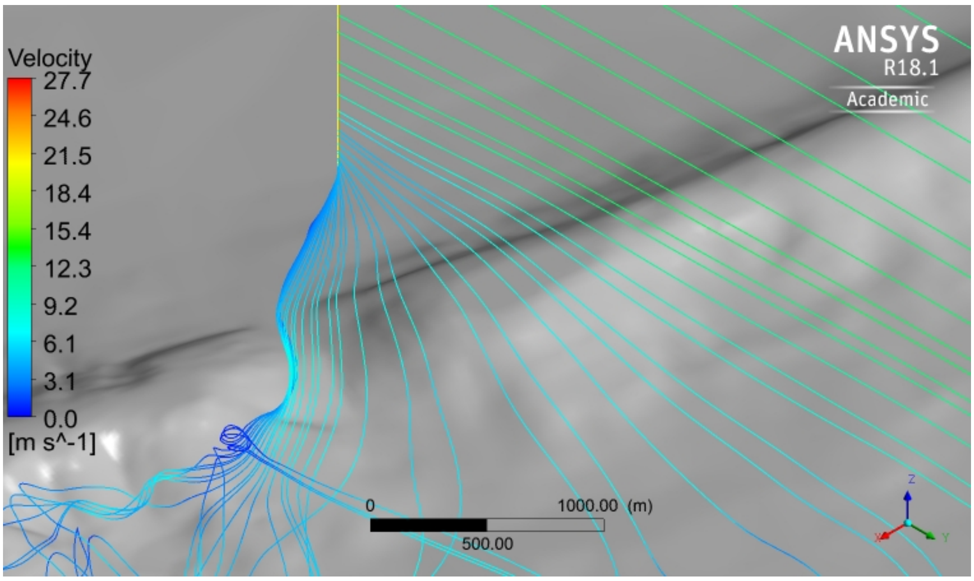


Figure 5.10: Effect of atmospheric stability on velocity streamlines - DTU model Stable

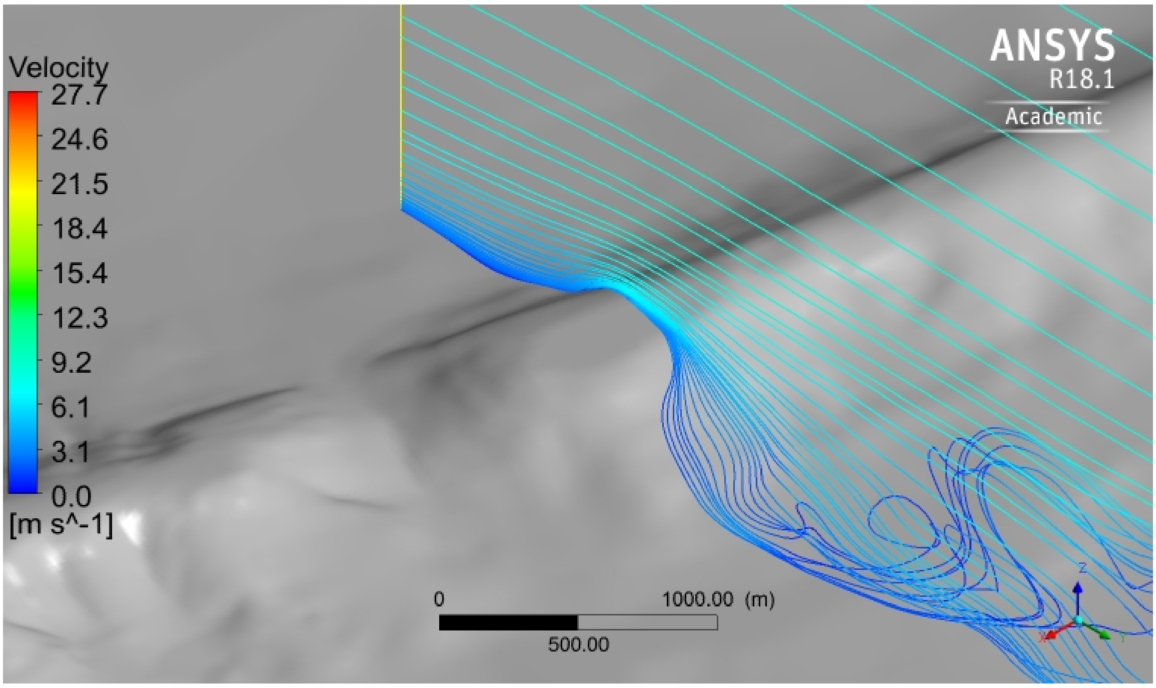


Figure 5.11: Effect of atmospheric stability on velocity streamlines - DTU model Unstable

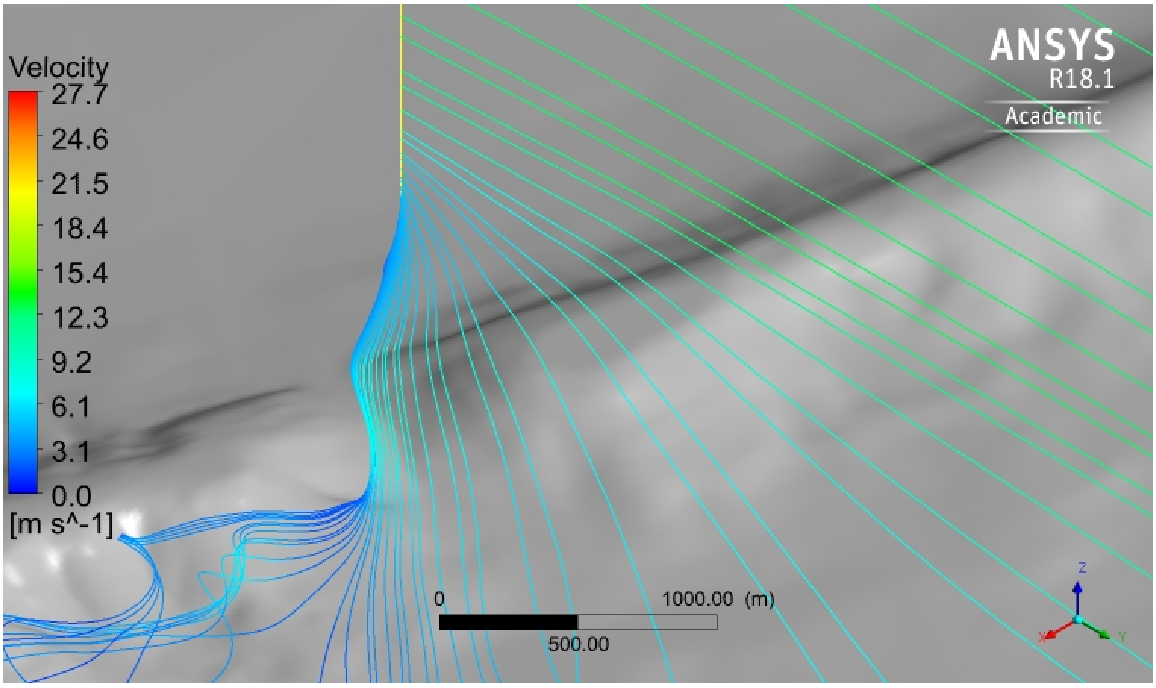


Figure 5.12: Effect of atmospheric stability on velocity streamlines - AM model Stable

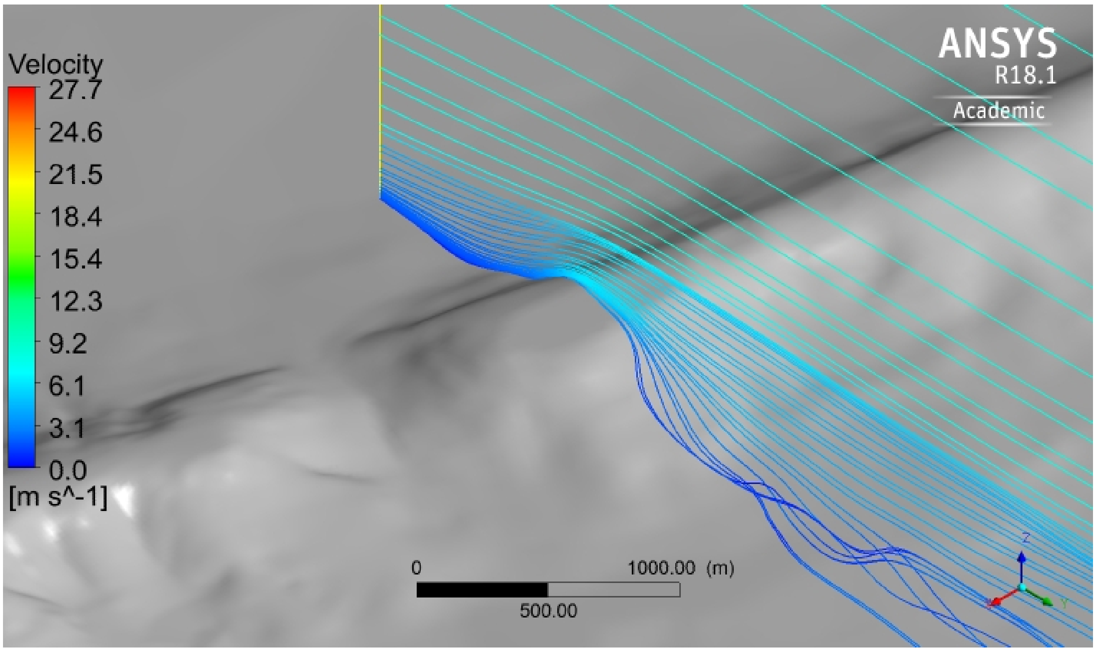


Figure 5.13: Effect of atmospheric stability on velocity streamlines - AM model Unstable

5.5 Summary

Based on the results presented in this chapter it can be concluded that both models were able to successfully model the onsite effects of atmospheric stability. Applying the developed data analysis procedure on a WRF mesoscale data point at the inlet and the primary mast at the centre of the site yielded accurate inputs to the CFD model. The cross-prediction study successfully validated the ABL model with low errors experienced in all non-neutral conditions. A total error of 8.5% was obtained for both models. The greatest errors occurred for conditions which are non-dominant and it can be concluded that care should be taken when analysing these conditions due the naturally increased variance in non-dominating conditions. The lifting and blocking effects known to be caused by stratification were also found to be in accordance to those described in literature.

The difference in errors from both models are negligible and not one clear model performed better than the other. The only major difference in cross-prediction error is in the extremely stable condition, however, as this condition is not one of the dominating conditions using it to decide on one model or the other is premature. Further cross prediction studies on wind farm locations with other conditions and terrains are therefore required to accurately comment on which model is best. Both models are successfully validated for modelling atmospheric stability. However, care should be taken in conditions where the AM model loses validity, as the polynomial used in its formulation is only valid for $-2.3 < z/L < 2.0$.

Chapter 6

Conclusions

This study presented an atmospheric boundary layer (ABL) CFD model which aims to describe neutral and non-neutral wind flow over complex terrain using site-specific stability parameters. The model was successfully validated using a horizontal homogeneity test and a cross-prediction study from a proposed wind farm location.

The prevalence and effect of atmospheric stability on the windfields were determined by applying Monin-Obukhov Similarity Theory (MOST) to two years of onsite measured time series data. The results indicated strong non-neutral conditions with neutral conditions present for only 11% of the measurement period. The central limit theorem was applied and mean conditions were determined using the diurnally weighted average. The results showed that large variations in conditions were present with increased wind shear during extremely stable conditions and increased turbulence during unstable conditions. The results highlight the shortcomings of assuming only neutral conditions when determining the site conditions. The data analysis method which applies MOST to measured time series data and uses the diurnally weighted average to determine sector-wise mean conditions were developed by the author. It is to the best of the authors knowledge a novel implementation of MOST to determine atmospheric stability and vertical profiles of velocity, temperature and turbulence.

MOST is known to be incompatible with the standard $k - \epsilon$ turbulence model and modifications to the standard model CFD equations are required in non-neutral ABL simulations. Modifications are also required to the standard wall-function methods. The required modifications to the standard CFD model equations were implemented by User Defined Functions (UDF).

The first step towards the validation of the ABL CFD model was the horizontal homogeneity test in which the MOST and wall-function modifications were tested to be in equilibrium by the model's ability to sustain inlet profiles in an empty domain. The results showed that the standard method of using a modified roughness value in the ABL model breaks down under high roughness. A modified ABL specific wall-function is preferred which is able to sustain neutral profiles accurately for distances of over 10000 m while allowing more freedom in mesh generation at ground level. Two MOST models were tested, the results from both models highlighted problems modelling extreme conditions and maintaining profiles for extended distances. Both models were able to accurately maintain profiles of velocity and turbulence up to 5000 m. The standard buoyancy turbulent production term was shown to be incompatible with steady-state simulations and the MOST formulation is preferred as it produced accurate profiles of velocity and turbulence in steady-state simulations.

The final model validation procedure applied the ABL CFD model to the complex wind farm terrain utilized in the data analysis study. Both MOST models were tested by using the CFD results to cross predict stability-dependent velocity profiles from the two onsite meteorological masts. During the two main stability conditions experienced, both models gave errors of less than 10%. The DTU model showed it is more capable of dealing with the extreme cases than the AM model due to being valid throughout the computational domain. Using the frequency classification, both models gave a total error of 8.5% which proves both models were successfully validated and able to accurately model non-neutral flows onsite. To the best of the authors knowledge this study presents the first application of the DTU model to complex terrain as well as the first comparison of the AM and DTU models in complex terrain. The AM model application to complex terrains has been studied [34].

The advantage of using the proposed ABL CFD model is the ability to model more of the large scale physical mechanisms of the ABL. This allows greater accuracy in the design of wind farms. On the proposed location used in this study the two masts are located more than 7000 m apart and the model was able to accurately predict the velocity profiles experienced at the other mast location. Using this method the measured stability-dependent profiles can be accurately extrapolated to any proposed turbine location onsite and used in turbine loading and power production calculations. In summary, the results showed that the implemented modifications and developed methods are applicable and reproduced the main wind flow characteristics in neutral and non-neutral flows over complex wind farm terrains.

6.1 Future Work

Although the methods developed in this study have shown significant improvements over the neutral CFD models there are several issues that warrant further investigation.

In the horizontal homogeneity test both models showed increased errors under extreme conditions. These models are presented in literature under standard non-neutral conditions. However, the DTU model is in balance for all conditions and based on an unpublished study the DTU model author was able to accurately model the extreme cases using the EllipSys3D CFD code. Further work can therefore be done to understand if the errors produced during non-neutral modelling are associated with the CFD code. The Ansys CFX and OpenFOAM codes are cadidates for further testing.

The study focused on steady state-simulations, using transient simulations the standard buoyant turbulence production term can be utilised.

The user defined function implementation of the MOST models currently uses a linear interpolation scheme between the inlet and primary mast location. Further investigations can be performed on sites where multiple masts are present to perform interpolation between each of the mast locations.

Incorporating the methods utilised in this study into the existing models currently used to access commercially proposed wind farms requires thorough model validation. The current state of field experiments available for ABL CFD models are not sufficient [7] [3]. During typical measurement campaigns temperature data are often not sufficient to classify stability satisfactorily. The data measured at commercial wind farms are also not readily available for open use. The Bolund Hill field experiment is only applicable to neutral modelling [35]. The Benakanahalli hill field experiment does include the necessary measurements and is subject to non-neutral stratification. However, the experiment conditions are not ideal with low wind speeds, high turbulence and only a fraction of wind flow from the sector of interest [3] [36].

Finally, other turbulence models can be investigated such as the $k - \omega$ model or an eddy-solving method. Using LES or DES would require new methods to be developed to provide non-neutral transient boundary conditions and the required modifications to the subgrid-scale turbulence model to account for buoyancy effects.

Bibliography

[1] M. P. van der Laan, M. C. Kelly, and N. N. Sørensen, "A new k-epsilon model consistent with Monin–Obukhov similarity theory," *Wind Energy*, vol. 20, no. 3, pp. 479–489, 2017.

[2] A. Parente, C. Gorlé, J. van Beeck, and C. Benocci, "Improved k-ϵ model and wall function formulation for the RANS simulation of ABL flows," *Journal of Wind Engineering and Industrial Aerodynamics*, vol. 99, no. 4, pp. 267–278, 2011.

[3] T. Koblitz, *CFD Modeling of Non-Neutral Atmospheric Boundary Layer Conditions*. PhD thesis, Danmarks Tekniske Universitet, 2013.

[4] Matlab, "MATLAB - MathWorks." [Online] (Date last accessed 2017-10-28) http://www.mathworks.com/products/matlab/.

[5] ANSYS, "ANSYS Fluent." [Online] (Date last accessed 2017-10-28) http://www.ansys.com/Products/Fluids/ANSYS-Fluent.

[6] J. M. Wallace and H. P. V., *Atmospheric Science An Introductory Survey*, vol. 2. Elsevier, 2006.

[7] C. Meissner, A. R. Gravdahl, and B. Steensen, "Including thermal effects in CFD simulations," *Journal of the environmental sciences*, p. 5, 2009.

[8] J. F. Manwell, J. G. McGowan, and A. L. Rogers, *Wind Energy Explained*. Wiley, 2 ed., 2009.

[9] T. Foken, "50 Years of the Monin–Obukhov Similarity Theory," *Boundary-Layer Meteorology*, vol. 119, no. 3, pp. 431–447, 2006.

[10] C. Alinot and C. Masson, "Aerodynamic Simulations of Wind Turbines Operating in Atmospheric Boundary Layer With Various Thermal Stratifications," *ASME 2002 Wind Energy Symposium*, no. July, pp. 206–215, 2002.

[11] A. J. Dyer, "A review of flux-profile relationships," *Boundary-Layer Meteorology*, vol. 7, no. 3, pp. 363–372, 1974.

[12] A. Sathe, J. Mann, T. Barlas, W. A. A. M. Bierbooms, and G. J. W. Van Bussel, "Influence of atmospheric stability on wind turbine loads," *Wind Energy*, vol. 16, pp. 1013–1032, Oct 2013.

[13] J. S. Irwin and F. S. Binkowski, "Estimation of the Monin-Obukhov scaling length using on-site instrumentation," *Atmospheric Environment (1967)*, vol. 15, no. 6, pp. 1091–1094, 1981.

[14] R. S. Davis, "Equation for the Determination of the Density of Moist Air (1981/91)," *Metrologia*, vol. 29, no. 1, p. 67, 1992.

[15] H. Versteeg and W. Malaasekera, *An introduction to computational fluid dynamics: the finite volume method*. Pearson Education Limited, 2 ed., 1995.

[16] G. Crasto, *Numerical Simulations of the Atmospheric Boundary Layer*. PhD thesis, University of Cagliari, 2007.

[17] ANSYS, "ANSYS Fluent Theory Guide 18.1," tech. rep., Ansys Fluent 18.1, 2017.

[18] B. E. Launder and D. B. Spalding, "The numerical computation of turbulent flows," *Computer Methods in Applied Mechanics and Engineering*, vol. 3, pp. 269–289, Mar 1974.

[19] ANSYS, "ANSYS Fluent User Guide 18.1," tech. rep., Ansys Fluent 18.1, 2017.

[20] B. Blocken, T. Stathopoulos, and J. Carmeliet, "CFD simulation of the atmospheric boundary layer: wall function problems," *Atmospheric Environment*, vol. 41, no. 2, pp. 238–252, 2007.

[21] X. Zhang, *CFD simulation of neutral ABL flows*. PhD thesis, Danmarks Tekniske Universitet, Apr 2009.

[22] F. R. Freedman and M. Z. Jacobson, "Modification of the standard e-equation for the stable ABL through enforced consistency with Monin-Obukhov similarity theory," *Boundary-Layer Meteorology*, vol. 106, no. 3, pp. 383–410, 2003.

[23] A. Parente, C. Gorlé, J. van Beeck, and C. Benocci, "Improved k-ϵ model and wall function formulation for the RANS simulation of ABL flows," *Journal of Wind Engineering and Industrial Aerodynamics*, vol. 99, no. 4, pp. 267–278, 2011.

[24] H. A. Panofsky and J. A. J. A. Dutton, "Atmospheric turbulence : models and methods for engineering applications," 1984.

[25] P. J. Richards and R. P. Hoxey, "Appropriate boundary conditions for computational wind engineering models using the k-e turbulence model," *Journal of Wind Engineering and Industrial Aerodynamics*, vol. 46, pp. 145–153, 1993.

[26] A. Sogachev, M. Kelly, and M. Y. Leclerc, "Consistent Two-Equation Closure Modelling for Atmospheric Research: Buoyancy and Vegetation Implementations," *Boundary-Layer Meteorology*, vol. 145, pp. 307–327, Nov 2012.

[27] C. Alinot and C. Masson, "k-epsilon Model for the atmospheric boundary layer under various thermal stratifications," *Journal of Solar Energy Engineering-Transactions of The ASME*, vol. 127, no. 4, pp. 438–443, 2005.

[28] T. G. Farr, P. A. Rosen, E. Caro, R. Crippen, R. Duren, S. Hensley, M. Kobrick, M. Paller, E. Rodriguez, L. Roth, D. Seal, S. Shaffer, J. Shimada, J. Umland, M. Werner, M. Oskin, D. Burbank, and D. E. Alsdorf, "The shuttle radar topography mission," *Reviews of Geophysics*, vol. 45, p. RG2004, May 2007.

[29] J. Mann, S. Ott, B. H. Jørgensen, and P. Frank, "WAsP Engineering 2000," *Risø–R–1356(EN)*, vol. 1356, p. 91, Aug 2002.

[30] M. Strack, "MEASNET Procedure „Evaluation of Site-Specific Wind Conditions" Released," *DEWI Magazin*, vol. 36, pp. 76–81, 2010.

[31] EMD, "EMD International A/S – EMD-WRF South Africa Mesoscale Data." [Online] (Date last accessed 2017-10-28) http://www.emd.dk/windpro/mesoscale-data/emd-wrf-south-africa-mesoscale-data/.

[32] ANSYS, "ANSYS Fluent Customization Manual 18.1," tech. rep., Ansys Fluent 18.1, 2017.

[33] P. G. Tucker, C. L. Rumsey, P. R. Spalart, R. E. Bartels, and R. T. Biedron, "Computations of wall distances based on differential equations," *AIAA Journal*, vol. 43, no. 3, pp. 539–549, 2005.

[34] J. Pieterse and T. Harms, "CFD investigation of the atmospheric boundary layer under different thermal stability conditions," *Journal of Wind Engineering and Industrial Aerodynamics*, vol. 121, pp. 82–97, Mar 2013.

[35] J. Berg, J. Mann, A. Bechmann, M. S. Courtney, and H. E. Jørgensen, "The Bolund Experiment, Part I: Flow Over a Steep, Three-Dimensional Hill," *Boundary-Layer Meteorology*, vol. 141, no. 2, pp. 219–243, 2011.

[36] T. Koblitz, A. Bechmann, J. Berg, A. Sogachev, N. Sørensen, and P.-E. Réthoré, "Atmospheric stability and complex terrain: comparing measurements and CFD," *Journal of Physics: Conference Series*, vol. 555, p. 12060, 2014.

Appendices

Appendix A

Roughness Lengths

Table A.1: Typical Roughness Lengths [8]

Terrain description	z_0 (mm)
Very smooth, ice or mud	0.01
Calm open sea	0.2
Blown sea	0.5
Snow surface	3
Lawn grass	8
Rough pasture	10
Fallow field	30
Crops	50
Few trees	100
Many trees, hedges, few buildings	250
Forest and woodlands	500
Suburbs	1500
City centres with tall buildings	3000

Appendix B

Mast Data Sample

Table B.1: Mast data sample

TIMESTAMP TS	REC RN	S1V82M m/s Avg	S1V82M m/s Std	S1V82M m/s Max	D1V80M Deg Avg	D1V80M Deg Std
12/06/2015 11:00	562	4.898	0.478	5.773	112	5.063
12/06/2015 11:10	563	4.308	0.571	5.607	107.6	8.44
12/06/2015 11:20	564	3.938	0.405	5.207	113.7	8.29
12/06/2015 11:30	565	3.297	0.552	4.79	115.3	10.07
12/06/2015 11:40	566	3.461	0.51	4.657	115.1	9.49
12/06/2015 11:50	567	3.529	0.442	4.64	116.8	9.98
12/06/2015 12:00	568	3.58	0.447	4.673	113.8	9.43

TIMESTAMP TS	REC RN	Press5m mBar Avg	Temp5m Deg C Avg	RH5m % Avg
12/06/2015 11:00	562	843	10.2	75.51
12/06/2015 11:10	563	843	10.3	75.64
12/06/2015 11:20	564	843	10.62	74.74
12/06/2015 11:30	565	843	10.92	73.49
12/06/2015 11:40	566	843	11.15	73.2
12/06/2015 11:50	567	843	11.27	72.98
12/06/2015 12:00	568	843	11.5	72.18

Appendix C

Data Analysis Code

The data analysis code is included below. Coded using Matlab 2016a. Requires the optimization and statistics toolboxes. It accepts WindPro3.1 meteorogical mast data exports as inputs.

C.1 dataAnalysis.m

```matlab
function [mastStruct,profiles,turbModelConstants,sectorTables,diurnals,figs] = ...
    dataAnalysis()
%% Data Analysis
%
% Analyse Met Export data from WindPro to determine atmospheric stability
% based on the gradient Richardson number or MOL approach.
% Can handle Met mast and Mesoscale data sets
% Sectorwise stability, MOL, Shear and Velocity tables are given as outputs
% as well as a mast structure containing time series data split into the
% various stavbility cases.
% Metrics are presented in the Figure outputs
% Each function has its own description about the methods involved with
% references
% Requires the stabilityRose.m code on the path.
% Control preferences by changing values in input section
%
% Function Call Example:
%
% Rules for exporting from WindPro
% - Normal meteo object export
% - Remove names of heights
% - Only use one channel per height
% - Data must be exported after it has been cleaned, the functions will
%   clean data according to how it was originally done in Windpro
% - Do not include channels that are mostly disabalded. (Low Availability)
% - Temperature and Pressure should appear in the same channels if more
%   than one pressure is used
% - Do not repeat any channels
%
%
% ----------------------------------------------------------------------
% Owner: Hendri Breedt <u10028422@tuks.co.za>
% Date: 09/11/2017
% Version: 00 - Public release

clearvars
fclose all;
close all;
%% Load & Clean Data

% Load Data
[mast,mastName,header,dateRangeStr,inputFilenamePath] = dataImport();

% Clean Data
[mast,U,D,Ti,T,P,RH,Zs,Zt,TiAvail] = dataClean(mast,header,mastName,dateRangeStr);

% pause; % Paused so the user can now change the script below.

%% Inputs %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Start and End Dates if you want specific period: Format 'dd/mm/yyyy HH:MM'
startDateStr = '01/10/2015 00:00';
endDateStr = '01/11/2015 00:00';
% Empty sets and it will use the whole set
% startDateStr = [];
% endDateStr = [];

% Wind speed and Temperature channels
% These are the heights used to determine stability.
% The indexes must be [high, low]
% Note: if scaling is used then the ZsUse index is 'length(Zs)+1'
% ------------------------------------
ZsUse = [1, 3]; % Wind Speed
ZtUse = [2, 1]; % Temperature

```

```
% Number of sectors
% -----------------------------------
sectors  = 12;

% Source values
% The number is the height index as given above
% -----------------------------------
sourceVal = 1; % This height is used as the fixed value for shear and also the binning ...
    for direction, TI & speed

% Shear Scaling
% If Shear Scaling is required, If not the target can be set to a dummy [] value
% -----------------------------------
% shearScaling = 'Yes';
shearScaling = 'No';
targetZ = []; % Target height [m]

% MOL Calculation Method
% Richardson based or Profile Fitting
% Profile fit is more accurate but takes time (10minutes per year)
% -----------------------------------
 molCalcType = 'fit';
% molCalcType = 'Ri';

% Boundary Conditions
% Write profiles for U,T,k,e,w at the data position to use BC's
% -----------------------------------
bcZheight = 1000; % Total height of BC
% bcHeightFix = 500; % Height AGL at which fixed val for wind speed
% velAtBc = 15; % Wind speed at fixed val [m/s]
bcZstep = 1;
zo = 0.002; %[m] % Roughness Length
profSec = 6; % Sector to display turbulence model profiles for
k_eModel = 4; % Choose from the list below;
% |              1              |              2              |
% | k_e Orig (Jones and Launder) | k_e ASL neutral (Sorensen)   |
% |            3                |              4              |
% | k_e MOST (Alinot and Masson) | k_e MOST (Proposed DTU)      |


% Diurnul type
% Cant use 10min if only hourly data is available
% -----------------------------------
 diurType = '10Min';
%  diurType = 'hourly';

% Diurnul Smoothing
% Decide to smooth out the diurnals with a smoothing spline.
% Works well with 10Min diurnals, requires at least 3 data points
% -----------------------------------
% diurSmooth = 'Yes';
diurSmooth = 'No';

% Confidence Inverval Diurnal
% Display 95% confidince interval on diurnal Ri and MOL
% Removing this makes the graphs display more cleanly and clearly
% -----------------------------------
% diurConfi = 'Yes';
diurConfi = 'No';

% Sectorwise shear profiles to plot
% Always uses 4 or 6 sectors
% -----------------------------------
shearSectors2Plot = [4 5 6 7];

% Save Outputs Automatically
% -----------------------------------
saveOutput = 'Yes';
% saveOutput = 'No';
outputType = 'mat';
% outputType = 'txt'; % Saves text files instead of .mat files

% Density Calculation
% Perform density calculation using the selected channels, Selection works
% the same as for the source values.
% Height is dependent on the channels selected by the user below.
% -----------------------------------
densityCalc = 'Yes';
% densityCalc = 'No';
fixedDens = 1.225; % If density calc is not requested a fixed value is used

% Average Pressure
% -----------------------------------
% Use this if no pressure data is available
Pavg = 1000;

% Diurnal Filtering
% This is to clean outlier data that skews the image plot of the diurnal
% when the mean of each time step is taken. This is only a filtering on the
% display data for the diurnal graph. It does not effect the results output
% in the various tables. Alter these values to obtain better looking graphs
% -----------------------------------
diurRiFilterVal = [-25 25];
diurMOLFilterVal = [-1000 1000];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Calculations

% Cut data to start and end date
if ¬isempty(startDateStr)
    startInd = find(datenum(startDateStr,'dd/mm/yyyy HH:MM') == mast.TimeStamp);
    if isempty(startInd)
        error('Start date not found in data set')
    end
else
    startInd = 1;
```

```
    startDateStr = datestr(mast.TimeStamp(1),'dd/mm/yyyy HH:MM');
end
% End Date
if ¬isempty(endDateStr)
    endInd = find(datenum(endDateStr,'dd/mm/yyyy HH:MM') == mast.TimeStamp);
    if isempty(endInd)
        error('End date not found in data set')
    end
else
    endInd = height(mast);
    endDateStr = datestr(mast.TimeStamp(end),'dd/mm/yyyy HH:MM');
end

mast = mast(startInd:endInd,:);
U = U(startInd:endInd,:);
D = D(startInd:endInd,:);
Ti= Ti(startInd:endInd,:);
T = T(startInd:endInd,:);
P = P(startInd:endInd,:);
RH = RH(startInd:endInd,:);
dateRangeStr = [startDateStr,' - ',endDateStr];

if isempty(P)
    P = Pavg*ones(size(mast,1),1);
end

% Replace 0m heights with z0 height
Zt(Zt == 0) = zo;

% Determine Source Values
sourceZ = Zs(:,sourceVal);  % Height of the sources
sourceU = U(:,sourceVal);   % Speed used in figures
sourceD = D(:,sourceVal);   % This direction is used to bin sectors
sourceTi = Ti(:,sourceVal); % Ti used in figures if available
refU = U(:,sourceVal);

% Shear
[mast,U,Zs] = shearScale(shearScaling,mast,U,Zs,refU,sourceZ,targetZ,sourceVal);
% The scaling calculation is always run. This is to determine the shear
% exponent using the least squares method across all the heights. The
% scaled wind speed is only determined if the shear scaling has been
% requested

% Richardson Number
[mast,¬,potenTemp] = richardsonNumber(mast,Zt,ZtUse,Zs,ZsUse,U,T,P);

% Monin-Obukhov - Using nonlinfit
if strcmpi(molCalcType,'fit')
    [mast] = moninObukhovFit(mast,ZtUse,Zs,U,potenTemp,k_eModel,zo);
else
    % Monin-Obukhov - Using Richardson Number
    [mast] = moninObukhov(mast,ZsUse,Zs);
end
stabCond = mast.ConditionMol;

% Ti ans Shear Stability
[mast] = TiShearStab(mast,TiAvail,sourceTi);

% Stability Classification
[numReadings,exUnstable,unstable,neutral,stable,exStable] = ...
    stabilityClass(stabCond,sectors,sourceD);

stabClass = [exUnstable(end), unstable(end), neutral(end), stable(end), ...
    exStable(end)]/numReadings;

% Density Calculation
if strcmp(densityCalc,'Yes')
    try
        densT = T(:,1);
        densP = P(:,1);
        densRH = RH(:,1);
    catch
        densityCalc = 'No';
        waring('Error in density calculation, switching to fixed density')
    end
end
if strcmp(densityCalc, 'Yes')
    rho = airDensity(densT,densRH,densP*100);
    rho(isnan(sum([densT,densRH,densP],2))) = nan;
    mast.Density = rho;
else
    rho = fixedDens*ones(size(mast,1),1);
end

% Diurnal Calculation
% Which type to run
switch diurType
    case '10Min';
        [diurSpeed,diurAlpha,diurTi,diurRi,diurMOL,diurCondition, ...
            diurConditionUnWeight,DiurWeighting] = ...
            diurnal10Minutely(mast,sourceU,sourceTi,stabCond, ...
            diurRiFilterVal,diurMOLFilterVal);
    case 'hourly'
        [diurSpeed,diurAlpha,diurTi,diurRi,diurMOL,diurCondition, ...
            diurConditionUnWeight,DiurWeighting] = ...
            diurnalHourly(mast,sourceU,sourceTi,stabCond, ...
            diurRiFilterVal,diurMOLFilterVal);
    otherwise % If none specified run hourly
        [diurSpeed,diurAlpha,diurTi,diurRi,diurMOL,diurCondition, ...
            diurConditionUnWeight,DiurWeighting] = ...
            diurnalHourly(mast,sourceU,sourceTi,stabCond, ...
            diurRiFilterVal,diurMOLFilterVal);
end
diurnals = struct('Velocity',diurSpeed,'Alpha',diurAlpha,'Ti', ...
    diurTi,'Ri',diurRi,'MOL',diurMOL,'Condition',diurCondition);

```

```
%% Outputs

% Construct Sectorwise MOL Distribution
[MOLTable,maxFreq] = sectorWiseMOL(mast,stabCond,sourceD,sectors,DiurWeighting);

% Construct Sectorwise Shear Exponent Distribution
[shearTable] = sectorWiseShear(mast,stabCond,sourceD,sectors,MOLTable,DiurWeighting);

% Construct Sectorwise Velocity Distribution
[velocityTable,velSecAllHeights] = ...
    sectorWiseVelocity(stabCond,refU,sourceD,sectors,MOLTable,U,Zs,DiurWeighting);

% Construct Sectorwise Ti Distribution if Ti is available
% and create output variable sectorTables
if strcmpi('No', TiAvail)
    sourceTi = 0.5*ones(size(sourceU));
    % Without TiTable
    sectorTables = {MOLTable,shearTable,velocityTable};
elseif strcmpi('Yes', TiAvail)
    [TiTable] = sectorWiseTi(stabCond,sourceTi,sourceD,sectors,MOLTable,DiurWeighting);
    % With TiTable
    sectorTables = {MOLTable,shearTable,velocityTable,TiTable};
end

% Construct Sectorwise Density Distribution
[densTable,densSec] = ...
    sectorWiseDensity(stabCond,rho,sourceD,sectors,MOLTable,DiurWeighting);
if strcmp(densityCalc,'Yes')
    sectorTables = {sectorTables{:},densTable};
end

% Turbulence Model, Profiles and Heat Flux
[profiles,turbModelConstants,qoTable,PotenTempSecAllHeights,uStarTable] = ...
    modelProfiles(mast,Zs,Zt,ZsUse,sourceZ,zo,U,ZtUse,potenTemp,stabCond,sourceD, ...
    sectors,sectorTables,bcZheight,bcZstep, ...
    MOLTable,velSecAllHeights,DiurWeighting,k_eModel,densSec,molCalcType);
sectorTables = {sectorTables{:},qoTable,uStarTable};

% WindSpeed Vs. Stab condition
[velocityStab] = conditionalVelocity(stabCond,refU);

% Create Figures
[figs,stabilityTable] = createFigs(mast,mastName,dateRangeStr, sectors, ...
    sourceU,sourceTi,sourceD, numReadings,exUnstable, unstable,neutral, ...
    stable,exStable, stabCond,diurSmooth, shearSectors2Plot,TiAvail,velSecAllHeights, ...
    Zs,maxFreq,shearScaling ,diurConfi,refU,profiles,PotenTempSecAllHeights,Zt ...
    ,diurSpeed,diurAlpha,diurTi, diurRi,diurMOL, diurCondition,diurConditionUnWeight ...
    ,turbModelConstants,profSec,k_eModel,sourceZ,velocityStab);
sectorTables = {stabilityTable,sectorTables{:}};

% Sector table contents
if strcmpi('No', TiAvail) && strcmp(densityCalc,'Yes')
    sectorTableContent = ...
        {'Frequencies','MOL','Shear','Velocity','Density','HeatFlux','Frictional ...
        Velocity'};
elseif strcmpi('Yes', TiAvail) && strcmp(densityCalc,'No')
    sectorTableContent = ...
        {'Frequencies','MOL','Shear','Velocity','Ti','HeatFlux','Frictional Velocity'};
elseif strcmpi('Yes', TiAvail) && strcmp(densityCalc,'Yes')
    sectorTableContent = ...
        {'Frequencies','MOL','Shear','Velocity','Ti','Density','HeatFlux','Frictional ...
        Velocity'};
elseif strcmpi('No', TiAvail) && strcmp(densityCalc,'No')
    sectorTableContent = {'Frequencies','MOL','Shear','Velocity','HeatFlux','Frictional ...
        Velocity'};
end

% Split Data
[mastStruct] = dataSplit(mast,stabCond);

% Save Outputs
if strcmpi(saveOutput,'Yes')
    dirPath = uigetdir(inputFilenamePath,'Select directory to save all outputs');
    imageSave(figs,mastName, dateRangeStr,shearSectors2Plot,dirPath,TiAvail,profSec);
    dataSave(mastName,dateRangeStr,dirPath, ...
        mastStruct,profiles,turbModelConstants,sectorTables, ...
        diurnals,stabClass,outputType, sectorTableContent,velSecAllHeights)
end

% ------------------------- Sub Functions ------------------------------- %

function [mast,mastName,header,dateRangeStr,inputFilenamePath] = dataImport()
%% dataImport
% Import data from met object stored from the .txt file
% Rules for exporting from WindPro:
% - Only use one channel at a selected height
% - No heights at 0m
% - Do not repeat sensors on channels (Except Direction)


% File information
[fileName,inputFilenamePath] = uigetfile({'*.txt','Meteo Object'},...
                          'Please select the .txt file of the exported met object');
% Open the text file.
fileID = fopen([inputFilenamePath,fileName],'r');
delimiter = '\t';

try % try once with start row = 24 and once with 32 if recalibration
    startRow = 24; % Manually change this if the mast was recalibrated as then the ...
        startrow is later

    % Create format string
    firstBlock = textscan(fileID, '%[^\n\r]', startRow-1, 'WhiteSpace', '', ...
        'ReturnOnError', false); % This reads the header block
    headerStrTotal = firstBlock{1,1}(startRow-2);

        % This part removes the |L|U| section of the header names
```

```
    toDelete = nan(1,2);
    ex = headerStrTotal{:};
    count = 0;
    for i = 1:length(ex)
        if strcmpi(ex(i),'|')
            startInd = i;
            tempInd = regexp(ex(startInd:end),'\t', 'once');
            endInd = tempInd + startInd -1;
            count = count+1;
            toDelete(count,:) = [startInd endInd];
        end
    end

    if ¬isnan(toDelete)
        [¬,uniqueInd] = unique(toDelete(:,2));
        toDelete = toDelete(uniqueInd,:);

        exNew = ex;
        sizeLost = 0;
        for i = 1:size(toDelete,1)
            exNew(toDelete(i,1)-sizeLost:toDelete(i,2)-1-sizeLost) = [];
            sizeLost = sizeLost + length(toDelete(i,1):toDelete(i,2)-1);
        end
        headerStrTotal = {exNew};
    end

    [¬, numChannels] = sscanf(headerStrTotal{:},'%s'); % Count number of channels

    mastName = firstBlock{1, 1}{4, 1}(14:end); % Reads the description of the mast
    if isempty(mastName) || strcmp(mastName,' ')
        mastName = firstBlock{1, 1}{5, 1}(13:end); % Reads the user label of the mast
    end

    if isempty(mastName) || strcmp(mastName,' ') % If still empty use a default name
        mastName = 'NoMastName';
        warning('No mast name detected')
    end
    dateType = firstBlock{1, 1}{6, 1}(19:29);
    % dateType = 'dd-MM-yyyy'; % You can manaully type in the date format here if it ...
        does not work

    formatSpecData = '%s';
    for i = 1:numChannels
        formatSpecData = [formatSpecData,'%f'];
    end

    % Create Header Names
    tabInd = regexp(headerStrTotal,'\s');

    header = cell(1,numChannels);
    header{1} = 'TimeStamp';
    temp = headerStrTotal{1};
    for i = 1:numChannels-1
        header{i+1} = [temp(tabInd{1,1}(i)+1:tabInd{1,1}(i+1)-4),'m'];
    end

catch
    warning('Recalibration detected, setting start row to 32. If no recalibration check ...
        met export')
    fclose all;
    clearvars -except inputFilenamePath fileName delimiter startDateStr endDateStr
    fileID = fopen([inputFilenamePath,fileName],'r');
    startRow = 32;

    % Create format string
    firstBlock = textscan(fileID, '%[^\n\r]', startRow-1, 'WhiteSpace', '', ...
        'ReturnOnError', false); % This reads the header block
    headerStrTotal = firstBlock{1,1}(startRow-3);

        % This part removes the |L|U| section of the header names
    toDelete = nan(1,2);
    ex = headerStrTotal{:};
    count = 0;
    for i = 1:length(ex)
        if strcmpi(ex(i),'|')
            startInd = i;
            tempInd = regexp(ex(startInd:end),'\t', 'once');
            endInd = tempInd + startInd -1;
            count = count+1;
            toDelete(count,:) = [startInd endInd];
        end
    end

    if ¬isnan(toDelete)
        [¬,uniqueInd] = unique(toDelete(:,2));
        toDelete = toDelete(uniqueInd,:);
        exNew = ex;
        sizeLost = 0;
        for i = 1:size(toDelete,1)
            exNew(toDelete(i,1)-sizeLost:toDelete(i,2)-1-sizeLost) = [];
            sizeLost = sizeLost + length(toDelete(i,1):toDelete(i,2)-1);
        end
        headerStrTotal = {exNew};
    end

    [¬, numChannels] = sscanf(headerStrTotal{:},'%s'); % Count number of channels

    mastName = firstBlock{1, 1}{4, 1}(14:end); % Reads the description of the mast
    if isempty(mastName) || strcmp(mastName,' ')
        mastName = firstBlock{1, 1}{5, 1}(13:end); % Reads the user label of the mast
    end

    if isempty(mastName) || strcmp(mastName,' ') % If still empty use a default name
        mastName = 'NoMastName';
        warning('No mast name detected')
    end
```

```
    dateType = firstBlock{1, 1}{6, 1}(19:29);
    % dateType = 'dd-MM-yyyy'; % You can manually type in the date format here if it ...
        does not work

    formatSpecData = '%s';
    for i = 1:numChannels
        formatSpecData = [formatSpecData,'%f'];
    end

    % Create Header Names
    tabInd = regexp(headerStrTotal,'\s');

    header = cell(1,numChannels);
    header{1} = 'TimeStamp';
    temp = headerStrTotal{1};
    for i = 1:numChannels-1
        header{i+1} = [temp(tabInd{1,1}(i)+1:tabInd{1,1}(i+1)-4),'m'];
    end

end
% Read columns of data according to format string data
textscan(fileID, '%[^\n\r]', 0, 'WhiteSpace', '', 'ReturnOnError', false); % This reads ...
    the header block
dataArray = textscan(fileID, formatSpecData, 'Delimiter', delimiter, 'EmptyValue' ...
    ,NaN,'ReturnOnError', false,'TreatAsEmpty','-');
mast = table(dataArray{1:end-1}, 'VariableNames', header);
fclose all;

mast.TimeStamp = datenum(mast.TimeStamp, [lower(dateType) 'HH:MM']); % Convert string ...
    dates to num

startDateStr = datestr(mast.TimeStamp(1),'dd/mm/yyyy HH:MM');
endDateStr = datestr(mast.TimeStamp(end),'dd/mm/yyyy HH:MM');
dateRangeStr = [startDateStr,' - ',endDateStr];

function [mast,U,D,Ti,T,P,RH,Zs,Zt,TiAvail] = dataClean(mast,header,mastName,dateRangeStr)
%% MetExportDataClean Data clean up of WindPro met mast export
% Clean according to the data filtering applied in WindPro
%

%% Pre Allocate

height = [];
for i = 1:length(header)
    heightIdx = regexp(header{i}, '\d');
    if ¬isempty(heightIdx)
  heightNew = str2double(header{i}(heightIdx));
  height = [height heightNew];
    end
end

[¬,I]=unique(height,'first');
height=height(sort(I)); % Find all heights on mast

height = floor(height); % In order to match the text import function
numHeights = length(height);
headerIdxDataStatus = false(numHeights, length(header));
heightStatus = zeros(length(mast{:,1}), numHeights);
headerIdxWSmean = headerIdxDataStatus;
headerIdxWDmean = headerIdxDataStatus;
headerIdxTI = headerIdxDataStatus;
headerIdxTemperature = headerIdxDataStatus;
headerIdxPressure = headerIdxDataStatus;
headerIdxRelHumid = headerIdxDataStatus;
active = [];
rep = [];

%% Get Data Status and Required Channels

for i = 1:numHeights
    % Match Headers to find needed channels
%     statusIdxCell = regexp(header, ['DataStatus\w*','_',num2str(height(i)),'m\w*']);
    statusIdxCell = regexp(header, ['SampleStatus\w*','_',num2str(height(i)),'m\w*']);
    WSmeanIdxCell = regexp(header, ['MeanWindSpeed\w*','_',num2str(height(i)),'m\w*']);
    WDmeanIdxCell = regexp(header, ['Direction\w*','_',num2str(height(i)),'m\w*']);
    TIIdxCell = regexp(header, ['TurbInt\w*','_',num2str(height(i)),'m\w*']);
    temperatureIdxCell = regexp(header, ['Temperature\w*','_',num2str(height(i)),'m\w*']);
    pressureIdxCell = regexp(header, ['Pressure\w*','_',num2str(height(i)),'m\w*']);
    relHumidIdxCell = regexp(header, ...
        ['RelativeHumidity\w*','_',num2str(height(i)),'m\w*']);
    for j = 1:length(header)
        headerIdxDataStatus(i,j) = (¬isempty(statusIdxCell{j}) && statusIdxCell{j} == 1);
        headerIdxWSmean(i,j) = (¬isempty(WSmeanIdxCell{j}) && WSmeanIdxCell{j} == 1);
        headerIdxWDmean(i,j) = (¬isempty(WDmeanIdxCell{j}) && WDmeanIdxCell{j} == 1);
        headerIdxTI(i,j) = (¬isempty(TIIdxCell{j}) && TIIdxCell{j} == 1);
        headerIdxTemperature(i,j) = (¬isempty(temperatureIdxCell{j}) && ...
            temperatureIdxCell{j} == 1);
        headerIdxPressure(i,j) = (¬isempty(pressureIdxCell{j}) && pressureIdxCell{j} == 1);
        headerIdxRelHumid(i,j) = (¬isempty(relHumidIdxCell{j}) && relHumidIdxCell{j} == 1);
    end

    heightStatus = sum(mast{:,headerIdxDataStatus(i,:)} ,2) == 0;
    % Status of the instruments at each height, all instruments active

    headerSize = sum(headerIdxWSmean(i,:) + headerIdxWDmean(i,:) + headerIdxTI(i,:) ...
                  + headerIdxTemperature(i,:) + headerIdxPressure(i,:)); % Size of new ...
                      header

    rep = repmat(heightStatus, [1, headerSize]);
    active  = logical([active rep]);
end

%% Build New Data Set

headerCol = logical(sum(headerIdxWSmean) + sum(headerIdxWDmean) + sum(headerIdxTI) ...
    + sum(headerIdxTemperature) + sum(headerIdxPressure) + sum(headerIdxRelHumid));
headerCol(1) = true(); % Activate DateTime
```

```
% New Sets
headerNew = header(headerCol);
mast = mast(:,headerCol);

% Non Active Values to NaN
mast{:,:}(¬([ones(length(active),1) active])) = nan ;
mast.Properties.VariableNames = headerNew;
remRows = sum(isnan(mast{:,:}),2) ≠ 0;
mast(remRows,:) = []; % Clear Rows with NaN's

% Get Values
U = mast{:,(strncmp(headerNew,'MeanWindSpeed',length('MeanWindSpeed')))};        % Mean ... Wind Speed
D = mast{:,(strncmp(headerNew,'Direction',length('Direction')))};                % ... Direction
Ti = mast{:,(strncmp(headerNew,'Turb',length('Turb')))};                         % ... Turbulence
T = mast{:,(strncmp(headerNew,'Temperature',length('Temperature')))};            % ... Temperature
P = mast{:,(strncmp(headerNew,'Pressure',length('Pressure')))};                  % Pressure
RH = mast{:,(strncmp(headerNew,'RelativeHumidity',length('RelativeHumidity')))}; % ... Relative Humidity

% Get Texts for display
UTxt = headerNew(:,(strncmp(headerNew,'MeanWindSpeed',length('MeanWindSpeed'))));
DTxt = headerNew(:,(strncmp(headerNew,'Direction',length('Direction'))));
TiTxt = headerNew(:,(strncmp(headerNew,'Turb',length('Turb'))));
TTxt = headerNew(:,(strncmp(headerNew,'Temperature',length('Temperature'))));
PTxt = headerNew(:,(strncmp(headerNew,'Pressure',length('Pressure'))));
RHTXT = headerNew(:,(strncmp(headerNew,'RelativeHumidity',length('RelativeHumidity'))));

 % Extract heights for wind speed and temp
Zs = [];
Zt = [];
for i = 1:length(headerNew)
    [startIndWS,endIndWS] = regexp(headerNew{:,i}, 'MeanWindSpeedUID_');
    [startIndT,endIndT] = regexp(headerNew{:,i}, 'TemperatureUID_');
    if ¬isempty(startIndWS) || ¬isempty(endIndWS)
        txt = headerNew{:,i};
        heightWS = str2double(txt(endIndWS+1:end-1));
        Zs = [Zs heightWS];
    elseif ¬isempty(startIndT) || ¬isempty(endIndT)
        txt = headerNew{:,i};
        heightT = str2double(txt(endIndT+1:end-1));
        Zt = [Zt heightT];
    end
end

% Determine if data set has TI data
if isempty(Ti)
    TiAvail = 'No';
    Ti = ones(size(U)); % Create dummy dummy value for Ti so it does not error below
    TiTxt = {'None'};
else
    TiAvail = 'Yes';
end

% Determine if data set has RH data
if isempty(RHTXT)
    RHTXT = {'None'};
end

chanTxt = {UTxt, DTxt, TiTxt, TTxt, PTxt, RHTXT}; % Create channel txts
chanTxtHeadings = {'Wind Speed', 'Wind Direction', 'Turbulence Intensity', ...
    'Temperature', 'Pressure', 'Relative Humidity'};
% Disp header to help user select the channels, in the final GUI this will
% be made into a drop down selection. The available dates are also shown
fprintf('Mast: %s \n',mastName)
fprintf('The available channels are shown below. Please select heights in the order ...
    they are presented \n')
fprintf('----------------------------------------------------------------------- \n')
for i = 1:length(chanTxtHeadings)
    fprintf('*** %s ***\n',chanTxtHeadings{i});
    fprintf('%s\n',chanTxt{i}{:});
    fprintf('------------------------------------------------------------------ \n')
end
fprintf('\n Data is available between the following dates %s \n', dateRangeStr)

function [mast,U,Zs] = shearScale(shearScaling,mast,U,Zs,refU,sourceZ,targetZ,sourceVal)
%% Scale using instantaneous shear profile
% Velocity profile using method from Wind Energy Explained (Manwell)
% Using a least sqaures implimentation with all velocity heights

refInd = 1:length(Zs) ≠ sourceVal;
ZsForScale = Zs(refInd);
UForScale = U(:,refInd);

A = nan(length(ZsForScale)-1, 1);
b = nan(length(ZsForScale)-1, size(UForScale,1));
for i = 1:length(ZsForScale)-1
    A(i,1) = log(ZsForScale(i)/sourceZ);
    b(i,:) = log(UForScale(:,i)./refU)';
end

% Solve least sqaures implimentation for alpha using remaining heights
alphaShear = (A'*A)\(A'*b)';
alphaShear(isinf(abs(alphaShear))) = nan;
if strcmpi(shearScaling, 'Yes') % Only create new entries if requested
    wsScaled =  refU.*(targetZ/refZ).^alphaShear;
%     wsScaled(alphaShear < 0.000001) = nan; % If inversion NaN the values
    % New entries
    U(:,end+1) = wsScaled;
    mast{:,end+1} = wsScaled;

    try % Overwrite if variable already exists
        mast.Properties.VariableNames{end} = ['MeanWindSpeedUID_',num2str(targetZ),'m'];
```

```
    catch
        warning(['Wind Speed at ',num2str(targetZ),'m already defined. Overwriting ...
            values'])
        mast.(['MeanWindSpeedUID_',num2str(targetZ),'m']) = wsScaled;
        mast(:,end) = [];
    end
    Zs(end+1) = targetZ;
    mast.AlphaShear = alphaShear;

else
    mast.AlphaShear = alphaShear;
end

function [mast,stabCond,potenTemp] = richardsonNumber(mast,Zt,ZtUse,Zs,ZsUse,U,T,P)
%% Richardson Number calculation
% Using Equations and limits given in (Ashrafi 2008): 'A Model to Determine
% Atmospheric Stability and its Correlation with CO Concentration'
% Also Potential Temp from Venora Master Thesis
%
%    Condition         | Richardson Number | Condition Number
% ---------------------------------------------------------
% Extremely unstable | Ri < -0.04         |         1
% Unstable           | -0.04 ≤ Ri < 0    |        2
% Neutral            | Ri = 0             |         3
% Stable             | 0 < Ri < 0.25      |         4
% Extremely stable   | Ri ≥ 0.25         |        5

% Zt = Zt(ZtUse);
% T = T(:,ZtUse);
% P = P(:,2);
potenTemp = nan(height(mast), length(Zt));

if size(P,2) == length(Zt)

    for i = 1:length(Zt);
        potenTemp(:,i) = (T(:,i) + 273.15).*(1013.25./P(:,i)).^0.286; % R/Cp = 0.286, ...
            Standard Pressure = 1013.25
    end
else
    for i = 1:length(Zt);
        potenTemp(:,i) = (T(:,i) + 273.15).*(1013.25./P).^0.286; % R/Cp = 0.286, ...
            Standard Pressure = 1013.25
    end

end

Ri = (9.81./(273.15+T(:,end))).*(((potenTemp(:,ZtUse(1)) - ...
    potenTemp(:,ZtUse(2)))/(Zt(1) - Zt(2)))./(((U(:,ZsUse(1)) - ...
    U(:,ZsUse(2)))/(Zs(ZsUse(1)) - Zs(ZsUse(2)))).^2));

Ri(mast.AlphaShear ≤ 0) = -1e03; % if Shear ≠ 0 then Highly Unstable

stabCond = nan(height(mast),1);
stabCond(Ri < -0.04) = 1;
stabCond(Ri ≥ -0.04 & Ri < -0.001) = 2;
stabCond(Ri ≥ -0.001 & Ri < 0.001) = 3;  % Wont achieve perfect 0 with machine ...
    precision will set to 0.0001 tolerance
stabCond(Ri ≥ 0.001 & Ri < 0.25) = 4;
stabCond(Ri ≥ 0.25) = 5;

mast.Richardson = Ri;
mast.ConditionRi = stabCond;

function [mast] = moninObukhov(mast,ZsUse,Zs)
%% Monin-Obukhov Length Calculation
% This method is based on the gradient Richardson number.
% Based on Dyer 1974 with turbine criteria in Sathe 2012 Influence of atmospheric
% stability on wind turbine loads
% This is the method used by Siemens
%
%     Condition        |   Monin-Length    | Condition Number
% ---------------------------------------------------------
% Extremely unstable | -100 < L < 0     |        1
% Unstable           | -500 < L < -100 |        2
% Neutral            | |L| > 500        |        3
% Stable             | 50 < L < 500     |        4
% Extremely stable   | 0 < L < 50       |        5
%
% ToDo: Impliment and compare a direct method using profile methods or frictional
% velocity calculation for MOL Calculation

lS = (Zs(ZsUse(1)) - Zs(ZsUse(2)))/log(Zs(ZsUse(1))/Zs(ZsUse(2))); %Length Scale
Ri = mast.Richardson;
cond = mast.ConditionRi;

L = nan(length(Ri),1);

ind =  (cond == 1 | cond == 2); % Unstable & Extremely Unstable
L(ind) = lS./Ri(ind);

ind = cond == 3; % Neutral;
L(ind) = 10000;

ind = cond == 4; % Stable
L(ind) = lS.*((1-5*Ri(ind))./Ri(ind));

ind = (Ri ≥ 0.2); % Extremely Stable
% L(ind) = nan;     % The method does not work for Ri > 0.2
L(ind) = 25;          % So we force the value to be 25

stabCondMol = nan(height(mast),1);
stabCondMol(L ≥ -100 & L < 0 ) = 1;
stabCondMol(L ≥ -500 & L < -100) = 2;
stabCondMol(abs(L) ≥ 500) = 3;
stabCondMol(L ≥ 50 & L < 500) = 4;
```

```
stabCondMol(L ≥ 0 & L < 50) = 5;

mast.MOL = L;
mast.ConditionMol = stabCondMol;

function [mast] = moninObukhovFit(mast,ZtUse,Zs,U,potenTemp,k_eModel,zo)
%% Monin Obukhov Length using nonlinfit
%

% Stability Splitting - Prelim, only 3 classes, according to the potential temp gradient

stabCondPrelim = nan(height(mast),1);
% 3 = neutral , 4 = stable, 2 = unstable
stabCondPrelim(potenTemp(:,ZtUse(1)) > potenTemp(:,ZtUse(2))) = 4;
stabCondPrelim(potenTemp(:,ZtUse(1)) < potenTemp(:,ZtUse(2))) = 2;
stabCondPrelim(abs(potenTemp(:,ZtUse(1)) - potenTemp(:,ZtUse(2))) < 0.1) = 3; % a 0.1 ...
    difference is used for the neutral condition


kVals = [0.4 0.4 0.42 0.4];
k = kVals(k_eModel);
options = optimoptions('lsqcurvefit');
options.Display = 'off';
h = waitbar(0,'Calculating MOL...');
xResult = nan(height(mast),2);
exitflagResult = nan(height(mast),1);

% Fit velocity profile at each time step to calculate MOL

timeStep = floor(height(mast)/100);
xData = Zs;
for i =1:height(mast)
    if ¬isnan(stabCondPrelim(i)) && ¬any(isnan(U(i,:)))
        yData = U(i,:);
        switch stabCondPrelim(i)
            case 2
                UzModelFun = @(x,xData) (x(1)/k)*(log(xData./zo) - ...
                    2*log((1+(1-16*xData./x(2)).^0.25) ...
                    /2)-log((1+((1-16*xData./x(2)).^0.25) .^2)/2) + ...
                    2*atan((1-16*xData./x(2)).^0.25)-pi/2);
                [x,¬,¬,exitflag] = lsqcurvefit(UzModelFun,[0.3 -5],xData,yData,[0 -inf] ...
                    , [10 0],options);
                xResult(i,:) = x;
                exitflagResult(i) = exitflag;
            case 3
                UzModelFun = @(x,xData) (x(1)/k)*(log(xData./zo));
                [x,¬,¬,exitflag] = lsqcurvefit(UzModelFun,0.3,xData,yData,0,10,options);
                xResult(i,1) = x;
                xResult(i,2) = 10^5;
                exitflagResult(i) = exitflag;
            case 4
                UzModelFun = @(x,xData) (x(1)/k)*(log(xData./zo) + 5*(xData./x(2)));
                [x,¬,¬,exitflag] = lsqcurvefit(UzModelFun,[0.3 5],xData,yData,[0 0] , ...
                    [10 inf],options);
                xResult(i,:) = x;
                exitflagResult(i) = exitflag;
        end
    end
    if rem(i,timeStep) == 0
        waitbar(i/height(mast))
    end
end
close(h)

% Stability Classification
L = xResult(:,2);
stabCondMol = nan(height(mast),1);
stabCondMol(L ≥ -100 & L < 0 ) = 1;
stabCondMol(L ≥ -500 & L < -100) = 2;
stabCondMol(abs(L) ≥ 500) = 3;
stabCondMol(L ≥ 50 & L < 500) = 4;
stabCondMol(L ≥ 0 & L < 50) = 5;

% Remove velocity and potential temperature profiles not matching
removalInd1 = stabCondPrelim == 2 & stabCondMol > 3;
removalInd2 = stabCondPrelim == 4 & stabCondMol < 3;
removalInd3 = stabCondMol == 3 & abs(potenTemp(:,ZtUse(1)) - potenTemp(:,ZtUse(2))) > 0.1;
removalInd = any([removalInd1 removalInd2 removalInd3],2);
xResult(removalInd,1) = nan;
L(removalInd) = nan;
stabCondMol(removalInd) = nan;

mast.uStar = xResult(:,1);
mast.MOL = L;
mast.ConditionMol = stabCondMol;

function [mast] = TiShearStab(mast,TiAvail,sourceTi)
%% Ti and Shear Based Stability Split
% Based on Atmospheric stability affects wind turbine power collection
% Authors: Sonia Wharton1 and Julie K Lundquist

stabCondTiShear = nan(height(mast),1);
shear = mast.AlphaShear;

if strcmpi(TiAvail,'Yes')
    stabCondTiShear(shear < 0 & sourceTi ≥ 0.2) = 1;
    stabCondTiShear((shear ≥ 0 & shear < 0.1) & (sourceTi ≥ 0.13 & sourceTi < 0.2)) = 2;
    stabCondTiShear(shear ≥ 0.1 & shear < 0.2 & (sourceTi ≥ 0.10 & sourceTi < 0.13)) = 3;
    stabCondTiShear(shear ≥ 0.2 & shear < 0.3 & (sourceTi ≥ 0.08 & sourceTi < 0.1)) = 4;
    stabCondTiShear(shear ≥ 0.3 & sourceTi < 0.08) = 5;
else % No TI data avail
    stabCondTiShear(shear < 0) = 1;
    stabCondTiShear(shear ≥ 0 & shear < 0.1) = 2;
    stabCondTiShear(shear ≥ 0.1 & shear < 0.2) = 3;
    stabCondTiShear(shear ≥ 0.2 & shear < 0.3) = 4;
    stabCondTiShear(shear ≥ 0.3) = 5;
end
```

```
mast.ConditionTiShear = stabCondTiShear;

function [numReadings,exUnstable,unstable,neutral,stable,exStable] = ...
    stabilityClass(stabCond,sectors,sourceD)
%% Stability Classification
% Bins conditions into sectors

secAng = 360/sectors; % This determines the sector angles
dirInt = (-secAng/2:secAng:360-secAng/2)';
dirInt(1) = 360-secAng/2;

exUnstable = nan(1,sectors);
unstable = exUnstable; neutral = exUnstable; stable = exUnstable; exStable = exUnstable;
    for j = 1:sectors % Sectors
        if j == 1 % Sector 1
            exUnstable(j) = sum(stabCond == 1 & (dirInt(1) ≤ sourceD  | sourceD  < ...
                dirInt(2)));
            unstable(j) = sum(stabCond == 2 & (dirInt(1) ≤ sourceD  | sourceD  < ...
                dirInt(2)));
            neutral(j) = sum(stabCond == 3 & (dirInt(1) ≤ sourceD  | sourceD  < ...
                dirInt(2)));
            stable(j) = sum(stabCond == 4 & (dirInt(1) ≤ sourceD  | sourceD  < dirInt(2)));
            exStable(j) = sum(stabCond == 5 & (dirInt(1) ≤ sourceD  | sourceD  < ...
                dirInt(2)));
        else % Remaining sectors
            exUnstable(j) = sum(stabCond == 1 & (dirInt(j) ≤ sourceD  & sourceD  < ...
                dirInt(j+1)));
            unstable(j) = sum(stabCond == 2 & (dirInt(j) ≤ sourceD  & sourceD  < ...
                dirInt(j+1)));
            neutral(j) = sum(stabCond == 3 & (dirInt(j) ≤ sourceD  & sourceD  < ...
                dirInt(j+1)));
            stable(j) = sum(stabCond == 4 & (dirInt(j) ≤ sourceD  & sourceD  < ...
                dirInt(j+1)));
            exStable(j) = sum(stabCond == 5 & (dirInt(j) ≤ sourceD  & sourceD  < ...
                dirInt(j+1)));
        end
    end

numReadings = sum(sum([exUnstable' unstable' neutral' stable' exStable'])); % Readings ...
    per sector
exUnstable(j+1) = sum(exUnstable,2);
unstable(j+1) = sum(unstable,2);
neutral(j+1) = sum(neutral,2);
stable(j+1) = sum(stable,2);
exStable(j+1) = sum(exStable,2);

function [rho] = airDensity(t,hr,p)
%% AIR_DENSITY calculates density of air
%  Usage :[ro] = air_density(t,hr,p)
%  Inputs:   t = ambient temperature (C)
%           hr = relative humidity [%]
%            p = ambient pressure [Pa]  (1000 mb = 1e5 Pa)
%  Output:  ro = air density [kg/m3]

%
%  Refs:
% 1)'Equation for the Determination of the Density of Moist Air' P. Giacomo  Metrologia ...
    18, 33-40 (1982)
% 2)'Equation for the Determination of the Density of Moist Air' R. S. Davis Metrologia ...
    29, 67-70 (1992)
%
% Downloaded from Matlab Central
% ver 1.0   06/10/2006    Jose Luis Prego Borges (Sensor & System Group, Universitat ...
    Politecnica de Catalunya)
% ver 1.1   05-Feb-2007   Richard Signell (rsignell@usgs.gov)  Vectorized
% ver 1.2   14/09/2016    Fixed vecorization - Hendri Breedt

%-------------------------------------------------------------------------
T0 = 273.16;         % Triple point of water (aprox. 0C)
 T = T0 + t;         % Ambient temperature in Kelvin

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------
% 1) Coefficients values

 R =  8.314510;           % Molar ideal gas constant   [J/(mol.K)]
Mv = 18.015*10^-3;        % Molar mass of water vapour [kg/mol]
Ma = 28.9635*10^-3;       % Molar mass of dry air      [kg/mol]

 A =  1.2378847*10^-5;    % [K^-2]
 B = -1.9121316*10^-2;    % [K^-1]
 C = 33.93711047;         %
 D = -6.3431645*10^3;     % [K]

a0 =  1.58123*10^-6;      % [K/Pa]
a1 = -2.9331*10^-8;       % [1/Pa]
a2 =  1.1043*10^-10;      % [1/(K.Pa)]
b0 =  5.707*10^-6;        % [K/Pa]
b1 = -2.051*10^-8;        % [1/Pa]
c0 =  1.9898*10^-4;       % [K/Pa]
c1 = -2.376*10^-6;        % [1/Pa]
 d =  1.83*10^-11;        % [K^2/Pa^2]
 e = -0.765*10^-8;        % [K^2/Pa^2]

%-------------------------------------------------------------------------
% 2) Calculation of the saturation vapour pressure at ambient temperature, in [Pa]
psv = exp(A.*(T.^2) + B.*T + C + D./T);   % [Pa]


%-------------------------------------------------------------------------
% 3) Calculation of the enhancement factor at ambient temperature and pressure
fpt = 1.00062 + (3.14*10^-8)*p + (5.6*10^-7)*(t.^2);


%-------------------------------------------------------------------------
```

```
% 4) Calculation of the mole fraction of water vapour
 xv = hr.*fpt.*psv.*(1./p)*(10^-2);

%-------------------------------------------------------------------------
% 5) Calculation of the compressibility factor of air
  Z = 1 - ((p./T).*(a0 + a1*t + a2*(t.^2) + (b0+b1*t).*xv + (c0+c1*t).*(xv.^2))) + ...
       ((p.^2./T.^2).*(d + e.*(xv.^2)));

%-------------------------------------------------------------------------
% 6) Final calculation of the air density in [kg/m^3]
 rho = (p.*Ma./(Z.*R.*T)).*(1 - xv.*(1-Mv./Ma));

function [MOLTable,maxFreq,MOLSec] = ...
    sectorWiseMOL(mast,stabCond,sourceD,sectors,DiurWeighting)
%% SectorWiseMol
% Determines the sectorwise MOL distribution
%

secAng = 360/sectors; % This determines the sector angles
dirInt = (-secAng/2:secAng:360-secAng/2)';
dirInt(1) = 360-secAng/2;
MOL = mast.MOL;

MOLSec = nan(5,sectors);
MOLSecStdDev = MOLSec;

for i = 1:5
    for j = 1:sectors
        if j == 1 % Sector 1
            x = MOL(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < dirInt(2)));
            w = DiurWeighting(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < ...
                dirInt(2)),i);

            MOLSec(i,j) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            MOLSecStdDev(i,j) = std(MOL(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD ...
                 < dirInt(2))),w,'omitnan');
        else      % Remaining sectors
            x = MOL(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < dirInt(j+1)));
            w = DiurWeighting(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < ...
                dirInt(j+1)),i);

            MOLSec(i,j) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            MOLSecStdDev(i,j) = std(MOL(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD ...
                 < dirInt(j+1))),w,'omitnan');
        end
    end
end

hFigTemp = figure(99);
[¬,stabilityTable] = stabilityRose(sourceD,stabCond,hFigTemp,'nDirections', sectors);
close(hFigTemp);

maxFreq = max(cell2mat(stabilityTable(3:end-2,end))); % Max freq used later when the ...
    stab rose is plotted

% Save time by reusing the format from the Stability Table
MOLTable = [stabilityTable;{'-','-','-','-','-','-','-','-'};stabilityTable];
MOLTable{1,1} = 'MOL Length[m]';
MOLTable{sectors+6, 1} = 'Std. Dev MOL Length[m]';
MOLTable(sectors+3:sectors+4,:) = '';
MOLTable(end-1:end,:) = '';
MOLTable(:,8) = [];

MOLTable(3:2+sectors,3:end) = num2cell(MOLSec'); % MOL
MOLTable(end-sectors+1:end,3:end) = num2cell(MOLSecStdDev'); % Std. Dev Gives ...
    indication of confidence of MOL

function [shearTable] = ...
    sectorWiseShear(mast,stabCond,sourceD,sectors,MOLTable,DiurWeighting)
%% SectorWiseShear
% Determines the sectorwise Shear exponent distribution
%

secAng = 360/sectors; % This determines the sector angles
dirInt = (-secAng/2:secAng:360-secAng/2)';
dirInt(1) = 360-secAng/2;
shear = mast.AlphaShear;

shearSec = nan(5,sectors);
shearSecStdDev = shearSec;

for i = 1:5
    for j = 1:sectors
        if j == 1 % Sector 1
            x = shear(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < dirInt(2)));
            w = DiurWeighting(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < ...
                dirInt(2)),i);

            shearSec(i,j) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            shearSecStdDev(i,j) = std(shear(stabCond == i & (dirInt(1) ≤ sourceD  | ...
                sourceD  < dirInt(2))),w,'omitnan');
        else       % Remaining sectors
            x = shear(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < dirInt(j+1)));
            w = DiurWeighting(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < ...
                dirInt(j+1)),i);

            shearSec(i,j) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            shearSecStdDev(i,j) = std(shear(stabCond == i & (dirInt(j) ≤ sourceD  & ...
                sourceD  < dirInt(j+1))),w,'omitnan');
        end
    end
end

% Save time by reusing the format from the MOL Table
shearTable = MOLTable;
```

```
shearTable{1,1} = 'Shear Exponent';
shearTable{sectors+4, 1} = 'Std. Dev Shear Exponent';

shearTable(3:2+sectors,3:end) = num2cell(shearSec'); % MOL
shearTable(end-sectors+1:end,3:end) = num2cell(shearSecStdDev'); % Std. Dev Gives ...
    indication of confidence of Shear value

function [velocityTable,velSecAllHeights] = ...
    sectorWiseVelocity(stabCond,refU,sourceD,sectors,MOLTable,U,Zs,DiurWeighting)
%% SectorWiseVelocity
% Determines the sectorwise average velocity distribution
%

secAng = 360/sectors; % This determines the sector angles
dirInt = (-secAng/2:secAng:360-secAng/2)';
dirInt(1) = 360-secAng/2;

velocitySec = nan(5,sectors);
velocitySecStdDev = velocitySec;

for i = 1:5
    for j = 1:sectors
        if j == 1 % Sector 1
            x = refU(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < dirInt(2)));
            w = DiurWeighting(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < ...
                dirInt(2)),i);

            velocitySec(i,j) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            velocitySecStdDev(i,j) = std(refU(stabCond == i & (dirInt(1) ≤ sourceD  | ...
                sourceD  < dirInt(2))),w,'omitnan');
        else        % Remaining sectors
            x = refU(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < dirInt(j+1)));
            w = DiurWeighting(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < ...
                dirInt(j+1)),i);

            velocitySec(i,j) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            velocitySecStdDev(i,j) = std(refU(stabCond == i & (dirInt(j) ≤ sourceD  & ...
                sourceD  < dirInt(j+1))),w,'omitnan');
        end
    end
end

% Save time by reusing the format from the MOL Table
velocityTable = MOLTable;
velocityTable{1,1} = 'Average Velocity [m/s]';
velocityTable{sectors+4, 1} = 'Std. Dev Velocity [m/s]';

velocityTable(3:2+sectors,3:end) = num2cell(velocitySec');
velocityTable(end-sectors+1:end,3:end) = num2cell(velocitySecStdDev');

% Tabulate the mean velocities at the heights available in the sectors
% Weigted against the diurnals
velSecAllHeights = nan(5,12,length(Zs));
for i = 1:5
    for j = 1:sectors
        for k = 1:length(Zs)
            if j == 1 % Sector 1
                x = U(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < dirInt(2)),k);
                w = DiurWeighting(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < ...
                    dirInt(2)),i);
                velSecAllHeights(i,j,k) = sum(w.*x)./sum(w);
            else        % Remaining sectors
                x = U(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < dirInt(j+1)),k);
                w = DiurWeighting(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < ...
                    dirInt(j+1)),i);
                velSecAllHeights(i,j,k) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            end
        end
    end
end

function [TiTable] = sectorWiseTi(stabCond,sourceTi,sourceD,sectors,MOLTable,DiurWeighting)
%% SectorWiseVelocity
% Determines the sectorwise average Ti distribution
%

secAng = 360/sectors; % This determines the sector angles
dirInt = (-secAng/2:secAng:360-secAng/2)';
dirInt(1) = 360-secAng/2;

TiSec = nan(5,sectors);
TiSecStdDev = TiSec;

for i = 1:5
    for j = 1:sectors
        if j == 1 % Sector 1
            x = sourceTi(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < dirInt(2)));
            w = DiurWeighting(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < ...
                dirInt(2)),i);

            TiSec(i,j) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            TiSecStdDev(i,j) = std(sourceTi(stabCond == i & (dirInt(1) ≤ sourceD  | ...
                sourceD  < dirInt(2))),w,'omitnan');
        else        % Remaining sectors
            x = sourceTi(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < dirInt(j+1)));
            w = DiurWeighting(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < ...
                dirInt(j+1)),i);

            TiSec(i,j) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            TiSecStdDev(i,j) = std(sourceTi(stabCond == i & (dirInt(j) ≤ sourceD  & ...
                sourceD  < dirInt(j+1))),w,'omitnan');
        end
    end
end

% Save time by reusing the format from the MOL Table
```

```
TiTable = MOLTable;
TiTable{1,1} = 'Turbulence Intensity';
TiTable{sectors+4, 1} = 'Std. Dev Turbulence Intensity';

TiTable(3:2+sectors,3:end) = num2cell(TiSec');
TiTable(end-sectors+1:end,3:end) = num2cell(TiSecStdDev');

function [densTable,densSec] = ...
    sectorWiseDensity(stabCond,rho,sourceD,sectors,MOLTable,DiurWeighting)
%% SectorWiseDensity
% Determines the sectorwise average density distribution
%

secAng = 360/sectors; % This determines the sector angles
dirInt = (-secAng/2:secAng:360-secAng/2)';
dirInt(1) = 360-secAng/2;

densSec = nan(5,sectors);
densSecStdDev = densSec;

for i = 1:5
    for j = 1:sectors
        if j == 1 % Sector 1
            x = rho(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < dirInt(2)));
            w = DiurWeighting(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < ...
                dirInt(2)),i);

            densSec(i,j) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            densSecStdDev(i,j) = std(rho(stabCond == i & (dirInt(1) ≤ sourceD  | ...
                sourceD  < dirInt(2))),w,'omitnan');
        else      % Remaining sectors
            x = rho(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < dirInt(j+1)));
            w = DiurWeighting(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < ...
                dirInt(j+1)),i);

            densSec(i,j) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            densSecStdDev(i,j) = std(rho(stabCond == i & (dirInt(j) ≤ sourceD  & ...
                sourceD  < dirInt(j+1))),w,'omitnan');
        end
    end
end

% Save time by reusing the format from the MOL Table
densTable = MOLTable;
densTable{1,1} = 'Density [kg/m3]';
densTable{sectors+4,1} = 'Std. Dev Density';

densTable(3:2+sectors,3:end) = num2cell(densSec');
densTable(end-sectors+1:end,3:end) = num2cell(densSecStdDev');

function [velocityStab] = conditionalVelocity(stabCond,refU)
%% conditionalVelocity
% Bin velocities by stabilty class for use in figure
% Based on the source U

bins = ceil(max(refU));
velocityStab = nan(6,bins+1);
colTotal = nan(1,bins);
for j = 0:bins
    for i = 1:5
    velocityStab(i,j+1) = sum(stabCond == i & refU ≥ j & refU ≤ j+1);
    end
    colTotal(j+1) = sum(velocityStab(1:5,j+1)); % Last Row = Velocity
    velocityStab(6,j+1) = j;
end

delInd = sum(velocityStab(1:end-1,:)) ≤ 3;
velocityStab(:,delInd) = [];
colTotal(delInd) = [];

for j = 0:size(velocityStab,2)-1;
    velocityStab(1:end-1,j+1) = velocityStab(1:end-1,j+1)./colTotal(j+1);
end

function [profiles,turbModelConstants,qoTable,PotenTempSecAllHeights,uStarTable] = ...
    modelProfiles(mast,Zs,Zt,ZsUse,¬,zo,U, ...
    ZtUse,potenTemp,stabCond,sourceD,sectors,sectorTables,bcZheight, ...
    bcZstep,MOLTable,velSecAllHeights, DiurWeighting,k_eModel,densSec,molCalcType)
%% Profiles based on MOST
% Assuming the shear stress and heat flux to be constant over the lower part
% of the atmospheric boundary layer, a modified logarithmic velocity and
% temperature profles are created. Used as boundary conditions
% ref: AERODYNAMIC SIMULATIONS OF WIND TURBINES OPERATING IN ATMOSPHERIC
% BOUNDARY LAYER WITH VARIOUS THERMAL STRATIFICATIONS [Alinot and Masson]
% Frictional velocity using Pieterse 2013 with Dyer approximations for
% fluxes from Dyer 1974 and Venora 2013
%
% Based on measurements of the turbulent kinetic energy budget terms in the
% surface layer of an atmospheric boundary layer over a at terrain one can
% find k,epsilon and Omega. Ref [Alinot and Masson]
%
% Used as boundary conditions from profiles for temp and velocity Ref
% [Monin-Obukhov Similarity Theory Applied to Offshore Wind Data]
% The stability conditon based on MOL is used for the profile creations

%% Constants
modelNames = {'k_e Orig (Jones and Launder)' 'k_e ASL neutral (Sorensen)' 'k_e MOST ...
     (Alinot and Masson)' 'k_e MOST (Proposed DTU)'}';
constantVals = {1.44 1.92 1.0 0.09 1.0 1.3 [] 0.4
    1.21 1.92 0 0.03 1 1.3 [] 0.4
    1.176 1.92 'F_Ce3' 0.033 1 1.3 1 0.42
    1.21 1.92 'F_Ce3' 0.03 1 1.3 1 0.4};

Ce1 = constantVals(:,1);
Ce2 = constantVals(:,2);
Ce3 = constantVals(:,3);
```

```
Cmu = constantVals(:,4);
sigma_k = constantVals(:,5);
sigma_e = constantVals(:,6);
sigma_theta = constantVals(:,7);
K = constantVals(:,8);
k_eConstants = table(modelNames,Ce1,Ce2,Ce3,Cmu,sigma_k,sigma_e,sigma_theta,K);

% Values from model selected
Cmu = cell2mat(k_eConstants.Cmu(k_eModel));
sigma_theta = cell2mat(k_eConstants.sigma_theta(k_eModel));
Ce1 = cell2mat(k_eConstants.Ce1(k_eModel));
Ce2 = cell2mat(k_eConstants.Ce2(k_eModel));
k = cell2mat(k_eConstants.K(k_eModel));

g = 9.81;
Cp = 1003.5;

L = mast.MOL;
MOLSector = cell2mat(sectorTables{1}(3:2+sectors,3:end));
zFine = 0:bcZstep:bcZheight;

%% Frictional Values
U = U(:,ZsUse);
Z1 = Zs(ZsUse(1));
Z2 = Zs(ZsUse(2));
Zt1 = Zt(ZtUse(1));
Zt2 = Zt(ZtUse(2));

psiM1 = nan(length(L),1);
psiM2 = psiM1;
psiT1 = psiM1;
psiT2 = psiM1;

% Using two heights to obtain an initial approximation for the frictional
% values

ind = stabCond == 1 | stabCond == 2; % Extremly Unstable and Unstable
psiM1(ind) = ...
    2*log((1+(1-16*Z1./L(ind)).^0.25)/2)+log((1+((1-16*Z1./L(ind)).^0.25).^2)/2) - ...
    2*atan((1-16*Z1./L(ind)).^0.25)+pi/2;
psiM2(ind) = ...
    2*log((1+(1-16*Z2./L(ind)).^0.25)/2)+log((1+((1-16*Z2./L(ind)).^0.25).^2)/2) - ...
    2*atan((1-16*Z2./L(ind)).^0.25)+pi/2;
psiT1(ind) = 2*log((1+((1-16*Z1./L(ind)).^0.25).^2)/2);
psiT2(ind) = 2*log((1+((1-16*Z2./L(ind)).^0.25).^2)/2);

ind = stabCond == 3; % Neutral
psiM1(ind) = 0;
psiM2(ind) = 0;
psiT1(ind) = 0;
psiT2(ind) = 0;

ind = stabCond == 4 | stabCond == 5; % Stable and Ex Stable
psiM1(ind) = -5*Z1./L(ind);
psiM2(ind) = -5*Z2./L(ind);
psiT1(ind) = psiM1(ind);
psiT2(ind) = psiM2(ind);

% Evaluation of Stability Corrections in
% Wind Speed Profiles Over the North Sea [A.J.M. Van Wijk]
% ind = stabCond == 5; % Extremely Stable
% psiM1(ind) = -0.7*Z1./L(ind) - (0.75*Z1./L(ind) - 10.72*exp(-0.35*Z1./L(ind))) - 10.72;
% psiM2(ind) = -0.7*Z2./L(ind) - (0.75*Z2./L(ind) - 10.72*exp(-0.35*Z2./L(ind))) - 10.72;
% psiT1(ind) = psiM1(ind);
% psiT2(ind) = psiM2(ind);

% Frictional Velocity and Temp Approximations
if  strcmpi(molCalcType,'fit')
    uStarAprox = mast.uStar;
else
    uStarAprox = k*(U(:,2)-U(:,1))./(log(Z2/Z1) - psiM2 + psiM1);
end
potenTempStarAprox = k*(potenTemp(:,2)-potenTemp(:,1))./(log(Zt2/Zt1) - psiT2 + psiT1);
potenTemp0Aprox = (potenTempStarAprox./(uStarAprox.^2)).*(g*k*L);
potenTemp0Aprox(isinf(potenTemp0Aprox)) = nan;
%% Split Sectorwise/Stability Class

secAng = 360/sectors; % This determines the sector angles
dirInt = (-secAng/2:secAng:360-secAng/2)';
dirInt(1) = 360-secAng/2;

uStarSec = nan(sectors,5);
potenTempStarSec = uStarSec;
Temp0Sec = uStarSec;

for i = 1:5
    for j = 1:sectors
        if j == 1 % Sector 1
            uStarSec(j,i) = mean(uStarAprox(stabCond == i & (dirInt(1) ≤ sourceD  | ...
                sourceD  < dirInt(2))),'omitnan');
            potenTempStarSec(j,i) = mean(potenTempStarAprox(stabCond == i & (dirInt(1) ≤...
                 sourceD  | sourceD  < dirInt(2))),'omitnan');
            Temp0Sec(j,i) = mean(potenTemp0Aprox(stabCond == i & (dirInt(1) ≤ sourceD  ...
                | sourceD  < dirInt(2))),'omitnan');
        else        % Remaining sectors
            uStarSec(j,i) = mean(uStarAprox(stabCond == i & (dirInt(j) ≤ sourceD  & ...
                sourceD  < dirInt(j+1))),'omitnan');
            potenTempStarSec(j,i) = mean(potenTempStarAprox(stabCond == i & (dirInt(j) ≤...
                 sourceD  & sourceD  < dirInt(j+1))),'omitnan');
            Temp0Sec(j,i) = mean(potenTemp0Aprox(stabCond == i & (dirInt(j) ≤ sourceD  ...
                & sourceD  < dirInt(j+1))),'omitnan');
        end
    end
end

% Tabulate the mean potenTemp at the heights available in the sectors
```

```
PotenTempSecAllHeights = nan(5,12,length(Zt));
for i = 1:5
    for j = 1:sectors
        for k2 = 1:length(Zt)
            if j == 1 % Sector 1
                x = potenTemp(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < ...
                    dirInt(2)),k2);
                w = DiurWeighting(stabCond == i & (dirInt(1) ≤ sourceD  | sourceD  < ...
                    dirInt(2)),i);
                PotenTempSecAllHeights(i,j,k2) = sum(w.*x)./sum(w);
            else      % Remaining sectors
                x = potenTemp(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < ...
                    dirInt(j+1)),k2);
                w = DiurWeighting(stabCond == i & (dirInt(j) ≤ sourceD  & sourceD  < ...
                    dirInt(j+1)),i);
                PotenTempSecAllHeights(i,j,k2) = sum(w.*x,'omitnan')./sum(w,'omitnan');
            end
        end
    end
end
%% Profiles
stabText = {'Extremely Unstable', 'Unstable', 'Neutral', 'Stable', 'Extremely Stable'};
Uz = nan(sectors,5,length(zFine)); % Velocity Profile
potenTempZ = Uz;                   % Potential Temp Profile
epsilonZ = Uz;                     % epsilon profile
kZ = Uz;                           % k profile
omegaZ = Uz;                       % omega profile
f_Ce3 = Uz;                        % Ce3 k-epsilon profile from DTU MOST model (This is ...
    the default)
qoSec = nan(size(MOLSector));
uStarSecFinal = nan(size(MOLSector));

for i = 1:sectors
    for j = 1:5
        yVel = reshape(velSecAllHeights(j,i,:),length(velSecAllHeights(j,i,:)),1)';
        xZs = Zs;
        beta0Vel = uStarSec(i,j); % Use the UstarSec computed from 2 heights as the ...
            initial guess
        % uStar is the value that we fit for on the heights available

        yTemp = ...
            reshape(PotenTempSecAllHeights(j,i,:),length(PotenTempSecAllHeights(j,i,:)),1)';
        xTemp = Zt;
        beta0Temp = [potenTempStarSec(i,j) 280]; % Use the PotenTempstarSec computed ...
            from 2 heights as the initial guess
        % PotenTemp is the value that we fit for on the heights available
        switch j
            case {1,2} % UnStable
                try
                    UzModelFun =@(uStarBeta,z) (uStarBeta(1)/k)*(log(z./zo) - ...
                        2*log((1+(1-16*z./MOLSector(i,j)).^0.25)/2) ...
                        -log((1+((1-16*z./MOLSector(i,j)).^0.25).^2)/2) + ...
                        2*atan((1-16*z./MOLSector(i,j)).^0.25)-pi/2);
                    %if strcmpi(molCalcType,'fit')
                       % betaVel = beta0Vel; % If profile MOL fit was run use the ...
                           calculated ustar
                    %else
                        [betaVel] = nlinfit(xZs,yVel,UzModelFun,beta0Vel);
                  % end
                    Uz(i,j,:) = UzModelFun(betaVel,zFine);
                    Uz(i,j,1) = 0;

                    potenTempzModelFun =@(potenTempStarBeta,z) potenTempStarBeta(2) + ...
                        (potenTempStarBeta(1)/k)*(log(z./zo) - ...
                        (2*log((1+((1-16*z./MOLSector(i,j)).^0.25).^2)/2)));
                    [betaTemp] = nlinfit(xTemp,yTemp,potenTempzModelFun,beta0Temp);
                    potenTempZ(i,j,:) = potenTempzModelFun(betaTemp,zFine);
                    potenTempZ(i,j,1) = betaTemp(2);

                    psiM = (1-16*zFine./MOLSector(i,j)).^(-0.25);
                    psiT = sigma_theta*(1-16*zFine./MOLSector(i,j)).^(-0.5);
                    psiE = 1-zFine./MOLSector(i,j);

                    epsilonZ(i,j,:) = (betaVel./(k*zFine)).*psiE;
                    kZ(i,j,:) = ((1/sqrt(Cmu))*betaVel^2)*sqrt(psiE./psiM);
                    omegaZ(i,j,:) = k*epsilonZ(i,j,:)./Cmu; %Todo: confirm this eqaution

                    % DTU K-e MOST Model
                    fe = psiM.^(5/2).*(1-0.75*16*zFine./MOLSector(i,j));
                    f_Ce3(i,j,:) = ...
                        (sigma_theta./(zFine./MOLSector(i,j))).*(psiM./psiT).*(Ce1.*psiM ...
                        - Ce2.*psiM + (Ce2-Ce1)./(sqrt(psiE).*fe));

                catch
                    warning('Not enough data to create profile in the %s condition for ...
                        sector %s',stabText{j},num2str(i))
                end

            case 3     % Neutral
                try
                    UzModelFun =@(uStarBeta,z) (uStarBeta/k)*(log(z./zo));
%                     if strcmpi(molCalcType,'fit')
%                         betaVel = beta0Vel; % If profile MOL fit was run use the ...
    calculated ustar
%                     else
                        [betaVel] = nlinfit(xZs,yVel,UzModelFun,beta0Vel);
%                     end
                    Uz(i,j,:) = UzModelFun(betaVel,zFine);
                    Uz(i,j,1) = 0;

                    potenTempzModelFun =@(potenTempStarBeta,z) potenTempStarBeta(2) + ...
                        (potenTempStarBeta(1)/k)*(log(z./zo));
                    [betaTemp] = nlinfit(xTemp,yTemp,potenTempzModelFun,beta0Temp);
                    potenTempZ(i,j,:) = potenTempzModelFun(betaTemp,zFine);
                    potenTempZ(i,j,1) = betaTemp(2);
```

```
                    epsilonZ(i,j,:) = (betaVel^3)./(k*zFine);
                    kZ(i,j,:) = (betaVel^2)./sqrt(Cmu);
                    omegaZ(i,j,:) = k*epsilonZ(i,j,:)./Cmu; %Todo: confirm this eqaution

                    % f_Ce3 = nan; Remains nan's
                catch
                    warning('Not enough data to create profile in the %s condition for ...
                        sector %s',stabText{j},num2str(i))
                end

            case {4,5} % Stable
                try
                    UzModelFun =@(uStarBeta,z) (uStarBeta/k)*(log(z./zo) + ...
                        5*(z./MOLSector(i,j)));
%                     if strcmpi(molCalcType,'fit')
%                         betaVel = beta0Vel; % If profile MOL fit was run use the ...
    calculated ustar
%                     else
                        [betaVel] = nlinfit(xZs,yVel,UzModelFun,beta0Vel);
%                     end
                    Uz(i,j,:) = UzModelFun(betaVel,zFine);
                    Uz(i,j,1) = 0;

                    potenTempzModelFun =@(potenTempStarBeta,z) potenTempStarBeta(2) + ...
                        (potenTempStarBeta(1)/k)*(log(z./zo) + 5*(z./MOLSector(i,j)));
                    [betaTemp] = nlinfit(xTemp,yTemp,potenTempzModelFun,beta0Temp);
                    potenTempZ(i,j,:) = potenTempzModelFun(betaTemp,zFine);
                    potenTempZ(i,j,1) = betaTemp(2);

                    psiM = 1+5*zFine./MOLSector(i,j);
                    psiT = psiM;
                    psiE = psiM-zFine./MOLSector(i,j);

                    epsilonZ(i,j,:) = (betaVel./(k*zFine)).*psiE;
                    kZ(i,j,:) = ((1/sqrt(Cmu))*betaVel^2)*sqrt(psiE./psiM);
                    omegaZ(i,j,:) = k*epsilonZ(i,j,:)./Cmu; %Todo: confirm this eqaution

                    fe = psiM.^(-5/2).*(2*psiM-1);
                    f_Ce3(i,j,:) = ...
                        (sigma_theta./(zFine./MOLSector(i,j))).*(psiM./psiT).*(Ce1.*psiM ...
                        - Ce2.*psiM + (Ce2-Ce1)./(sqrt(psiE).*fe));

                catch
                    warning('Not enough data to create profile in the %s condition for ...
                        sector %s',stabText{j},num2str(i))
                end

        end

        try
            qoSec(i,j) = -Cp*densSec(j,i).*betaVel.*betaTemp(1); % Heat Flux Calc - ...
                Based on Pieterse 2013 - q0 [W/m^2]
        catch
            qoSec(i,j) = nan;
        end

        try
            uStarSecFinal(i,j) = betaVel;
        catch
            uStarSecFinal(i,j) = nan;
        end

    end
end

notes = {'f(sector,condition,height) - condition = [exUnstab,Unstab,Neutral,Stab,exStab]'};
notes2 = {'Inspect values at small Z. Unstable Ce3 should be [+] and Stable Ce3 should ...
    be [-]'};

%% Alinot and Masson
% Only valid of -2.3 < z/L < 2 and also highly sensitive
% z/L< 0.33 | z/L > 0.33 | z/L < -0.25 | z/L > -0.25
if k_eModel == 3
    AMcons = [4.181 5.225 -0.0609 1.765
        33.994 -5.269 -33.672 17.1346
        -442.398 5.115 -546.880 19.165
        2368.12 -2.406 -3234.06 11.912
        -6043.544 0.435 -9490.792 3.821
        5970.776 0.000 -11163.202 0.492];

    n = 0:5;
    aMat = nan(length(zFine),6);
    f_Ce3 = nan(sectors,5,length(zFine)); % A&M k-epsilon model (Numerically unstable ...
        - Consider constant mean values)

    for i = 1:sectors
        for j = 1:5
            zeta = zFine./MOLSector(i,j);
            for l = 1:4
                switch l
                    case 1
                        ind = zeta ≥ 0 & zeta < 0.33;
                    case 2
                        ind = zeta ≥ 0.33 & zeta < 0;
                    case 3
                        ind = zeta < -0.25 & zeta >-2.3;
                    case 4
                        ind = zeta ≥ -0.25 & zeta < 0;
                end
                aMat(ind,1) = AMcons(1,l)';
                aMat(ind,2) = AMcons(2,l)';
                aMat(ind,3) = AMcons(3,l)';
                aMat(ind,4) = AMcons(4,l)';
                aMat(ind,5) = AMcons(5,l)';
                aMat(ind,6) = AMcons(6,l)';
            end
            for k = 1:length(zFine);
```

```
                f_Ce3(i,j,k) =  sum(aMat(k,:).*zeta(k).^n);
            end
        end
    end

    notes2 = {'This model is numerically over sensitive - Consider mean values for Ce3 ...
        Ranges from -0.8 for unstable conditions up to 2.15 in stable conditions'};
end

%% Profile & Constant Outputs
profiles = struct('Uz',Uz,'potenTempZ',potenTempZ,'epsilonZ',epsilonZ, ...
    'kZ',kZ,'omegaZ',omegaZ,'Z',zFine,'Note',notes);

if k_eModel == 3 || k_eModel == 4
    turbModelConstants = struct('Model',modelNames(k_eModel),'Constants' ...
        ,k_eConstants(k_eModel,:),'F_Ce3',f_Ce3,'Note',notes2);
else
    turbModelConstants = struct('Model',modelNames(k_eModel),'Constants' ...
        ,k_eConstants(k_eModel,:));
end
%% Heat Flux Table

qoTable = MOLTable;
qoTable{1,1} = 'Heat Flux [W/m^2]';
qoTable(3:2+sectors,3:end) = num2cell(qoSec);
qoTable = qoTable(1:2+sectors,1:end);

%% Frictional Velocity Table
uStarTable = MOLTable;
uStarTable{1,1} = 'Frictional Velocity [m/s]';
uStarTable(3:2+sectors,3:end) = num2cell(uStarSecFinal);
uStarTable = uStarTable(1:2+sectors,1:end);

function [diurSpeed,diurAlpha,diurTi,diurRi, ...
    diurMOL,diurCondition,diurConditionUnWeight,DiurWeighting] = ...
    diurnalHourly(mast,sourceU,sourceTi,stabCond, diurRiFilterVal,diurMOLFilterVal)
%% Hourly Diurnal Calculation
% Creates 24 hour diurnal using daily hourly averages using the mean value for
% each time cycle. The method uses the normal distribution (mean,stdDev) of
% each time step i.e 01:10 to determine the values. Validated using
% normfit() function

hourVec = datevec(mast.TimeStamp);
hourVec = hourVec(:,4);

% This is the Richardson number used for calculating the diurnal, the
% outliers are removed as to not scew the result with inf values
RiFilterInd = mast.Richardson < diurRiFilterVal(2)  & mast.Richardson > diurRiFilterVal(1);
RiFilter = mast.Richardson(RiFilterInd);
MOLFilterInd = mast.MOL < diurMOLFilterVal(2) & mast.MOL > diurMOLFilterVal(1);
MOLFilter = mast.MOL(MOLFilterInd);

diurAlpha = nan(24,7);
diurTi = diurAlpha;
diurSpeed = diurAlpha;
diurRi = nan(24,2);
diurMOL = diurRi;
diurConditionUnWeight = nan(24,5);
diurCondition = diurConditionUnWeight;
diurCondition2 = diurCondition;
DiurWeighting = nan(height(mast),5);
diurRiMean = nan(24,5);
diurMOLMean = diurRiMean;
diurRiVar = diurRiMean;
diurMOLVar = diurRiMean;


%% Diurnal Hourly

for i = 0:23 % Values for diurnal
    for j = 1:5 % Stability Class
        diurSpeed(i+1,j) = mean(sourceU(hourVec == i & stabCond == j),'omitnan');
        diurAlpha(i+1,j) = mean(mast.AlphaShear(hourVec == i & stabCond == j),'omitnan');
        diurTi(i+1,j) = mean(sourceTi(hourVec == i & stabCond == j),'omitnan');
        diurConditionUnWeight(i+1,j) = sum(hourVec == i & stabCond == j,'omitnan'); % ...
            How many times each condition appeared

        diurRiMean(i+1,j) = mean(RiFilter(hourVec(RiFilterInd) == i & ...
            stabCond(RiFilterInd) == j),'omitnan');
        diurMOLMean(i+1,j) = mean(MOLFilter(hourVec(MOLFilterInd) == i & ...
            stabCond(MOLFilterInd) == j),'omitnan');
        diurRiVar(i+1,j) = var(RiFilter(hourVec(RiFilterInd) == i & ...
            stabCond(RiFilterInd) == j),'omitnan');
        diurMOLVar(i+1,j) = var(MOLFilter(hourVec(MOLFilterInd) == i & ...
            stabCond(MOLFilterInd) == j),'omitnan');
    end

    % Total in second last row
    diurSpeed(i+1,end-1) = mean(sourceU(hourVec == i),'omitnan');
    diurAlpha(i+1,end-1) = mean(mast.AlphaShear(hourVec == i),'omitnan');
    diurTi(i+1,end-1) = mean(sourceTi(hourVec == i),'omitnan');


    % Standard devitaion of the total in last row
    diurSpeed(i+1,end) = std(sourceU(hourVec == i),1,'omitnan');
    diurAlpha(i+1,end) = std(mast.AlphaShear(hourVec == i),1,'omitnan');
    diurTi(i+1,end) = std(sourceTi(hourVec == i),1,'omitnan');
end

%% Normalization Hourly
%Normalize data againts time step for each condition
for j = 1:5
    diurCondition(:,j) = diurConditionUnWeight(:,j)./max(diurConditionUnWeight(:,j)); ...
        %Normalize data
end

%Normalize data against the most dominant condition at each time step
```

```
for i = 1:24
    diurCondition2(i,:) = diurConditionUnWeight(i,:)./max(diurConditionUnWeight(i,:));
end

%% Stability Class Diurnal 10min
% Weighted diurnal conditions
for i = 0:23 % 10 Min values for diurnal
    xRi = diurRiMean(i+1,:);
    xMOL = diurMOLMean(i+1,:);
    varRi = diurRiVar(i+1,:);
    varMOL = diurMOLVar(i+1,:);
    w = diurCondition2(i+1,:);
    %     Total
    diurRi(i+1,1) = sum(w.*xRi,'omitnan')./sum(w,'omitnan');
    diurMOL(i+1,1) = sum(w.*xMOL,'omitnan')./sum(w,'omitnan');

    %     Standard devitaion: From weighted Var method ref: ...
        http://mathworld.wolfram.com/NormalSumDistribution.html
    diurRi(i+1,2) = sqrt(sum((varRi).*(w.^2),'omitnan'));
    diurMOL(i+1,2) = sqrt(sum((varMOL).*(w.^2),'omitnan'));
end

%% Set weighting at each hour
for i = 0:23 % Values for diurnal
    for j = 1:5 % Stability Class
    DiurWeighting(hourVec == i & stabCond == j,j) = diurCondition(i+1,j);
    end
end

function [diurSpeed,diurAlpha,diurTi,diurRi,diurMOL ...
    ,diurCondition,diurConditionUnWeight,DiurWeighting] = ...
    diurnal10Minutely(mast,sourceU,sourceTi,stabCond,diurRiFilterVal,diurMOLFilterVal)
%% 10 Minute Diurnal Calculation
% Creates 24 hour diurnal using daily 10min averages using the mean value for
% each time cycle. The method uses the normal distribution (mean,stdDev) of
% each time step i.e 01:10 to determine the values. Validated using
% normfit() function

hourVec = datevec(mast.TimeStamp);
minsVec = hourVec(:,5);
hourVec = hourVec(:,4);

% This is the Richardson number used for calculating the diurnal, the
% outliers are removed as to not scew the result with inf values
RiFilterInd = mast.Richardson < diurRiFilterVal(2)  & mast.Richardson > diurRiFilterVal(1);
RiFilter = mast.Richardson(RiFilterInd);
MOLFilterInd = mast.MOL < diurMOLFilterVal(2) & mast.MOL > diurMOLFilterVal(1);
MOLFilter = mast.MOL(MOLFilterInd);

diurAlpha = nan(144,7);
diurTi = diurAlpha;
diurSpeed = diurAlpha;
diurRiMean = nan(144,5);
diurMOLMean = diurRiMean;
diurRiVar = diurRiMean;
diurMOLVar = diurRiMean;
diurRi = nan(144,2);
diurMOL = diurRi;
diurConditionUnWeight = nan(144,5);
diurCondition = diurConditionUnWeight;
diurCondition2 = diurConditionUnWeight;
DiurWeighting = nan(height(mast),5);

minComb = [0 10 20 30 40 50]; % Possible 10min combinations

%% Diurnal 10min
countMin = 0;
countHrs = 0;
hrs = 0;
for i = 0:143 % 10 Min values for diurnal

    countMin = countMin + 1;
    countHrs = countHrs + 1;
    mins = minComb(countMin);
    if countMin == 6
        countMin = 0;
    end
    if countHrs == 7
        hrs = hrs + 1;
        countHrs = 1;
    end

    for j = 1:5 % Stability Classes
        diurSpeed(i+1,j) = mean(sourceU(hourVec == hrs & stabCond == j & minsVec == ...
            mins),'omitnan');
        diurAlpha(i+1,j) = mean(mast.AlphaShear(hourVec == hrs & stabCond == j & ...
            minsVec == mins),'omitnan');
        diurTi(i+1,j) = mean(sourceTi(hourVec == hrs & stabCond == j & minsVec == ...
            mins),'omitnan');
        diurConditionUnWeight(i+1,j) = sum(hourVec == hrs & stabCond == j & minsVec == ...
            mins,'omitnan'); % How many times each condition appeared

        diurRiMean(i+1,j) = mean(RiFilter(hourVec(RiFilterInd) == hrs & ...
            stabCond(RiFilterInd) == j & minsVec(RiFilterInd) == mins),'omitnan');
        diurMOLMean(i+1,j) = mean(MOLFilter(hourVec(MOLFilterInd) == hrs & ...
            stabCond(MOLFilterInd) == j & minsVec(MOLFilterInd) == mins),'omitnan');
        diurRiVar(i+1,j) = var(RiFilter(hourVec(RiFilterInd) == hrs & ...
            stabCond(RiFilterInd) == j & minsVec(RiFilterInd) == mins),'omitnan');
        diurMOLVar(i+1,j) = var(MOLFilter(hourVec(MOLFilterInd) == hrs & ...
            stabCond(MOLFilterInd) == j & minsVec(MOLFilterInd) == mins),'omitnan');
    end

%   Total in second last row
    diurSpeed(i+1,end-1) = mean(sourceU(hourVec == hrs & minsVec == mins),'omitnan');
    diurAlpha(i+1,end-1) = mean(mast.AlphaShear(hourVec == hrs & minsVec == ...
        mins),'omitnan');
    diurTi(i+1,end-1) = mean(sourceTi(hourVec == hrs & minsVec == mins),'omitnan');
```

```
%     Standard devitaion of the total in last row
    diurSpeed(i+1,end) = std(sourceU(hourVec == hrs & minsVec == mins),1,'omitnan');
    diurAlpha(i+1,end) = std(mast.AlphaShear(hourVec == hrs & minsVec == ...
        mins),1,'omitnan');
    diurTi(i+1,end) = std(sourceTi(hourVec == hrs & minsVec == mins),1,'omitnan');

end

%% Normilization 10min
%Normalize data againts time step for each condition
for j = 1:5
    diurCondition(:,j) = diurConditionUnWeight(:,j)./max(diurConditionUnWeight(:,j));
end

%Normalize data against the most dominant condition at each time step
for i = 1:144
    diurCondition2(i,:) = diurConditionUnWeight(i,:)./max(diurConditionUnWeight(i,:));
end

%% Stability Class Diurnal 10min
% Weighted diurnal conditions
for i = 0:143 % 10 Min values for diurnal
    xRi = diurRiMean(i+1,:);
    xMOL = diurMOLMean(i+1,:);
    varRi = diurRiVar(i+1,:);
    varMOL = diurMOLVar(i+1,:);
    w = diurCondition2(i+1,:);
    %     Total
    diurRi(i+1,1) = sum(w.*xRi,'omitnan')./sum(w,'omitnan');
    diurMOL(i+1,1) = sum(w.*xMOL,'omitnan')./sum(w,'omitnan');

    %     Standard devitaion: From weighted Var method ref: ...
        http://mathworld.wolfram.com/NormalSumDistribution.html
    diurRi(i+1,2) = sqrt(sum((varRi).*(w.^2),'omitnan'));
    diurMOL(i+1,2) = sqrt(sum((varMOL).*(w.^2),'omitnan'));
end

%% Set weighting at each 10min

countMin = 0;
countHrs = 0;
hrs = 0;
for i = 0:143 % Values for diurnal

        countMin = countMin + 1;
    countHrs = countHrs + 1;
    mins = minComb(countMin);
    if countMin == 6
        countMin = 0;
    end
    if countHrs == 7
        hrs = hrs + 1;
        countHrs = 1;
    end

    for j = 1:5 % Stability Class
        DiurWeighting(hourVec == hrs & stabCond == j & minsVec == mins,j) = ...
            diurCondition(i+1,j);
    end
end

function [figs,stabilityTable] = createFigs(mast,mastName,dateRangeStr,sectors, ...
    sourceU,sourceTi,sourceD,numReadings,exUnstable, ...
    unstable,neutral,stable,exStable,stabCond,diurSmooth, ...
    shearSectors2Plot,TiAvail,velSecAllHeights,Zs,maxFreq,shearScaling, ...
    diurConfi,refU,profiles,PotenTempSecAllHeights,Zt,diurSpeed,diurAlpha,diurTi ...
    ,diurRi,diurMOL,diurCondition,diurConditionUnWeight, ...
    turbModelConstants,profSec,k_eModel,sourceZ,velocityStab)
%% Create figures
% Create the required output figures

% set(groot,'defaultTextInterpreter','latex')
scrsz = get(groot,'ScreenSize');
stabColor = flipud(jet(25));
stabColor = stabColor([1 8 12 18 25],:); % Create colormap for 5 stability cases
% colormap(stabColor);

stabText = {'Extremely Unstable', 'Unstable', 'Neutral', 'Stable', 'Extremely Stable'};
diurTitleText = {'Diurnal Average Wind Speed', 'Diurnal Shear Exponent' , 'Diurnal ...
    Turbulence Intensity', 'Diurnal Richardson Number', 'Diurnal Monin Obukhov Length'};
diurYlabelText = {['$ U_{',num2str(sourceZ),'m} [m/s] $'], '$ \alpha $' , 'Ti', ...
    'Richardson Number', 'Monin Obukhov Length'};
turbModelxLabel = {'$ k $', '$ \epsilon $', '$ \omega $', '$ C_{\epsilon 3} $'};

%% Figure 1 - Total Stability Cases
hFig1 = figure(1);
set(hFig1,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])
y = 100*[exUnstable(end), unstable(end), neutral(end), stable(end), ...
    exStable(end)]/numReadings;

hPie1 = pie(y);
% Fix Pie Chart
hText = findobj(hPie1,'Type','text'); % text object handles
percentValues = get(hText,'String'); % percent values
pieLabeltxt = {'Extremely Unstable: '; 'Unstable: '; 'Neutral: '; 'Stable: '; ...
    'Extremely Stable: '}; % strings
oldExtents_cell = get(hText,'Extent'); % cell array
oldExtents = cell2mat(oldExtents_cell); % numeric array
hText(1).String = strcat(pieLabeltxt(1),strrep(percentValues(1),'%','\%'));
hText(2).String = strcat(pieLabeltxt(2),strrep(percentValues(2),'%','\%'));
hText(3).String = strcat(pieLabeltxt(3),strrep(percentValues(3),'%','\%'));
hText(4).String = strcat(pieLabeltxt(4),strrep(percentValues(4),'%','\%'));
hText(5).String = strcat(pieLabeltxt(5),strrep(percentValues(5),'%','\%'));
newExtents_cell = get(hText,'Extent'); % cell array
newExtents = cell2mat(newExtents_cell); % numeric array
width_change = newExtents(:,3)-oldExtents(:,3);
```

```
signValues = sign(oldExtents(:,1));
offset = signValues.*(width_change/2);
textPositions_cell = get(hText,{'Position'}); % cell array
textPositions = cell2mat(textPositions_cell); % numeric array
textPositions(:,1) = textPositions(:,1) + offset; % add offset
hText(1).Position = textPositions(1,:);
hText(2).Position = textPositions(2,:);
hText(3).Position = textPositions(3,:);
hText(4).Position = textPositions(4,:);
hText(5).Position = textPositions(5,:);
hPie1(1).FaceColor = stabColor(1,:);
hPie1(3).FaceColor = stabColor(2,:);
hPie1(5).FaceColor = stabColor(3,:);
hPie1(7).FaceColor = stabColor(4,:);
hPie1(9).FaceColor = stabColor(5,:);


%% Figure 2 - Sectorwise Bar Chart
hFig2 = figure(2);
set(hFig2,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])
colormap(stabColor)
y = nan(5, sectors);
yTemp = [exUnstable(1:end-1)', unstable(1:end-1)', neutral(1:end-1)', stable(1:end-1)', ...
    exStable(1:end-1)']';
for i = 1:sectors
    y(:,i) = 100*yTemp(:,i)./sum(yTemp(:,i),1); %Normalize with the sectors
end
% y(:,end+1) = 100*[exUnstable(end), unstable(end), neutral(end), stable(end), ...
    exStable(end)]/numReadings;
% if you want the total in the last row
bar(y',1,'stacked') % Bar chart that shows the percentage for each sector
title(['Sectorwise Stability Classification ', mastName, ' ', dateRangeStr])
ylabel('\% Of Sector')
axis tight
xlabel('Sector')
hFig2.CurrentAxes.XTick= 1:sectors;
legend(stabText,'Location','bestoutside','Interpreter','latex');

%% Figure 3 - Stability rose
maxFreq = ceil(maxFreq);
if rem(maxFreq,2) ≠ 0 % To have whole numbers of graph
    maxFreq = maxFreq + 1;
end
optionsStabRose  = {'nDirections', sectors,...
                    'AngleNorth',0,...
                    'AngleEast',90,...
                    'nFreq',maxFreq/2,...
                    'MaxFrequency',maxFreq,...
                    'TitleString', {['Stability Rose ', mastName, ' ', dateRangeStr];''}};
echo stabilityRose off % Turn of warnings from Stability Rose function
hFig3 = figure(3);
set(hFig3,'Position',[1 1 scrsz(3)/2 scrsz(4)/1.5],'Color',[1 1 1])
[hFig3,stabilityTable] = stabilityRose(sourceD,stabCond,hFig3,optionsStabRose);

%% Figure 4 - Stability velocity comparison
hFig4 = figure(4);
set(hFig4,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.25],'Color',[1 1 1])
for i = 1:5
    subplot(2,3,i)
    stabInd = stabCond == i;
    [Ux, Uy] = pol2cart(deg2rad(sourceD(stabInd)),sourceU(stabInd));
    uLim = ceil(max(abs([Ux Uy])));
    scatter(Ux,Uy,2,'b','filled')
    title(stabText(i))
    xlabel(['$ U_{x',num2str(sourceZ),'m} [m/s] $'])
    ylabel(['$ U_{y',num2str(sourceZ),'m} [m/s] $'])
    try
        axis([-uLim(1) uLim(1) -uLim(2) uLim(2)])
    catch
        axis tight
    end
end

%% Figure 5 - TI Vs. Windspeed
hFig5 = figure(5);
set(hFig5,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])
if strcmpi(TiAvail,'Yes')
    for i = 1:5
        subplot(2,3,i)
        stabInd = stabCond == i & sourceU > 0 & ¬isnan(sourceU) & ¬isnan(sourceTi); % ...
            Can not fit with non positive values or NaN's
        try
            f = fit(sourceU(stabInd),sourceTi(stabInd),'power2'); % Fits Power Law of ...
                the form f(x) = a*x^b+c
            plot(f,sourceU(stabInd),sourceTi(stabInd))
            %      plot(sourceU(stabInd), sourceTi(stabInd),'b.')
            coeffNum = coeffvalues(f);
            legend('Data',[num2str(coeffNum(1),2), ...
                '\timesx^{',num2str(coeffNum(2),2),'}+' ...
                ,num2str(coeffNum(3),2)],'Interpreter','latex')
        catch
            warning('Not enough data to fit TI model for %s condition', stabText{i})
            plot(sourceU(stabInd), sourceTi(stabInd),'b.')
        end

        title(stabText(i))
        xlabel(['$ U_{',num2str(sourceZ),'m} [m/s] $'])
        ylabel('Ti')
        if max(sourceTi) > 1
            ylim([0 1])
        else
            ylim([0 max(sourceTi)])
        end
        if max(sourceU) > 25
            xlim([0 25])
        else
```

```
                xlim([0 max(sourceU)])
            end
        end
else
    close(5)
end
%% Figure 6 - Diurnals

diurnals = {diurSpeed,diurAlpha,diurTi,diurRi,diurMOL,diurCondition};
stabColor(end+1,:) = 0; % Last Color = black
diurAmount = size(diurRi,1)-1;

hFig6 = figure(6);
set(hFig6,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.25],'Color',[1 1 1])
for i = 1:4
    subplot(2,2,i)
    if i ≠4
        for j = 1:5
            hold on
            if strcmpi(diurSmooth,'Yes')
                % Spline
                x = linspace(0,24,diurAmount+1)';
                y = diurnals{i}(:,j);
                perDataAvail = 1-sum(isnan(y))/length(y); % percentage data available - ...
                    if less than 25% < then ignore it completely
                indNaN = isnan(x) | isnan(y) | isinf(x) | isinf(y);
                if perDataAvail > 0.25 % Play around iwth the value to get the best cut ...
                    off point to get a clean smoothing spline
                    f = fit(x,y,'smoothingspline', 'Exclude', indNaN); % Fits Spline ...
                        over data to smooth out
                    yNew = feval(f,x);
                    plot(x,yNew,'Color',stabColor(j,:),'LineWidth',1.5)

                else
                    stabTextandTotal = [stabText, 'Total'];
                    warning('Too few diurnal data points available to use smoothing ...
                        spline %s for %s condition. \n Using direct data ...
                        instead',diurTitleText{i}, stabTextandTotal{j})
                    % Direct
                    plot(linspace(0,24,diurAmount+1),diurnals{i}(:,j), ...
                        'Color',stabColor(j,:),'LineWidth',1.5)
                end
            else
                % Direct
                plot(linspace(0,24,diurAmount+1),diurnals{i}(:,j), ...
                    'Color',stabColor(j,:),'LineWidth',1.5)
            end
            hold off
            if i == 3 && strcmp(TiAvail,'No')
                plot(linspace(0,24,diurAmount+1),diurnals{i}(:,j),'Color',[1 1 1]) ...
                    %Overwrites the graph to clear it
                text(12,0.5,'No TI Data ...
                    Available','Color','red','FontSize',14,'HorizontalAlignment','center')
            end
        end
    else
        hold on
        plot(linspace(0,24,diurAmount+1),diurnals{i}(:,1),'k','LineWidth',1.5)
        plot([0 24],[0 0],'Color','k')
        hold off
    end
    title(diurTitleText(i))
    xlabel('Hours')
    axis tight
    hFig6.CurrentAxes.XTick = 0:24;
    ylabel(diurYlabelText(i))

    if i == 1
        legend(stabText,'Location','best','Interpreter','latex')
    end
end

%% Figure 7 - Stability Diurnal
hFig7 = figure(7);
set(hFig7,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])
colormap(stabColor(1:end-1,:)) %Remove black color
y = nan(diurAmount+1,5);
for i = 1:diurAmount+1
    y(i,:) = diurConditionUnWeight(i,:)/sum(diurConditionUnWeight(i,:)); % Normalize ...
        with amount of readings
end

bar(linspace(0,24,diurAmount+1),100*y,1,'stacked') % Bar chart that shows the ...
    percentage for each stability class diurnally
title(['Diurnal Stability Classification ', mastName, ' ', dateRangeStr])
ylabel('\% Of Stability Class')
ylim([0 100])
xlabel('Hours')
axis tight
hFig7.CurrentAxes.XTick = 0:24;
legend(stabText,'Location','bestoutside','Interpreter','latex')

%% Figure 8 - Diurnal Richard and MOL
hFig8 = figure(8);
set(hFig8,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])
colormap(stabColor(1:end-1,:)) %Remove black color
for i = 4:5
    subplot(2,1,i-3)
    hold on
        plot(linspace(0,24,diurAmount+1),diurnals{i}(:,1),'k','LineWidth',1.5)
    if strcmpi(diurConfi,'Yes')
        plot(linspace(0,24,diurAmount+1),diurnals{i}(:,1) ...
            +2*diurnals{i}(:,2),'--r','LineWidth',0.25)
        plot(linspace(0,24,diurAmount+1),diurnals{i}(:,1) ...
            -2*diurnals{i}(:,2),'--r','LineWidth',0.25)
    end
```

```
%     plot([0 24],[0 0],'Color',stabColor(3,:))
    plot([0 24],[0 0],'k')
    if i == 4
        %       Plot Stability values - This can be removed, it is a bit messy
        %           plot([0 24],[-0.04 -0.04],'Color',stabColor(1,:))
        %           plot([0 24],[0 0],'Color',stabColor(3,:))
        %           plot([0 24],[0.25 0.25],'Color',stabColor(5,:))
        %TODO: Fix this so the arrows and text are in the best location or
        %just remove it
        text(0.5,2.5,' $ \uparrow $ Extremely ...
            Stable','VerticalAlignment','bottom','Color','k')
        text(0.5,-2.5,' $ \downarrow $ Extremly ...
            Unstable','VerticalAlignment','top','Color','k')
    end
    hold off
    title(diurTitleText(i))
    ylabel(diurYlabelText(i))
    xlabel('Hours')
    axis tight
    hFig8.CurrentAxes.XTick = 0:24;
end

%% Figure 9 - Alpha Vs Windspeed
hFig9 = figure(9);
set(hFig9,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])
alphaShear = mast.AlphaShear;
for i = 1:5
    subplot(2,3,i)
    stabInd = stabCond == i & refU > 0 & ¬isnan(refU) & ¬isnan(alphaShear) & ¬...
        isinf(abs(alphaShear)); % Can not fit with non positive values or NaN's
    try
        f = fit(refU(stabInd),alphaShear(stabInd),'power2'); % Fits Power Law of the ...
            form f(x) = a*x^b+c
        plot(f,refU(stabInd),alphaShear(stabInd))
        %           plot(refU(stabInd),alphaShear(stabInd),'bx')

        coeffNum = coeffvalues(f);
        legend('Data',[num2str(coeffNum(1),2), '\timesx^{',num2str(coeffNum(2),2),'} ...
            +',num2str(coeffNum(3),2)], 'Interpreter','latex')
    catch
        warning('Shear inversion for %s condition', stabText{i})
        plot(refU(stabInd), alphaShear(stabInd),'b.')
    end
    title(stabText(i))
    xlabel(['$ U_{',num2str(sourceZ),'m} [m/s] $'])
    ylabel('$ \alpha $ ')

    try
        if min(alphaShear(stabInd)) < 0 && min(alphaShear(stabInd)) > -2;
            yMin = min(alphaShear(stabInd));
        elseif min(alphaShear(stabInd)) < 0 && min(alphaShear(stabInd)) < -2
            yMin = -2;
        else
            yMin = 0;
        end

        if max(alphaShear) > 2
            yMax = 2;
        else
            yMax = max(alphaShear);
        end

        ylim([yMin yMax]);

        if max(refU) > 25
            xlim([0 25])
        else
            xlim([0 max(refU)])
        end
    catch
        yMin = 0;
        yMax = 2;
        ylim([yMin yMax]);
        xlim([0 max(refU)])
    end
end

%% Figure 10 - Sectorwise Velocity Profiles
% Only plot 4 or 6 sectors at a time, any more than this and the graphs become
% too messy

% Create suitable height for velocity profile
if max(Zs) ≥ 100 && max(Zs) < 150
    ZMax = 150;
elseif max(Zs) ≥ 150
    ZMax = 200;
else
    ZMax = 100;
end

zProfile = profiles.Z';
zInd = zProfile ≤ ZMax & zProfile≥ 1;

if length(shearSectors2Plot) == 4
    a = 2;
    b = 2;
elseif length(shearSectors2Plot) == 6
    a = 2;
    b = 3;
else
   error('Error in shear profile sectors to plot')
end

velSecAllHeightsSec2Plot = velSecAllHeights(:,shearSectors2Plot,:); % Only extract ...
    sectors that you need

hFig10 = figure(10);
```

```
set(hFig10,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])
for i = 1:length(shearSectors2Plot)
    subplot(a,b,i)
    hold on
    for j = 1:5
        Uprofile = reshape(profiles.Uz(shearSectors2Plot(i),j,2:end) ...
            ,length(profiles.Uz(shearSectors2Plot(i),j,2:end)),1);
        plot(Uprofile(zInd), zProfile(zInd),'Color',stabColor(j,:) ,'LineWidth',3);
        % Plot the observed data as crosses
        if strcmpi(shearScaling, 'Yes') % To not plot the scaled value as an observed value
            plot(reshape(velSecAllHeightsSec2Plot(j,i,1:end-1),1,  ...
                length(Zs)-1),Zs(1:end-1),'x','Color', stabColor(j,:),'LineWidth',2.5) ...
                % This plots the observed data as crosses
        else
            plot(reshape(velSecAllHeightsSec2Plot(j,i,:),1, ...
                length(Zs)),Zs,'x','Color',stabColor(j,:),'LineWidth',2.5)
        end
    end
    hold off
    title(['Sector ',num2str(shearSectors2Plot(i))])
    xlabel('$ U [m/s] $')
    ylabel('Height AGL [m]')
    %     %ToDo: Sort out this legend
    %     if i == 1
    %     legend()
    %     end
end

%% Figure 11 - Sectorwise PotenTemp Profiles
PotenTempSecAllHeightsSec2Plot = PotenTempSecAllHeights(:,shearSectors2Plot,:); % Only ...
    extract sectors that you need

hFig11 = figure(11);
set(hFig11,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])
for i = 1:length(shearSectors2Plot)
    subplot(a,b,i)
    hold on
    for j = 1:5
        potenTempProfile = reshape(profiles.potenTempZ(shearSectors2Plot(i),j,2:end), ...
            length(profiles.potenTempZ(shearSectors2Plot(i),j,2:end)),1);
        plot(potenTempProfile(zInd), zProfile(zInd),'Color',stabColor(j,:),'LineWidth',3);
        % Plot the observed data as crosses
        plot(reshape(PotenTempSecAllHeightsSec2Plot(j,i,:), 1,length(Zt)),Zt, ...
            'x','Color',stabColor(j,:),'LineWidth',2.5)
    end
    hold off
    title(['Sector ',num2str(shearSectors2Plot(i))])
    xlabel('$ \theta $ [K]')
    ylabel('Height AGL [m]')
    %     %ToDo: Sort out this legend
    %     if i == 1
    %     legend()
    %     end
end

%% Figure 12 - Turbulence Model Profiles
hFig12 = figure(12);
set(hFig12,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])

if k_eModel == 3 || k_eModel == 4
    turbProf = {reshape(profiles.kZ(profSec,:,zInd),[5 length(zProfile(zInd))])
        reshape(profiles.epsilonZ(profSec,:,zInd),[5 length(zProfile(zInd))])
        reshape(profiles.omegaZ(profSec,:,zInd),[5 length(zProfile(zInd))])
        reshape(turbModelConstants.F_Ce3(profSec,:,zInd),[5 length(zProfile(zInd))])};
else
    turbProf = {reshape(profiles.kZ(profSec,:,zInd),[5 length(zProfile(zInd))])
        reshape(profiles.epsilonZ(profSec,:,zInd),[5 length(zProfile(zInd))])
        reshape(profiles.omegaZ(profSec,:,zInd),[5 length(zProfile(zInd))])
        table2array(turbModelConstants.Constants{1,4})*ones([5 length(zProfile(zInd))])};
end

for i = 1:4
    subplot(2,2,i)
    hold on
    for j = 1:5
        plot(turbProf{i}(j,:),zProfile(zInd),'Color',stabColor(j,:),'LineWidth',3);
    end
    hold off
    xlabel(turbModelxLabel(i))
    ylabel('$ z [m] $')
end

%% Figure 13 - Velocity Vs. Stability Class
hFig13 = figure(13);
set(hFig13,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])
colormap(stabColor(1:end-1,:)) %Remove black color

bar(velocityStab(end,:)',flipud(100*velocityStab(1:end-1,:))',1,'stacked') % Bar chart ...
    that shows the percentage for each stability class diurnally
title(['Windspeed Vs. Stability ', mastName, ' ', dateRangeStr])
ylabel('\% Stability Class')
ylim([0 100])
xlabel(['$ U_{',num2str(sourceZ),'m} [m/s] $'])
axis tight
legend(stabText,'Location','bestoutside','Interpreter','latex')

%% Figure 14 - Comparison of stability calculation methods
hFig14 = figure(14);
set(hFig14,'Position',[1 1 scrsz(3) scrsz(4)/1.5],'Color',[1 1 1])
colormap(stabColor(1:end-1,:)) %Remove black color

if strcmpi(TiAvail,'Yes')
    pieTitles = {'Ri','MOL','Ti \& Shear'};
else
    pieTitles = {'Ri','MOL','Shear'};
end
```

```

for i = 1:3
    switch i
        case 1 % Ri
            exUnstable2 = sum(mast.ConditionRi == 1)/sum(¬isnan(mast.ConditionRi));
            unstable2 = sum(mast.ConditionRi == 2)/sum(¬isnan(mast.ConditionRi));
            neutral2 = sum(mast.ConditionRi == 3)/sum(¬isnan(mast.ConditionRi));
            stable2 = sum(mast.ConditionRi == 4)/sum(¬isnan(mast.ConditionRi));
            exStable2 = sum(mast.ConditionRi == 5)/sum(¬isnan(mast.ConditionRi));
        case 2 % MOL
            exUnstable2 = sum(mast.ConditionMol == 1)/sum(¬isnan(mast.ConditionMol));
            unstable2 = sum(mast.ConditionMol == 2)/sum(¬isnan(mast.ConditionMol));
            neutral2 = sum(mast.ConditionMol == 3)/sum(¬isnan(mast.ConditionMol));
            stable2 = sum(mast.ConditionMol == 4)/sum(¬isnan(mast.ConditionMol));
            exStable2 = sum(mast.ConditionMol == 5)/sum(¬isnan(mast.ConditionMol));
        case 3 % Shear/Ti
            exUnstable2 = sum(mast.ConditionTiShear == ...
                1)/sum(¬isnan(mast.ConditionTiShear));
            unstable2 = sum(mast.ConditionTiShear == 2)/sum(¬isnan(mast.ConditionTiShear));
            neutral2 = sum(mast.ConditionTiShear == 3)/sum(¬isnan(mast.ConditionTiShear));
            stable2 = sum(mast.ConditionTiShear == 4)/sum(¬isnan(mast.ConditionTiShear));
            exStable2 = sum(mast.ConditionTiShear == 5)/sum(¬isnan(mast.ConditionTiShear));
    end
    y = 100*[exUnstable2, unstable2, neutral2, stable2, exStable2];
    subplot(1,3,i)
    hPie2 = pie(y);
    title(pieTitles{i})
    hPie2(1).FaceColor = stabColor(1,:);
    hPie2(3).FaceColor = stabColor(2,:);
    hPie2(5).FaceColor = stabColor(3,:);
    hPie2(7).FaceColor = stabColor(4,:);
    hPie2(9).FaceColor = stabColor(5,:);

    if i ≠ 3
        legend1 = legend(stabText);
        set(legend1,...
        'Position',[0.0107681274692211 0.448399554666699 0.104809760854814 ...
            0.142384102012938]);
    end
end

%% Figure 15  - Wind speed vs. freq
hFig15 = figure(15);
set(hFig15,'Position',[1 1 scrsz(3)/1.5 scrsz(4)/1.5],'Color',[1 1 1])

edges = 0:0.5:ceil(max(sourceU));
hold on
for i = 1:5
  [N] = histcounts(sourceU(stabCond == i),edges,'Normalization', 'probability');
  N(N==0) = nan;
  plot(edges(2:end)-0.25,100.*N,'-','Color',stabColor(i,:),'LineWidth',3)
end
hold off
ylabel('Frequency \% ')
xlabel(['$ U_{',num2str(sourceZ),'m} [m/s] $'])
axis tight
legend(stabText,'Location','bestoutside','Interpreter','latex')

%% Create figure output

figs = [hFig1, hFig2, hFig3, hFig4, hFig5, hFig6, hFig7, hFig8, hFig9, hFig10, hFig11, ...
    hFig12, hFig13, hFig14, hFig15];

function [mastStruct] = dataSplit(mast,stabCond)
%% dataSplit
% Split data into stability based timeseries based on the Richardson Number

mast.TimeStamp = datestr(mast.TimeStamp,'yyyy-mm-dd HH:MM');
mastExtremelyUnstable = mast(stabCond == 1,:);
mastUnstable = mast(stabCond == 2,:);
mastNeutral = mast(stabCond == 3,:);
mastStable = mast(stabCond == 4,:);
mastExtremelyStable = mast(stabCond == 5,:);

% mastStruct = struct('Total',mast,'Stable',mastStable,'unStable', ...
    mastUnstable,'Neutral',mastNeutral,'unStableAndNeutral',mastUnstableAndNeutral);
mastStruct = struct('Total',mast,'ExtremelyStable',mastExtremelyStable, ...
    'Stable',mastStable,'Neutral', mastNeutral,'Unstable',mastUnstable, ...
    'extremelyUnstable',mastExtremelyUnstable);

function [] = ...
    imageSave(figs,mastName,dateRangeStr,shearSectors2Plot,dirPath,TiAvail,profSec)
%% imageSave
% Saves all active image handles in the selected folder

% Create date range and sector string that works as a figure name
dateRangeStr = [dateRangeStr(7:10),dateRangeStr(4:5),dateRangeStr(1:2),'-', ...
    dateRangeStr(26:29),dateRangeStr(23:24),dateRangeStr(20:21)];

figNames = {[mastName,' Stability Classification ',dateRangeStr,'.png']
            [mastName,' Sectorwise Stability Classification ',dateRangeStr,'.png']
            [mastName,' Stability Rose ',dateRangeStr,'.png']
            [mastName,' Ux Vs Uy ',dateRangeStr,'.png']
            [mastName,' Ti Vs Windspeed ',dateRangeStr,'.png']
            [mastName,' Diurnals ',dateRangeStr,'.png']
            [mastName,' Diurnal Stability Classification ',dateRangeStr,'.png']
            [mastName,' Diurnal Richardson and MOL ',dateRangeStr,'.png']
            [mastName,' Shear Vs Windspeed ',dateRangeStr,'.png']
            [mastName,' Velocity Profile Sectors [',int2str(shearSectors2Plot(:)'),'] ...
                ',dateRangeStr,'.png']
            [mastName,' Temperature Profile Sectors ...
                [',int2str(shearSectors2Plot(:)'),'] ',dateRangeStr,'.png'];
            [mastName,' Turbulence Model Profiles Sector ',int2str(profSec),' ...
                ',dateRangeStr,'.png']
            [mastName,' Stability Vs Windspeed ',dateRangeStr,'.png']
            [mastName,' Stability Classifictaion Comparison ',dateRangeStr,'.png']
```

```
            [mastName,' Windspeed Vs Frequency ',dateRangeStr,'.png']};

for i=1:length(figs)
    if strcmpi(TiAvail,'No') && i ≠ 5
        print(figs(i),[dirPath,'\',figNames{i}],'-dpng','-r0')
        disp(['Figure ' figNames{i},' Saved'])
    elseif strcmpi(TiAvail,'Yes')
        print(figs(i),[dirPath,'\',figNames{i}],'-dpng','-r0')
        disp(['Figure ' figNames{i},' Saved'])
    end
end

function [] = dataSave(mastName, dateRangeStr,dirPath, mastStruct,profiles, ...
    turbModelConstants, sectorTables,diurnalProfiles,stabClass, ...
    outputType,sectorTableContent,velSecAllHeights)  %#ok<INUSL>
%% Save Data
% Save .mat or .txt file to selected dir

dateRangeStr = [dateRangeStr(7:10),dateRangeStr(4:5),dateRangeStr(1:2),'-', ...
    dateRangeStr(26:29),dateRangeStr(23:24),dateRangeStr(20:21)];
fileName = [dirPath,'\',mastName,'_',dateRangeStr,'.mat'];

mastTableContent = ...
    {'Total','ExtremelyStable','Stable','Neutral','Unstable','extremelyUnstable'};
mastTable = struct2cell(mastStruct);
if strcmpi(outputType,'mat')
    save(fileName,'mastStruct','profiles','turbModelConstants', ...
        'sectorTables','diurnalProfiles','stabClass','velSecAllHeights')
else
    for i = 1:6
        writetable(mastTable{i},[dirPath,'\',mastTableContent{i},'_',dateRangeStr,'.txt'])
    end

    for i = 1:length(sectorTables)
        writetable(cell2table(sectorTables{i}), ...
            [dirPath,'\',sectorTableContent{i},'_',dateRangeStr,'.txt'])
    end
end
```

C.2 stabilityRose.m

```
function [figure_handle,Table] = stabilityRose(direction,speed,figHandle,varargin)
    %% StabilityRose
    %  Draw a Stability Rose knowing direction and condition number
    %  This is an edit of the original windrose code from Daniel Pereira - ...
        daniel.pereira.valades@gmail.com 22/06/2015
    %  It is implimented in the dataAnalysis.m code
    %
    %    Condition        |  Condition Number
    % ---------------------------------------------------
    % Extremely unstable |        1
    % Unstable           |        2
    % Neutral            |        3
    % Stable             |        4
    % Extremely stable   |        5
    %
    %
    % -----------------------------------------------------------------------
    % Revised: Hendri Breedt <u10028422@tuks.co.za>
    % Date: 09/11/2017
    % Version: 00 - Public release

    %% Check funciton call
    if nargin<2
        error('stabilityRose needs at least two inputs');        % function needs 2 ...
            input arguments
    elseif mod(length(varargin),2)≠0                           % If varargin are not paired
        if (length(varargin)==1 && isstruct(varargin{1}))    % Could be a single ...
            structure with field names and field values.
            varargin = reshape([fieldnames(varargin{1}) ...
                struct2cell(varargin{1})]',1,[]); % Create varargin as if they were ...
                separate inputs
        elseif (length(varargin)==1 && iscell(varargin{1})) % Could be a single cell ...
            array with all the varargins
            varargin = reshape(varargin{1},1,[]);            % Reshape just in case, and ...
                create varargin as if they were separate inputs.
        else
            error('Inputs must be paired: ...
                stabilityRose(Speed,Direction,''PropertyName'',PropertyValue,...)'); % ...
                If not any of the two previous cases, error
        end
    elseif ¬isnumeric(speed) || ¬isnumeric(direction)        % Check that speed and ...
        direction are numeric arrays.
        error('Speed and Direction must be numeric arrays.');
    elseif ¬isequal(size(speed),size(direction))             % Check that speed and ...
        direction are the same size.
        error('Speed and Direction must be the same size.');
    end

%% Default parameters
SCS               = get(0,'screensize');

CeteredIn0        = true;
ndirections       = 36;
FrequenciesRound  = 1;
NFrequencies      = 5;
WindSpeedRound    = [];
NSpeeds           = 5;
circlemax         = [];
FreqLabelAngle    = 60;
TitleString       = {'Wind Rose';' '};
lablegend         = '';
colorfun          = 'jet';
height            = min(SCS(3:4))*2/3;
width             = min(SCS(3:4))*2/3;
figcolor          = 'w';
TextColor         = 'k';
label.N           = 'N';
label.S           = 'S';
label.W           = 'W';
label.E           = 'E';
titlefontweight   = 'bold';
legendvariable    = 'W_S';
RefN              = 90;
RefE              = 0;
min_radius        = 1/15;
LegendType        = 2;
MenuBar           = 'figure';
ToolBar           = 'figure';
colors            = [];
inverse           = false;
vwinds            = [];
scalefactor       = 1;
axs               = [];

%% User-.specified parameters

for i=1:2:numel(varargin)
    switch lower(varargin{i})
        case 'centeredin0'
            CeteredIn0       = varargin{i+1};
        case 'ndirections'
            ndirections      = varargin{i+1};
        case 'freqround'
            FrequenciesRound = varargin{i+1};
        case 'nfreq'
            NFrequencies     = varargin{i+1};
```

```
        case 'speedround'
            WindSpeedRound   = varargin{i+1};
        case 'nspeeds'
            NSpeeds          = varargin{i+1};
        case 'freqlabelangle'
            FreqLabelAngle   = varargin{i+1};
        case 'titlestring'
            TitleString      = varargin{i+1};
        case 'lablegend'
            lablegend        = varargin{i+1};
        case 'cmap'
            colorfun         = varargin{i+1};
        case 'height'
            height           = varargin{i+1};
        case 'width'
            width            = varargin{i+1};
        case 'figcolor'
            figcolor         = varargin{i+1};
        case 'textcolor'
            TextColor        = varargin{i+1};
        case 'min_radius'
            min_radius       = varargin{i+1};
        case 'maxfrequency'
            circlemax        = varargin{i+1};
        case 'titlefontweight'
            titlefontweight  = varargin{i+1};
        case 'legendvariable'
            legendvariable   = varargin{i+1};
        case 'legendtype'
            LegendType       = varargin{i+1};
        case 'inverse'
            inverse          = varargin{i+1};
        case 'labelnorth'
            label.N          = varargin{i+1};
        case 'labelsouth'
            label.S          = varargin{i+1};
        case 'labeleast'
            label.E          = varargin{i+1};
        case 'labelwest'
            label.W          = varargin{i+1};
        case 'labels'
            label.N          = varargin{i+1}{1};
            label.S          = varargin{i+1}{2};
            label.E          = varargin{i+1}{3};
            label.W          = varargin{i+1}{4};
        case 'menubar'
            MenuBar          = varargin{i+1};
        case 'toolbar'
            ToolBar          = varargin{i+1};
        case 'scalefactor'
            scalefactor      = varargin{i+1};
        case 'vwinds'
            k = any(arrayfun(@(x) strcmpi(x,'nspeeds'),varargin));
            if k
                warning('''vwinds'' and ''nspeeds'' have been specified. The value for ...
                    ''nspeeds'' wil be omitted');
            end
            vwinds           = varargin{i+1};
        case 'colors'
            k = any(arrayfun(@(x) strcmpi(x,'nspeeds'),varargin)) + any(arrayfun(@(x) ...
                strcmpi(x,'vwinds'),varargin));
            if ¬k
                error('To specify ''colors'' matrix, you need to specify the number of ...
                    speed bins ''nspeeds'' or the speeds to be used ''vwinds''');
            end
            k = any(arrayfun(@(x) strcmpi(x,'cmap'),varargin));
            if k
                warning('Specified CMAP is not being used, since ''colors'' argument ...
                    has been set by user');
            end
            colors           = varargin{i+1};
        case 'anglenorth'
            k = any(arrayfun(@(x) strcmpi(x,'angleeast'),varargin));
            if ¬k
                error('Reference angles need to be specified for AngleEAST and ...
                    AngleNORTH directions');
            end
        case 'angleeast'
            k = find(arrayfun(@(x) strcmpi(x,'anglenorth'),varargin));
            if isempty(k)
                error('Reference angles need to be specified for AngleEAST and ...
                    AngleNORTH directions');
            else
                RefE         = varargin{i+1};
                RefN         = varargin{k+1};
            end
            if abs(RefN-RefE)≠90
                error('The angles specified for north and east must differ in 90 degrees');
            end
        case 'axes'
            axs = varargin{i+1};
        otherwise
            error([varargin{i} ' is not a valid property for stabilityRose function.']);
    end
end

if ¬isempty(vwinds)
    vwinds  = unique(reshape(vwinds(:),1,[]));    % ?? Should have used vwinds  = ...
        unique([0 reshape(vwinds(:),1,[])]); to ensure that values in the interval [0 ...
        vmin) appear. If user want hat range to appear, 0 must be included.
    NSpeeds = length(vwinds);
end

if ¬isempty(colors)
    if ¬isequal(size(colors),[NSpeeds 3])
```

```
        error('colors must be a nspeeds by 3 matrix');
    end
    if any(colors(:)>1) || any(colors(:)<0)
        error('colors must be in the range 0-1');
    end
end

if inverse
    colorfun = regexprep(['inv' colorfun],'invinv','');
    colors   = flipud(colors);
end

% Create Custom colormap for 5 stability cases
colors = [0.857142857142857,0,0;1,0.857142857142857,0;0.571428571428571,1, ...
    0.428571428571429;0,0.714285714285714,1;0,0,0.714285714285714];
% colors = flipud(colors);

speed           = reshape(speed,[],1);                          % Convert ...
    wind speed into a column vector
direction       = reshape(direction,[],1);                      % Convert ...
    wind direction into a column vector
NumberElements  = numel(direction);                             % Coun the ...
    actual number of elements, to consider winds = 0 when calculating frequency.
dir             = mod((RefN-direction)/(RefN-RefE)*90,360);     % Ensure ...
    that the direction is between 0 and 360
dir             = dir(speed>0);                                 % Wind = 0 ...
    does not have direction, so it cannot appear in a wind rose, but the number of ...
    appeareances must be considered.
speed           = speed(speed>0);                               % Only show ...
    winds higher than 0. See comment before.

% if isempty(axs) % If no axes were specified, create a new figure
%     figure_handle = ...
    figure('color',figcolor,'units','pixels','position',[SCS(3)/2-width/2 ...
    SCS(4)/2-height/2 width height],'menubar',MenuBar,'toolbar',ToolBar);
% else % If axes are specified, use the figure in which the axes are located
%     figure_handle = get(axs,'parent');
% end

figure_handle = figHandle;
%% Bin Directions
N     = linspace(0,360,ndirections+1);                          % Create ...
    ndirections direction intervals (ndirections+1 edges)
N     = N(1:end-1);                                             % N is the ...
    angles in which direction bins are centered. We do not want the 360 to appear, ...
    because 0 is already appearing.
n     = 180/ndirections;                                        % Angle that ...
    should be put backward and forward to create the angular bin, 1st centered in 0
if ¬CeteredIn0                                                  % If user ...
    does not want the 1st bin to be centered in 0
    N = N+n;                                                    % Bin goes ...
        from 0 to 2n (N to N+2n), instead of from -n to n (N-n to N+n), so Bin is not ...
        centered in 0 (N) angle, but in the n (N+n) angle
end

%% Bin intensities
if isempty(vwinds)                                              % If user ...
    did not specify the wind speeds he/she wants to show
    if ¬isempty(WindSpeedRound)                                 % If user ...
        did specify the rounding value
        if isempty(NSpeeds); NSpeeds = 6; end                   % Default ...
            value for NSpeeds if not user-specified
        vmax      = ceil(max(speed)/WindSpeedRound)*WindSpeedRound;      % Max wind ...
            speed rounded to the nearest whole multiple of WindSpeedRound (Use round or ...
            ceil as desired)
                    if vmax==0; vmax=WindSpeedRound; end;       % If max ...
                        wind speed is 0, make max wind to be WindSpeedRound, so wind ...
                        speed bins are correctly shown.
        vwinds    = linspace(0,vmax,NSpeeds);                   % Wind ...
            speeds go from 0 to vmax, creating the desired number of wind speed intervals
    else                                                        % If user ...
        did nor specify the rounding value
        figure2 = figure('visible','off'); plot(speed);         % Plot wind ...
            speed
        vwinds = get(gca,'ytick'); delete(figure2);             % Yaxis will ...
            automatically make divisions for us.
        if ¬isempty(NSpeeds)                                    % If a ...
            number of speeds are specified
            vwinds = linspace(min(vwinds),max(vwinds),NSpeeds); % create a ...
                vector with that number of elements, distributed along the plotted ...
                windspeeds.
        end
    end
end

%% Histogram in each direction + Draw
count     = PivotTableCount(N,n,vwinds,speed,dir,NumberElements);         % For each ...
    direction and for each speed, value of the radius that the windorose must reach ...
    (Accumulated in speed).

if isempty(circlemax)                                           % If no max ...
    frequency is specified
    circlemax = ceil(max(max(count))/FrequenciesRound)*FrequenciesRound;  % Round ...
        highest frequency to closest whole multiple of theFrequenciesRound  (Use round ...
        or ceil as desired)
end

min_radius = min_radius*circlemax;                              % The ...
    minimum radius is initially specified as a fraction of the circle max, convert it ...
    to absolute units.
isaxisempty = isempty(axs);                                     % ...
    isaxisempty will allow us to identify whether the axes where specified or not, ...
    because we are going to assign in the next line a value, so axs will be never again ...
    empty.
```

```
[color,axs] = ...
    DrawPatches(N,n,vwinds,count,colorfun,figcolor,min_radius,colors,inverse,axs); % ...
    Draw the windrose, knowing the angles, the range for each direction, the speed ...
    ranges, the count (frequency) values, the colormap used and the colors used.

axis off;                                                          % turn axis off
axis equal;                                                        % equal axis
circlemax = circlemax/max(eps,scalefactor);                        % If a scale ...
    factor is specified, embiggen the circelmax (which defines x and y limits)

if isaxisempty; set(axs,'position',[0 0 1 1]); end                 % If no axes ...
    were specified, set the axes position to fill the whole figure.
%% Constant frequecy circles and x-y axes + Draw + Labels

[x,y]     = cylinder(1,50); x = x(1,:); y = y(1,:);               % Get x and ...
    y for a unit-radius circle
circles   = linspace(0,circlemax,NFrequencies+1); circles = circles(2:end);% Radii of ...
    the circles that must be drawn (frequencies). We do not want to spend time drawing ...
    radius=0.

radius      = circles     + min_radius;                            % for each ...
    circle, add the minimum radius
radiusmax   = circlemax   + min_radius;

radius      = radius      * scalefactor;                           % scale up ...
    or down the radius values.
radiusmax   = radiusmax   * scalefactor;
min_radius  = min_radius  * scalefactor;

if ¬isaxisempty % If axis are specified (not empty)
    h=fill(x'*radiusmax,y'*radiusmax,figcolor);                    % create a ...
        background circle
    hAnnotation = get(h,'Annotation');                             % get ...
        annotation from the circle
    hLegendEntry = get(hAnnotation','LegendInformation');          % get legend ...
        information from the circle
    set(hLegendEntry,'IconDisplayStyle','off')                     % remove the ...
        cricle from the legened information.
    uistack(h,'bottom');                                           % the circle ...
        must be placed below everything.
end
plot(axs,x'*radius,y'*radius,':','color',TextColor);               % Draw ...
    dooted circle lines
plot(axs,x*radiusmax,y*radiusmax,'-','color',TextColor);           % Redraw ...
    last circle line in solid style

axisangles = 0:30:360; axisangles = axisangles(1:end-1);           % Angles in ...
    which to draw the radial axis (trigonometric reference)
R = [min_radius;radiusmax];                                        % radius
plot(axs,R*cosd(axisangles),R*sind(axisangles),':','color',TextColor);    % Draw ...
    radial axis, in the specified angles

FrequecyLabels(circles,radius,FreqLabelAngle,TextColor);           % Display ...
    frequency labels
CardinalLabels(radiusmax,TextColor,label);                         % Display N, ...
    S, E, W

xlim(axs,[-radiusmax radiusmax]/scalefactor);                      % Set limits
ylim(axs,[-radiusmax radiusmax]/scalefactor);

%% Title and Legend
title(TitleString,'color',TextColor,'fontweight',titlefontweight);    % Display a ...
    title
if isaxisempty; set(axs,'outerposition',[0 0 1 1]); end            % Check that ...
    the current axis fills the figure, only if axis were not specified
if LegendType==2                                                   % If legend ...
    type is box:
    leyenda = CreateLegend(vwinds,lablegend,legendvariable,inverse);   % Create a ...
        legend cell string
    % This overwrites the above section to create the legend we need
%     leyenda(2:end) = {'Extremely Unstable', 'Unstable', 'Neutral', 'Stable', ...
    'Extremely Stable'};
    leyenda(2:end) = {'Extremely Stable', 'Stable', 'Neutral', 'Unstable', 'Extremely ...
        Unstable'};

    l       = legend(axs,leyenda,'location','northeast','Interpreter','latex'); ...
                 % Display the legend wherever (position is corrected)
    if isaxisempty                                                 % If axis ...
        were not specified
        PrettyLegend(l,TextColor);                                 % Display ...
            the legend in a good position
    else                                                           % If axis ...
        were specified
        set(l,'textcolor',TextColor,'color',figcolor);             % change ...
            only the legend colour (text and background)
    end
elseif LegendType==1                                               % If legend ...
    type is colorbar
    caxis(axs,[vwinds(1) vwinds(end)]);                            % Set ...
        colorbar limits
    colormap(axs,interp1(vwinds,color,linspace(min(vwinds),max(vwinds),256))); % set ...
        colorbar colours (colormap)
    colorbar('YTick',vwinds);                                      % The values ...
        shown in the colorbar are the intenisites.
end


%% Outputs
[count,speeds,directions,Table] = CreateOutputs(count,vwinds,N,n,RefN,RefE); % Create ...
    output arrays and tables.

function count = PivotTableCount(N,n,vwinds,speed,dir,NumberElements)
    count   = zeros(length(N),length(vwinds));
    for i=1:length(N)
        d1 = mod(N(i)-n,360);                                      % Direction ...
            1 is N-n
```

```
        d2 = N(i)+n;                                                    % Direction ...
            2 is N+n
        if d1>d2                                                        % If ...
            direction 1 is greater than direction 2 of the bin (d1 = -5 = 355, d2 = 5)
            cond = or(dir≥d1,dir<d2);                                  % The ...
                condition is satisfied whenever d≥d1 or d<d2
        else                                                            % For the ...
            rest of the cases,
            cond = and(dir≥d1,dir<d2);                                 % Both ...
                conditions must be met for the same bin
        end
%           counter     = histc(speed(cond),vwinds);                    %# REMOVED ...
    2015/Jun/22  % If vmax was for instance 25, counter will have counts for these ...
    intervals: [≥0 y <5] [≥5 y <10] [≥10 y <15] [≥15 y <20] [≥20 y <25] [≥25]
        counter     = histc(speed(cond),[vwinds(:)' inf]);              %# ADDED ...
            2015/Jun/22: Consider the wind speeds greater than max(vwinds), by adding ...
            inf into the histogram count
        counter     = counter(1:length(vwinds));                        %# ADDED ...
            2015/Jun/22: Crop the resulting vector form histc, so as it has only ...
            length(Vwinds) elements
        if isempty(counter); counter = zeros(1,size(count,2)); end      % If counter ...
            is empty for any reason, set the counts to 0.
        count(i,:) = cumsum(counter);                                   % Computing ...
            cumsum will make count to have the counts for [<5] [<10] [<15] [<20] [<25] ...
            [≥25] (cumulative count, so we have the radius for each speed)
    end
    count = count/NumberElements*100;                                   % Frequency ...
        in percentage

function [color,axs] = ...
    DrawPatches(N,n,vwinds,count,colorfun,figcolor,min_radius,colors,inverse,axs)
    if isempty(colors)
        inv = strcmp(colorfun(1:3),'inv');                                  % INV = ...
            First three letters in cmap are inv
        if inv; colorfun = colorfun(4:end); end                             % if ...
            INV, cmap is the rest, excluding inv
        color = feval(colorfun,256);                                        % Create ...
            color map
        color = interp1(linspace(1,length(vwinds),256),color,1:length(vwinds));% Get ...
            the needed values.
        if inv; color = flipud(color); end;                                 % if ...
            INV, flip upside down the colormap
    else
        color = colors;
    end
    if isempty(axs)
        plot(0,0,'.','color',figcolor, 'markeredgecolor',figcolor, ...
            'markerfacecolor',figcolor); % This will create an empty legend entry.
        axs = gca;
    else
        plot(axs,0,0,'.','color',figcolor, 'markeredgecolor',figcolor, ...
            'markerfacecolor',figcolor); % This will create an empty legend entry.
    end
    set(gcf,'currentaxes',axs);
    hold on; axis square; axis off;

    if inverse                                                      % If wind ...
        speeds are shown in inverse way (slowest is outside)
        count          = [count(:,1) diff(count,1,2)];              % De-compose ...
            cumsum
        count          = cumsum(fliplr(count),2);                   % Cumsum ...
            inverting count.
    end

    for i=1:length(N)                                               % For every ...
        angle
        for j=length(vwinds):-1:1;                                  % For every ...
            wind speed range (last to first)
            if j>1                                                  % If the ...
                wind speed range is not the first
                r(1) = count(i,j-1);                                % the lower ...
                    radius of this bin is the upper radius of the one with lower speeds
            else                                                    % If the ...
                wind speed range is the first
                r(1) = 0;                                           % the lower ...
                    radius is 0
            end
            r(2)  = count(i,j);                                     % The upper ...
                radius is the cumulative count for this angle and this speed range
            r     = r+min_radius;                                   % We have to ...
                sum the minimum radius.

            alpha = linspace(-n,n,100)+N(i);                        % these are ...
                the angles for which the bins are plotted
            x1    = r(1) * sind(fliplr(alpha));                     % convert 1 ...
                radius and 100 angles into a line, x
            y1    = r(1) * cosd(fliplr(alpha));                     % and y
            x     = [x1 r(2)*sind(alpha)];                          % Create ...
                circular sectors, completing x1 and y1 with the upper radius.
            y     = [y1 r(2)*cosd(alpha)];
            fill(x,y,color(j,:),'edgecolor',hsv2rgb(rgb2hsv(color(j,:)).*[1 1 0.7])); % ...
                Draw them in the specified coloe. Edge is slightly darker.
        end
    end

function FrequecyLabels(circles,radius,angulo,TextColor)
    s = sind(angulo); c = cosd(angulo);                                 % Get the ...
        positions in which labels must be placed
    if c>0; ha = 'left';   elseif c<0; ha = 'right'; else ha = 'center'; end % ...
        Depending on the sign of the cosine, horizontal alignment should be one or another
    if s>0; va = 'bottom'; elseif s<0; va = 'top';    else va = 'middle'; end % ...
        Depending on the sign of the sine  , vertical   alignment should be one or another
    for i=1:length(circles)
        text(radius(i)*c,radius(i)*s,[num2str(circles(i)) ...
            '\%'],'HorizontalAlignment',ha,'verticalalignment',va,'color',TextColor); % ...
            display the labels for each circle
```

```
    end
    rmin = radius(1)-abs(diff(radius(1:2)));
    if rmin>0
        if c>0; ha = 'right'; elseif c<0; ha = 'left';   else ha = 'center'; end % ...
            Depending on the sign of the cosine, horizontal alignment should be one or ...
            another
        if s>0; va = 'top';   elseif s<0; va = 'bottom'; else va = 'middle'; end % ...
            Depending on the sign of the sine  , vertical  alignment should be one or ...
            another
% text(rmin*c,rmin*s,'0%','HorizontalAlignment', ha,'verticalalignment',va,'color', ...
    TextColor); % display the labels for each circle
    end

function CardinalLabels(circlemax,TextColor,labels)
    text( circlemax,0,[' ' labels.E],'HorizontalAlignment','left'  ...
        ,'verticalalignment','middle','color',TextColor); % East  label
    text(-circlemax,0,[labels.W ' '],'HorizontalAlignment','right' ...
        ,'verticalalignment','middle','color',TextColor); % West  label
    text(0, circlemax,labels.N       ...
        ,'HorizontalAlignment','center','verticalalignment','bottom','color',TextColor); ...
        % North label
    text(0,-circlemax,labels.S       ...
        ,'HorizontalAlignment','center','verticalalignment','top'   ...
        ,'color',TextColor); % South label

function leyenda = CreateLegend(vwinds,lablegend,legendvariable,inverse)
    leyenda = cell(length(vwinds),1);                                   % Initialize ...
        legend cell array
    cont    = 0;                                                        % Initialize ...
        Counter
    if inverse                                                          % If wind ...
        speed order must bu shown in inverse order
        orden = length(vwinds):-1:1;                                    % Set order ...
            backwards
    else                                                                % Else
        orden = 1:length(vwinds);                                       % Set normal ...
            order (cont will be equal to j).
    end

    for j=orden                                                         % Cross the ...
        speeds in the specified direction
        cont = cont+1;                                                  % Increase ...
            counter
        if j==length(vwinds)                                            % When last ...
            index is reached
            string = sprintf('%s %s %g',legendvariable,'\geq',vwinds(j));  % Display ...
                wind ≤ max wind
        else                                                            % For the ...
            rest of the indices
            string = sprintf('%g %s %s < ...
                %g',vwinds(j),'\leq',legendvariable,vwinds(j+1)); % Set v1 ≤ v2 < v1
        end
        string = regexprep(string,'0 \leq','0 <');                      % Replace "0 ...
            ≤" by "0 <", because wind speed = 0 is not displayed in the graph.
        leyenda{length(vwinds)-cont+1} = string;
    end
    if isempty(lablegend); lablegend = ' '; end                         % Ensure ...
        that lablegend is not empty, so windspeeds appear in the right position.
    leyenda = [lablegend; leyenda];                                     % Add the ...
        title for the legend

function PrettyLegend(l,TextColor)
    set(l,'units','normalized','box','off');                            % Do not ...
        display the box
    POS = get(l,'position');                                            % get legend ...
        position (width and height)
    set(l,'position',[0 1-POS(4) POS(3) POS(4)],'textcolor',TextColor);    % Put the ...
        legend in the upper left corner
    uistack(l,'bottom');                                                % Put the ...
        legend below the axis

function [count,speeds,directions,Table] = CreateOutputs(count,vwinds,N,n,RefN,RefE)
    count          = [count(:,1) diff(count,1,2)];                      % Count had ...
        the accumulated frequencies. With this line, we get the frequency for each ...
        single direction and each single speed with no accumulation.
    speeds         = vwinds;                                            % Speeds are ...
        the same as the ones used in the Wind Rose Graph
    directions     = mod(RefN - N'/90*(RefN-RefE),360);                 % Directions ...
        are the directions in which the sector is centered. Convert function reference ...
        to user reference
    vwinds(end+1)  = inf;                                               % Last wind ...
        direction is inf (for creating intervals)

    [directions,i] = sort(directions);                                  % Sort ...
        directions in ascending order
    count          = count(i,:);                                        % Sort count ...
        in the same way.

    wspeeds        = cell(1,length(vwinds)-1);
    for i=1:(length(vwinds)-1)
        if vwinds(i) == 0; s1 = '('; else s1 = '['; end                  % If ...
            vwinds(i) =0 interval is open, because count didn't compute windspeed = 0. ...
            Otherwise, the interval is closed [
        wspeeds{i} = [s1 num2str(vwinds(i)) ' , ' num2str(vwinds(i+1)) ')'];% Create ...
            wind speed intervals, open in the right.
    end

    wdirs = cell(length(directions),1);
    for i=1:length(directions)
        wdirs{i} = sprintf('[%g , %g)',mod(directions(i)-n,360),directions(i)+n); % ...
            Create wind direction intervals [a,b)
    end

    WindZeroFreqency = 100-sum(sum(count));                             % Wind speed ...
        = 0 appears 100-sum(total) % of the time. It does not have direction.
```

```
    WindZeroFreqency = WindZeroFreqency*(WindZeroFreqency/100>eps);        % If ...
        frequency/100% is lower than eps, do not show that value.

    Table            = [{'Frequencies (%)'},{''},{'Stability ...
        Class'},repmat({''},1,numel(wspeeds));'Direction Interval (deg)','Avg. ...
        Direction',wspeeds,'TOTAL';[wdirs num2cell(directions) num2cell(count) ...
        num2cell(sum(count,2))]]; % Create table cell. Ready to xlswrite.
    Table(end+1,:)   = [{'[0 , ...
        360)','TOTAL'},num2cell(sum(count,1)),{sum(sum(count))}]; % the last row is the ...
        total
    Table(end+1,1:2) = {'[0 , 360)', 'Data Unavailable'};                  % add an ...
        additional row showing Wind Speed = 0 on table.
    Table{end,end}   = WindZeroFreqency;                                   % at the end ...
        of the table (last row, last column), show the total number of elements with 0 ...
        speed.
    Table(2,3:7) = {'Extremely Unstable', 'Unstable', 'Neutral', 'Stable', 'Extremely ...
        Stable'};
```

Appendix D

User Defined Functions Code

The UDF codes are included below, coded using C. Three UDF sets are included, one each for neutral, unstable and stable. Each UDF is controlled via the *#define* parameters included at the top of each UDF code.

D.1 Neutral.c

```
#include "udf.h"
#include "math.h"

/* ***************************************************************
   **                          Neutral                           **
   ***************************************************************
   Fluent UDFs for simulating neutral ABL flow

   Control via the defined parameters
   Ensure the solver is in expert mode
   Use compiled UDF method

   Model Axis. xz = inlet/outlet plane, yz = sides, z = AGL, origin at the inlet, ...
       positive in direction of flow and AGL

   C_UDMI - 3  User memory slots, 1User scalar slot
   0 Wall Distance
   1 Cor x
   2 Cor y

   C_UDSI
   0 wallPhi - See description in define cell wall distance

  -------------------------------------------
  Owner: Hendri Breedt <u10028422@tuks.co.za>
  Date: 09/11/2017
  Version: 00 - Public release */

/* Model Constants - DTU */
#define Cmu 0.03
#define vonKarman 0.4
#define Ce1 1.21
#define Ce2 1.92
#define sigma_k 1.0
#define sigma_e 1.3
#define sigma_theta 1.0
#define PrTurb 0.85

/* Model Constants AM */
#define CmuAM 0.033
#define vonKarmanAM 0.42
#define Ce1AM 1.176
/* The rest are the same as the DTU model */

/* Wind speed relations */
#define z0 0.03 /*m*/
#define Cs 0.5 /* Roughness Constant */
#define uStar 0.1439 /* uStar = (vonKarman*uRef)/log(zRef/z0); */
#define ablHeight 1000.0 /* Height of ABL, this is the height for fixed values of all ...
    profiles and sources */

/* Site */
#define globalLat -33.0 /* Latitude of the origin in degrees - This is a dummy value ...
    for confidentiality*/
#define siteElevation 0.0 /* Altitude of site AMSL - If you specify the operating ...
    pressure from site data then DO NOT change this value. */
#define earthRot 0.000072921159 /* Earth rotational speed */
#define offset 477.0 /* Use to control the z value, this is deducted from the mesh z ...
    coordinate.  This is the height AGL of the inlet location of the mesh */
#define offsetY -3000.0 /* This is deducted from the local lattitude in the corliolis ...
    calculation */

/* General */
#define pi 3.141592
#define g -9.80665
#define R 8.3144598 /* Universal Gas Constant - Dry Air */
#define M 0.0289644 /* Molar mass of Earth's air */
```

```
#define Lb -0.0065 /* Standard temperature lapse rate */

/* Operating Conditions - Material Air */
#define presOper 101325 /* Operating Pressure Pa - Internal Solver Pressure. This is ...
    the pressure specified at 0m and for this you can use lowest mast pressure reading */
#define tempOper 288.16 /* Operating Temperature - Internal Solver Standard ...
    Tempearture. This is the temperature based from the lowest measurement height on ...
    the mast. But can be left as the standard value */
#define densOper 1.0919 /* Problem density */
#define Cp 1006.43
#define beta 0.032
#define viscosity 1.7894e-05

/* Initilization */
/*  Due to HAGL variations and Fluent not being able to compute cell distance before ...
    initialiazing we have to manually set the initializе values. These are used for z ...
    values lower than maxZInit, afterwards it returns to the inlt profile values */
#define maxZInit 1000.0 /* Height before using init values from inlet profiles */
#define initVelocity 10.0 /* y velocity */
#define initK 2.0         /* k */
#define initEpsilon 2.0   /* epsilon */

/* ********************** Inlet Velocity ********************** */
DEFINE_PROFILE(inletVelocityNeutral, t, i)
{
    real x[ND_ND];
    real z;
    face_t f;

  begin_f_loop(f, t)
    {
      F_CENTROID(x,f,t);
      z = x[2] + z0 - offset;
      if (z > ablHeight){
        z = ablHeight;
      }
      F_PROFILE(f, t, i) = (uStar/vonKarman)*log(z/z0);
    }
  end_f_loop(f, t)
}

/* ********************** Inlet k ********************** */
DEFINE_PROFILE(inlet_k_Neutral, t, i)
{
  real x[ND_ND];
  face_t f;

  begin_f_loop(f, t)
    {
      F_CENTROID(x,f,t);
      F_PROFILE(f, t, i) = pow(uStar,2.0)/sqrt(Cmu);
    }
  end_f_loop(f, t)
}

/* ********************** Inlet epsilon ********************** */
DEFINE_PROFILE(inlet_e_Neutral, t, i)
{
    real x[ND_ND];
    real z;
    face_t f;

  begin_f_loop(f, t)
    {
      F_CENTROID(x,f,t);
      z = x[2] + z0  - offset;
      if (z > ablHeight){
        z = ablHeight;
      }
      F_PROFILE(f, t, i) = pow(uStar,3.0)/(vonKarman*z);
    }
  end_f_loop(f, t)
}


/*  ********************** Wall Roughness ********************** */

/* Use this if you are using the ABL log law wall function */
DEFINE_PROFILE(wallRoughness,t,i)
{
  real x[ND_ND];
  face_t f;
  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      F_PROFILE(f,t,i) = z0; /* Use this if you are using the ABL log law wall function */
    }
  end_f_loop(f,t)
}

/* Modified wall roughness */
DEFINE_PROFILE(wallRoughnessModified,t,i)
{
  real x[ND_ND];
  face_t f;
  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      F_PROFILE(f,t,i) = 9.793*z0/Cs;
    }
  end_f_loop(f,t)
}

/* ************************* Cell Wall Distance ************************* */
/* To Use:  Define a UDS with Flux Function = none, no Inlet Diffusion
            Add Material Property "UDS Diffusivity"; defined-per-uds: constant, ...
                Coefficient = 1 [kg/ms]
```

```
            Add Source Terms for User Scalars in the cell zone: Source Term = 1
            Set Boundary Conditions for User Scalar: Specified Value = 0 on all ...
                boundaries to which the distance should be computed (boundary lower in ...
                the attached sample case); Specified Flux = 0 on all other boundaries.
            Define a User-Defined Memory Location in which the UDF stores the computed ...
                distance
            Hook to Fluent */

DEFINE_EXECUTE_AT_END(computeSelectedWallDistance)
{
    Domain *d=Get_Domain(1);
    Thread *t;
    cell_t c;
    real wallPhi, gradWallPhi, wallDistance;

    /* Check if UDM and UDS exist */
    if (N_UDM < 3 || N_UDS < 1) {
        Message0("\n  Error: No UDM or no UDS defined! Abort UDF execution.\n");
        return;
     }

    /* Loop over all threads and cells to compute the wall distance */
    thread_loop_c(t,d)
    {
        begin_c_loop(c,t)
        {
            /* Retrieve wallPhi from UDS-0 */
            wallPhi = C_UDSI(c,t,0);
            /* Compute magnitude of gradient of wallPhi */
            gradWallPhi = NV_MAG(C_UDSI_G(c,t,0));
            /* Compute local wall distance */
            wallDistance = -gradWallPhi + sqrt(MAX(gradWallPhi*gradWallPhi + 2*wallPhi, ...
                0));

            /* Store local wall distance in UDM-0 */
            C_UDMI(c,t,0) = wallDistance; /* Call C_UDMI(c,t,0) to retrieve the wall ...
                distance */
        }
        end_c_loop(c,t)
    }
}

/* ****************** Sources ********************
 **************** Corliolis Force *************** */
 DEFINE_SOURCE(Coriolis_X_source,c,t,dS,eqn)
 {
    real x[ND_ND];
    real source;
    real Lat, density;

    C_CENTROID(x,c,t);

    Lat = globalLat + (x[1] - offsetY)*9.0066*1e-6; /* Add the local lattitude change ...
        converted from m to degrees */
    density = C_R(c,t);

    source = 2.0*earthRot*sin(Lat * 3.1459/180)*density*C_V(c,t);
    dS[eqn] = 0.0;
    C_UDMI(c,t,1) = source;
    return source;
 }

 DEFINE_SOURCE(Coriolis_Y_source,c,t,dS,eqn)
 {
    real x[ND_ND];
    real source;
    real Lat, density;

    C_CENTROID(x,c,t);

    Lat = globalLat + (x[1] - offsetY)*9.0066*1e-6; /* Add the local lattitude change ...
        converted from m to degrees */
    density = C_R(c,t);

    source = -2.0*earthRot*sin(Lat * 3.1459/180)*density*C_U(c,t);
    dS[eqn] = 0.0;
    C_UDMI(c,t,2) = source;
    return source;
 }

 /* ************************* Initilization ************************** */

 DEFINE_INIT(initNeutral,d)
 {
    cell_t c;
    Thread *t;
    real x[ND_ND];
    real z;
    /* loop over all cell threads in the domain */
    thread_loop_c(t,d)
      {
      /* loop over all cells */
      begin_c_loop_all(c,t)
      {
      C_CENTROID(x,c,t);
       z = x[2] + z0;
      if (z > ablHeight){
        z = ablHeight;
      }

      if (z > maxZInit){
            C_U(c,t) = 0.0; /*x velocity */
            C_V(c,t) = (uStar/vonKarman)*log(z/z0); /* y velocity */
            C_W(c,t) = 0.0; /* z velocity */
            C_K(c,t) = pow(uStar,2.0)/sqrt(Cmu); /* k */
            C_D(c,t) = pow(uStar,3.0)/(vonKarman*z); /* epsilon */
            C_P(c,t) = 0.0; /*Pressure*/
```

```
                          }
        else{
            C_U(c,t) = 0.0;
            C_V(c,t) = initVelocity;
            C_W(c,t) = 0.0;
            C_K(c,t) = pow(uStar,2.0)/sqrt(Cmu);
            /* C_K(c,t) = initK; */
            C_D(c,t) = initEpsilon;
            C_P(c,t) = 0.0;
            }
        }
      end_c_loop_all(c,t)
      }
    }


  /* ************************* Wall Functions ************************* */

 /* Designed around u/uStar = 1/K*log(z/z0) ref: Improved k-e model and wall function ...
     formulation for the RANS simulation of ABL flows, Parente et al
    Removes the need for multiplying z0 by 9.73/Cs and can thus use roughness lengths ...
        directly from ABL modelling with first cell height = 2*z0*/


  DEFINE_WALL_FUNCTIONS(ABL_logLaw, f, t, c0, t0, wf_ret, yPlus, Emod)
  {
    real ustar_ground, E_prime, yPlus_prime, zp, dx_mag, wf_value;
    real mu=C_MU_L(c0,t0);
    real xf[ND_ND];
    real xc[ND_ND];
    real dx[ND_ND];

    F_CENTROID(xf, f, t);
    C_CENTROID(xc, c0,t0);

    dx[0] = xc[0] - xf[0];
    dx[1] = xc[1] - xf[1];
    dx[2] = xc[2] - xf[2];
    dx_mag = NV_MAG(dx);
    zp = dx_mag;

    ustar_ground = pow(C_K(c0,t0),0.5)*pow(Cmu, 0.25);
    E_prime = (mu/densOper)/(z0*ustar_ground);
    yPlus_prime = (zp+z0)*ustar_ground/(mu/densOper);

    switch (wf_ret)
      {
      case UPLUS_LAM:
         wf_value = yPlus;
         break;
      case UPLUS_TRB:
         wf_value = log(E_prime*yPlus_prime)/vonKarman;
         /*wf_value = log(Emod*yPlus)/vonKarman; Standard Fluent*/
         break;
      case DUPLUS_LAM:
         wf_value = 1.0;
         break;
      case DUPLUS_TRB:
         wf_value = 1.0/(vonKarman*yPlus_prime);
         break;
      case D2UPLUS_TRB:
         wf_value = -1.0/(vonKarman*yPlus_prime*yPlus_prime);
         break;
      default:
         printf("Wall function return value unavailable\n");
      }
    return wf_value;
  }
```

D.2 Unstable.c

```
#include "udf.h"
#include "math.h"

/* ***************************************************************
   **                         Unstable                          **
   ***************************************************************

   Fluent UDFs for simulating unstable ABL flow

   Control via the defined parameters
   Ensure the solver is in expert mode
   Use compiled UDF method

   /* C_UDMI - 12 User memory slots, 1 User scalar slot
   0 Wall Distance
   1 Cor x
   2 Cor y
   3 k DTU
   4 k Dtu Norm
   5 epsilon Fluent
   6 epsilon AM
   7 epsilon AM Ce3
   8 epsilon AM Gb
   9 epsilon DTU
   10 epsilon DTU - Ce3
   11 DTU Gb

   C_UDSI
   0 wallPhi - See description in define cell wall distance

  ------------------------------------------
  Owner: Hendri Breedt <u10028422@tuks.co.za>
  Date: 09/11/2017
  Version: 00 - Public release */

/* Model Constants - DTU */
#define Cmu 0.03
#define vonKarman 0.4
#define Ce1 1.21
#define Ce2 1.92
#define sigma_k 1.0
#define sigma_e 1.3
#define sigma_theta 1.0
#define PrTurb 0.85

/* Model Constants AM */
#define CmuAM 0.033
#define vonKarmanAM 0.42
#define Ce1AM 1.176
/* The rest are the same as the DTU model */

/* Wind speed relations */
#define z0  0.03 /*m*/
#define Cs 0.5 /* Roughness Constant */
#define uStar 0.3739
#define Lin -254.5957 /* L at the inlet - L must be < 0 to use this UDF set!!! */
#define Lmast -224.2239 /* L at the mast position Interpolation is performed from the ...
    inlety to the mast for the L values so that at the inlet the value is Lin and at ...
    the mast the value is L mast */
#define T0 313.0
#define Tstar -0.108
#define ablHeight 800.0

/* Site */
#define globalLat -33.0 /* Latitude of the origin in degrees - This is a dummy value ...
    for confidentiality*/
#define siteElevation 0.0 /*  Altitude of site AMSL - If you specify the operating ...
    pressure from site data then DO NOT change this value. */
#define earthRot 0.000072921159 /* Earth rotational speed */
#define offset 477.0 /* Use to control the z value, this is deducted from the mesh z ...
    coordinate.  This is the height AGL of the inlet location of the mesh */
#define offsetY -3000.0 /* This is deducted from the local lattitude in the corliolis ...
    calculation */
#define mastLocation 8687.0

/* General */
#define pi 3.141592
#define g -9.80665
#define R 8.3144598 /* Universal Gas Constant - Dry Air */
#define M 0.0289644 /* Molar mass of Earth's air */
#define Lb -0.0065  /* Standard temperature lapse rate*/

/* Operating Conditions - Material Air */
#define presOper 101325 /* Operating Pressure Pa - Internal Solver Pressure. This is ...
    the pressure specified at 0m and for this you can use lowest mast pressure reading */
#define tempOper 288.16 /* Operating Temperature - Internal Solver Standard ...
    Tempearture. This is the temperature based from the lowest measurement height on ...
    the mast. But can be left as the standard value */
#define densOper 1.0827  /* Problem density */
#define Cp 1006.43
#define beta 0.0032
#define viscosity 1.7894e-05

/* Initilization */
/*  Due to HAGL variations and Fluent not being able to compute cell distance before ...
    initialiazing we have to manually set the initialiaze values. These are used for z ...
    values lower than maxZInit, afterwards it returns to the inlt profile values */
#define maxZInit 1000.0  /* Height before using init values from inlet profiles */
#define initVelocity 10.0 /* y velocity */
#define initK 2.0     /* k */
```

```
#define initEpsilon 2.0    /* epsilon */

double linearInterpolation(double y);


/* ********************* Profiles ******************************
   ********************** Inlet Velocity ********************** */
DEFINE_PROFILE(inletVelocityUnstable,t,i)
{
    real x[ND_ND];
    real z;
    real phiM;
    face_t f;

  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      z = x[2] + z0 - offset;
      if (z > ablHeight){
      z = ablHeight;
      }
      phiM = pow(1.0-16.0*(z/Lin),-0.25);
      F_PROFILE(f, t, i) = (uStar/vonKarman)*(log(8.0*(z/z0) * (pow(phiM,4.0))/( ...
          pow(phiM+1.0,2.0)*(pow(phiM,2.0)+1.0))) -pi/2.0 + 2.0*atan(1.0/phiM));
    }
  end_f_loop(f,t)
}


/* ********************* Inlet Temperature ********************* */
/* The site values for temperature are in potential temperature. This is converted back ...
    to standard temperature via the operating pressure of Fluent */
 DEFINE_PROFILE(inletTemperatureUnstable,t,i)
{
     real x[ND_ND];
     real z;
     real phiM,potenTemp,pressure,zAMSL;
     face_t f;

  begin_f_loop(f,t)
     {
       F_CENTROID(x,f,t);
      z = x[2] + z0 - offset;
      if (z > ablHeight){
      z = ablHeight;
      }
       zAMSL = z + siteElevation;
       phiM = pow(1.0-16.0*(z/Lin),-0.25);
       potenTemp = T0 + (Tstar/vonKarman)*(log(z/z0) -2.0*log(0.5*(1.0+pow(phiM,-2.0))));
       pressure = presOper*pow(tempOper/(tempOper+Lb*zAMSL),(-g*M)/(R*Lb));
       F_PROFILE(f,t,i) = potenTemp/(pow(presOper/pressure,0.286));
     }
   end_f_loop(f,t)
}


/* ********************* Inlet k ********************** */
DEFINE_PROFILE(inlet_k_Unstable,t,i)
{
     real x[ND_ND];
     real z;
     real phiE,phiM;
     face_t f;

  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      z = x[2] + z0 - offset;
      if (z > ablHeight){
      z = ablHeight;
      }
      phiE = 1.0-(z/Lin);
      phiM = pow(1.0-16.0*(z/Lin),-0.25);
      F_PROFILE(f,t,i)  = (pow(uStar,2.0)/sqrt(Cmu))*pow(phiE/phiM,0.5);
    }
  end_f_loop(f,t)
}

/* ********************* Inlet epsilon ********************** */
DEFINE_PROFILE(inlet_e_Unstable,t,i)
{
    real x[ND_ND];
    real z;
    real phiE;
    face_t f;

  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      z = x[2] + z0 - offset;
      if (z > ablHeight){
      z = ablHeight;
      }
      phiE = 1.0-z/Lin;
      F_PROFILE(f,t,i) = phiE*pow(uStar,3.0)/(vonKarman*z);
    }
  end_f_loop(f,t)
}


/* ********************** Walls **********************

/*  ********************** Wall Roughness ********************** */
/* Use this if you are using the ABL log law wall function */
DEFINE_PROFILE(wallRoughness,t,i)
{
  real x[ND_ND];
```

```
  face_t f;
  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      F_PROFILE(f,t,i) = z0; /* Use this if you are using the ABL log law wall function */
    }
  end_f_loop(f,t)
}

/* Modified wall roughness */
DEFINE_PROFILE(wallRoughnessModified,t,i)
{
  real x[ND_ND];
  face_t f;
  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      F_PROFILE(f,t,i) = 9.793*z0/Cs;
    }
  end_f_loop(f,t)
}

/* ********************* Wall Temperature ********************** */
DEFINE_PROFILE(wallTemperatureUnstable,t,i)
{
  real x[ND_ND];
  face_t f;
  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      F_PROFILE(f,t,i) = T0;
    }
  end_f_loop(f,t)
}

/* ************************* Cell Wall Distance ************************** */
/* To Use:  Define a UDS with Flux Function = none, no Inlet Diffusion
      Add Material Property "UDS Diffusivity"; defined-per-uds: constant, Coefficient = ...
          1 [kg/ms]
      Add Source Terms for User Scalars in the cell zone: Source Term = 1
      Set Boundary Conditions for User Scalar: Specified Value = 0 on all boundaries to ...
          which the distance should be computed (boundary lower in the attached sample ...
          case); Specified Flux = 0 on all other boundaries.
      Define a User-Defined Memory Location in which the UDF stores the computed distance
      Hook to define_excecute_at_end */

DEFINE_EXECUTE_AT_END(computeSelectedWallDistance)
{
  Domain *d=Get_Domain(1);
  Thread *t;
  cell_t c;
  real wallPhi, gradWallPhi, wallDistance;

  /* Check if UDM and UDS exist */
  if (N_UDM < 12 || N_UDS < 1) {
    Message0("\n  Error: No UDM or no UDS defined! Abort UDF execution.\n");
    return;
   }

  /* Loop over all threads and cells to compute the wall distance */
  thread_loop_c(t,d)
  {
    begin_c_loop(c,t)
    {
      /* Retrieve wallPhi from UDS-0 */
      wallPhi = C_UDSI(c,t,0);
      /* Compute magnitude of gradient of wallPhi */
      gradWallPhi = NV_MAG(C_UDSI_G(c,t,0));
      /* Compute local wall distance */
      wallDistance = -gradWallPhi + sqrt(MAX(gradWallPhi*gradWallPhi + 2*wallPhi, 0));

      /* Store local wall distance in UDM-0 */
      C_UDMI(c,t,0) = wallDistance; /* Call C_UDMI(c,t,0) to retrieve the wall distance */
    }
    end_c_loop(c,t)
  }
}

/* ******************** Sources ********************
  **************** Corliolis Force *************** */
 DEFINE_SOURCE(Coriolis_X_source,c,t,dS,eqn)
 {
    real x[ND_ND];
    real source;
    real Lat, density;

    C_CENTROID(x,c,t);

    Lat = globalLat + (x[1] - offsetY)*9.0066*1e-6; /* Add the local lattitude change ...
        converted from m to degrees */
    density = C_R(c,t);

    source = 2.0*earthRot*sin(Lat * 3.1459/180)*density*C_V(c,t);
    dS[eqn] = 0.0;
    C_UDMI(c,t,1) = source;
    return source;
 }

 DEFINE_SOURCE(Coriolis_Y_source,c,t,dS,eqn)
 {
    real x[ND_ND];
    real source;
    real Lat, density;

    C_CENTROID(x,c,t);
```

```
    Lat = globalLat + (x[1] - offsetY)*9.0066*1e-6; /* Add the local lattitude change ...
        converted from m to degrees */
    density = C_R(c,t);

    source = -2.0*earthRot*sin(Lat * 3.1459/180)*density*C_U(c,t);
    dS[eqn] = 0.0;
    C_UDMI(c,t,2) = source;
    return source;
 }

/* ************************ k ****************************
 DTU
 No energy eqaution is solved with this model*/
 DEFINE_SOURCE(k_source_DTU_Unstable,c,t,dS,eqn)
 {
    real fUn, phiM, phiE, phiH, CkD, source, Gb, Sk, uStarLocal;
    real x[ND_ND];
    real z, L;
    C_CENTROID(x,c,t);
    z = C_UDMI(c,t,0) + z0;
    L = linearInterpolation(x[1]);
    if (z > ablHeight){
    z = ablHeight;
    }


    if (N_ITER > 5) {
    phiM = pow(1.0-16.0*(z/L),-0.25);
    phiE = 1.0-(z/L);
    phiH = sigma_theta*pow(1.0-16.0*(z/L),-0.5);
    uStarLocal = pow(C_K(c,t),0.5)*pow(Cmu,0.25)*pow(phiM,0.25)*pow(phiE,-0.25);

    fUn = 2.0-(z/L) + 8.0*(1.0-12.0*(z/L)+7.0*pow(z/L,2.0)) - ...
        16.0*(z/L)*(3.0-54.0*(z/L)+35.0*pow(z/L,2.0));
    CkD = pow(vonKarman,2)/(sigma_k*sqrt(Cmu));
    Gb = -C_MU_T(c,t)*pow(sqrt(C_U_G(c,t)[2]*C_U_G(c,t)[2] + ...
        C_V_G(c,t)[2]*C_V_G(c,t)[2]),2.0)*((z/L)/(sigma_theta))*(phiH/pow(phiM,2)); /* ...
        DTU Formulation */
    Sk = pow(uStarLocal,3.0)/(vonKarman*L)*((L/z)*(phiM - phiE) - ...
        (phiH)/(sigma_theta*phiM) - 0.25*CkD*pow(phiM,6.5)*pow(phiE,-1.5)*fUn);

    source = -densOper*Sk + Gb;
    }
    else {
    source = 0.0; /* Only run this source after 5 iterations. The gradients can cuase ...
        divergence with an illposed initilization */
    Sk = 0.0;
    }

    dS[eqn] = 0.0;
    C_UDMI(c,t,3) = Sk;
    C_UDMI(c,t,4) = Sk*vonKarman*z/pow(uStar,3.0);
    return source;
 }


/* ************************ Epsilon *********************
 Epsilon is a function of the gradients and to save these the solver needs to be in ...
     expert mode
 Issue: 'solve/set/expert' in the FLUENT window, and answer YES when it asks if you ...
     want to free temporary memory

 Standard Fluent buoyancy treatment for epsilon
 Checking advanced buoyancy treatmnent in the viscous model box adds in the formulation ...
     below
 Changes in the model is made by changing Ce3 according to the AM or DTU method
 Not checking the box sets Gb = 0, this term is then re added in by the sources below. ...
     Do not check the box in the viscous box! */
 DEFINE_SOURCE(epsilon_source_Fluent_Unstable,c,t,dS,eqn)
 {
    real Gb, C3e, source;

    if (N_ITER > 5) {
    Gb = beta*g*C_MU_T(c,t)/PrTurb*C_T_G(c,t)[2]; /* Standard Fluent Gb formulation, ...
        C_MU_T = Turbulent Viscosity, PrTurb = Turbulent Prandtl number, C_T_G = ...
        [partial_T/partial_xi] */
    C3e = tanh(fabs(C_V(c,t)/C_U(c,t)));     /* Standard Fluent C3e formulation, C_V = ...
        v velocity, C_U = x velocity */
    source = Ce1*C_D(c,t)/C_K(c,t)*C3e*Gb;  /* C_D = epsilon, C_K = k */
    }
    else {
      source = 0.0; /* Only run this source after 5 iterations. The gradients can cuase ...
          divergence with an illposed initilization */
    }
    C_UDMI(c,t,5) = source;
    dS[eqn] = 0;
    return source;
 }


/* ALot & Masson */
/* Epsilon source treatment based on an anylytical expression for Ce3 */
/* % Only valid of -2.3 < z/L < 2 and also highly sensitive*/
 DEFINE_SOURCE(epsilon_source_AM_Unstable,c,t,dS,eqn)
 {
  real x[ND_ND];
  real z, L;
  real Gb, C3e, source;
  real a0, a1, a2, a3, a4, a5;
  C_CENTROID(x,c,t);
  z = C_UDMI(c,t,0) + z0;
  L = linearInterpolation(x[1]);

  if (z > ablHeight){
  z = ablHeight;
  }
```

```
  if (N_ITER > 5 && z/L > -2.3){
    if (z/L > -0.25) {
    a0 = -0.0609;
    a1 = -33.672;
    a2 = -546.88;
    a3 = -3234.06;
    a4 = -9490.792;
    a5 = -11163.202;
    }
    else {
    a0 = 1.1765;
    a1 = 17.1346;
    a2 = 19.165;
    a3 = 11.912;
    a4 = 3.821;
    a5 = 0.492;
    }

    Gb = beta*g*C_MU_T(c,t)/PrTurb*C_T_G(c,t)[2];
    C3e = a0*pow((z/L),0) + a1*pow((z/L),1.0) + a2*pow((z/L),2.0) + a3*pow((z/L),3.0) + ...
        a4*pow((z/L),4.0) + a5*pow((z/L),5.0); /* AM C3e formulation */
    }
    else if (N_ITER > 5 && z/L ≤ -2.3){
    Gb = beta*g*C_MU_T(c,t)/PrTurb*C_T_G(c,t)[2];
    C3e = -6.523095460000015;
    }
    else{
    Gb = 0.0;
    C3e = 0.0;
    }

  dS[eqn] = 0;
  source = Ce1AM*C_D(c,t)/C_K(c,t)*C3e*Gb;
  C_UDMI(c,t,6) = source;
  C_UDMI(c,t,7) = C3e;
  C_UDMI(c,t,8) = Gb*vonKarmanAM*z/pow(uStar,3.0);
  return source;
  }

/* 2 - This uses the DTU Gb formulation and is run without a temperature eqaution */
DEFINE_SOURCE(epsilon_source_AM_Unstable_2,c,t,dS,eqn)
 {
  real x[ND_ND];
  real z, L;
  real Gb, C3e, phiM, phiH, source;
  real a0, a1, a2, a3, a4, a5;
  C_CENTROID(x,c,t);
  z = C_UDMI(c,t,0) + z0;
  L = linearInterpolation(x[1]);

  if (z > ablHeight){
  z = ablHeight;
  }
  phiM = pow(1.0-16.0*(z/L),-0.25);
  phiH = sigma_theta*pow(1.0-16.0*(z/L),-0.5);

  if (N_ITER > 5 && z/L > -2.3){
    if (z/L > -0.25) {
    a0 = -0.0609;
    a1 = -33.672;
    a2 = -546.88;
    a3 = -3234.06;
    a4 = -9490.792;
    a5 = -11163.202;
    }
    else {
    a0 = 1.1765;
    a1 = 17.1346;
    a2 = 19.165;
    a3 = 11.912;
    a4 = 3.821;
    a5 = 0.492;
    }

    Gb = C_MU_T(c,t)*pow(sqrt(C_U_G(c,t)[2]*C_U_G(c,t)[2] + ...
        C_V_G(c,t)[2]*C_V_G(c,t)[2]),2.0)*((z/L)/(sigma_theta))*(phiH/pow(phiM,2.0));
    C3e = a0*pow((z/L),0) + a1*pow((z/L),1.0) + a2*pow((z/L),2.0) + a3*pow((z/L),3.0) + ...
        a4*pow((z/L),4.0) + a5*pow((z/L),5.0); /* AM C3e formulation */
    }
    else if (N_ITER > 5 && z/L ≤ -2.3){
    Gb = C_MU_T(c,t)*pow(sqrt(C_U_G(c,t)[2]*C_U_G(c,t)[2] + ...
        C_V_G(c,t)[2]*C_V_G(c,t)[2]),2.0)*((z/L)/(sigma_theta))*(phiH/pow(phiM,2.0));
    C3e = -6.523095460000015;
    }
    else{
    Gb = 0.0;
    C3e = 0.0;
    }

  dS[eqn] = 0;
  source = Ce1AM*C_D(c,t)/C_K(c,t)*C3e*Gb;
  C_UDMI(c,t,6) = source;
  C_UDMI(c,t,7) = C3e;
  C_UDMI(c,t,8) = Gb*vonKarmanAM*z/pow(uStar,3.0);
  return source;
  }


/* DTU */
/* Epsilon source treatment based on an anylytical expression for Ce3 */
/* No energy eqaution is solved with this model */
 DEFINE_SOURCE(epsilon_source_DTU_Unstable,c,t,dS,eqn)
 {
    real x[ND_ND];
    real z, L;
    real Gb, C3e, source;
```

```
    real phiM, phiH, phiE, fe;
    C_CENTROID(x,c,t);
    z = C_UDMI(c,t,0) + z0;
    L = linearInterpolation(x[1]);

    if (z > ablHeight){
    z = ablHeight;
    }


    if (N_ITER > 5) {
    phiM = pow(1.0-16.0*(z/L),-0.25);
    phiE = 1.0-(z/L);
    phiH = sigma_theta*pow(1.0-16.0*(z/L),-0.5);
    fe = pow(phiM,2.5)*(1.0-0.75*16.0*(z/L));
    /* Gb = beta*g*C_MU_T(c,t)/PrTurb*C_T_G(c,t)[2]; */ /*Standard Fluent Gb ...
        formulation, C_MU_T = Turbulent Viscosity, PrTurb = Turbulent Prandtl number, ...
        C_T_G = [partial_T/partial_xi] */
    Gb = -C_MU_T(c,t)*pow(sqrt(C_U_G(c,t)[2]*C_U_G(c,t)[2] + ...
        C_V_G(c,t)[2]*C_V_G(c,t)[2]),2.0)*((z/L)/(sigma_theta))*(phiH/pow(phiM,2.0)); ...
        /* DTU Formulation */
    C3e = ...
        (sigma_theta/(z/L))*(phiM/phiH)*(Ce1*phiM-Ce2*phiE+(Ce2-Ce1)*pow(phiE,-0.5)*fe); ...
            /* DTU C3e formulation */

    source = Ce1*C_D(c,t)/C_K(c,t)*C3e*Gb;  /*C_D = epsilon, C_K = k */
    }
    else {
    source = 0.0; /* Only run this source after 5 iterations. The gradients can cuase ...
        divergence with an illposed initilization */
    Gb = 0.0;
    C3e = 0.0;

    }
    dS[eqn] = 0.0;

    C_UDMI(c,t,9) = source;
    C_UDMI(c,t,10) = C3e;
    C_UDMI(c,t,11) = Gb*vonKarman*z/pow(uStar,3.0);
    return source;
 }

/* ************************* Initilization ************************* */

 DEFINE_INIT(initUnstable,d)
 {
    cell_t c;
    Thread *t;
    real x[ND_ND];
    real phiM, phiE, phiH, pressure, potenTemp, z, zAMSL, L;
    /* loop over all cell threads in the domain */
    thread_loop_c(t,d)
      {
         /* loop over all cells */
      begin_c_loop_all(c,t)
          {
      C_CENTROID(x,c,t);
      z = x[2] + z0 - offset;
      L = linearInterpolation(x[1]);
      if (z > ablHeight){
      z = ablHeight;
      }

      if (z > maxZInit){
      phiM = pow(1.0-16.0*(z/L),-0.25);
      phiE = 1.0-(z/L);
      phiH = sigma_theta*pow(1.0-16.0*(z/L),-0.5);
      C_U(c,t) = 0.0; /*x velocity */
      C_V(c,t) = (uStar/vonKarman)*(log(8.0*(z/z0) * (pow(phiM,4.0))/( ...
          pow(phiM+1.0,2.0)*(pow(phiM,2.0)+1.0))) -pi/2.0 + 2.0*atan(1.0/phiM)); /* y ...
          velocity */
      C_W(c,t) = 0.0; /* z velocity */
      /* C_T(c,t) = potenTemp/(pow(presOper/pressure,0.286)); /* Temperature */
      C_K(c,t) = (pow(uStar,2.0)/sqrt(Cmu))*pow(phiE/phiM,0.5); /* k */
      C_D(c,t) = phiE*pow(uStar,3.0)/(vonKarman*z); /* epsilon */
      C_P(c,t) = 0.0; /*Pressure*/
               }
      else{
      C_U(c,t) = 0.0;
      C_V(c,t) = initVelocity;
      C_W(c,t) = 0.0;
      C_K(c,t) = initK;
      C_D(c,t) = initEpsilon;
      C_P(c,t) = 0.0;
          }
        }
         end_c_loop_all(c,t)
  }
}


 /* ************************* Wall Functions ************************* */

/* Designed around u/uStar = 1/K*log(z/z0) ref: Improved k-e model and wall function ...
    formulation for the RANS simulation of ABL flows, Parente et al
    Removes the need for multiplying z0 by 9.73/Cs and can thus use roughness lengths ...
        directly from ABL modelling with first cell height = 2*z0*/

   DEFINE_WALL_FUNCTIONS(ABL_logLaw, f, t, c0, t0, wf_ret, yPlus, Emod)
 {
    real ustar_ground, E_prime, yPlus_prime, zp, dx_mag, wf_value;
    real mu=C_MU_L(c0,t0);
    real xf[ND_ND];
    real xc[ND_ND];
    real dx[ND_ND];

    F_CENTROID(xf, f, t);
```

```
    C_CENTROID(xc, c0,t0);

    dx[0] = xc[0] - xf[0];
    dx[1] = xc[1] - xf[1];
    dx[2] = xc[2] - xf[2];
    dx_mag = NV_MAG(dx);
    zp = dx_mag;

    ustar_ground = pow(C_K(c0,t0),0.5)*pow(Cmu, 0.25);
    E_prime = (mu/densOper)/(z0*ustar_ground);
    yPlus_prime = (zp+z0)*ustar_ground/(mu/densOper);

    switch (wf_ret)
      {
      case UPLUS_LAM:
         wf_value = yPlus;
         break;
      case UPLUS_TRB:
         wf_value = log(E_prime*yPlus_prime)/vonKarman;
         /*wf_value = log(Emod*yPlus)/vonKarman; Standard Fluent*/
         break;
      case DUPLUS_LAM:
         wf_value = 1.0;
         break;
      case DUPLUS_TRB:
         wf_value = 1.0/(vonKarman*yPlus_prime);
         break;
      case D2UPLUS_TRB:
         wf_value = -1.0/(vonKarman*yPlus_prime*yPlus_prime);
         break;
      default:
         printf("Wall function return value unavailable\n");
      }
    return wf_value;
 }

 /* ************************* Interpolation ************************* */
 /* Currently does linear interpolation, Must be run with 180degree inlet location. ...
     This function can be expanded in future to bilinear (or more) to include more ...
     mast/WRF locations */
double linearInterpolation(double y)
{
   double L;
      if (y > mastLocation){
      L = Lmast;
      }
      else{
      L = (Lin*(mastLocation - y) + Lmast*(y - offsetY))/(mastLocation - offsetY); /* ...
          Local L */
      }

   return L;
}
```

D.3 Stable.c

```c
#include "udf.h"
#include "math.h"

/* ***************************************************************
   **                           Stable                          **
   ***************************************************************
   Fluent UDFs for simulating stable ABL flow

   Control via the defined parameters
   Ensure the solver is in expert mode
   Use compiled UDF method

   C_UDMI - 12 User memory slots, 1 User scalar slot
   0 Wall Distance
   1 Cor x
   2 Cor y
   3 k DTU
   4 k Dtu Norm
   5 epsilon Fluent
   6 epsilon AM
   7 epsilon AM Ce3
   8 epsilon AM Gb
   9 epsilon DTU
   10 epsilon DTU - Ce3
   11 DTU Gb

   C_UDSI
   0 wallPhi - See description in define cell wall distance

  -------------------------------------------
  Owner: Hendri Breedt <u10028422@tuks.co.za>
  Date: 09/11/2017
  Version: 00 - Public release */

/* Model Constants - DTU */
#define Cmu 0.03
#define vonKarman 0.4
#define Ce1 1.21
#define Ce2 1.92
#define sigma_k 1.0
#define sigma_e 1.3
#define sigma_theta 1.0
#define PrTurb 0.85

/* Model Constants AM */
#define CmuAM 0.033
#define vonKarmanAM 0.42
#define Ce1AM 1.176
/* The rest are the same as the DTU model */

/* Wind speed relations */
#define z0 0.03 /*m*/
#define Cs 0.5 /* Roughness Constant */
#define uStar 0.1407
#define Lin 124.7334 /* L - Inlet L must be > 0 to use this UDF set!!! */
#define Lmast 222.1774 /* L at the mast position Interpolation is performed from the ...
    inlety to the mast for the L values so that at the inlet the value is Lin and at ...
    the mast the value is L mast */
#define T0 288.0
#define Tstar 0.0232
#define ablHeight 600.0 /* Height of ABL, this is the height for fixed values of all ...
    profiles */

/* Site */
#define globalLat -33.0 /* Latitude of the origin in degrees - This is a dummy value ...
    for confidentiality*/
#define siteElevation 0.0 /* Altitude of site AMSL - If you specify the operating ...
    pressure from site data then DO NOT change this value. */
#define earthRot 0.000072921159 /* Earth rotational speed */
#define offset 477.0 /* Use to control the z value, this is deducted from the mesh z ...
    coordinate.  This is the height AGL of the inlet location of the mesh */
#define offsetY -3000.0 /* This is deducted from the local lattitude in the corliolis ...
    calculation */
#define mastLocation 8687.0


/* General */
#define pi 3.141592
#define g -9.80665
#define R 8.3144598 /* Universal Gas Constant - Dry Air */
#define M 0.0289644 /* Molar mass of Earth's air */
#define Lb -0.0065 /* Standard temperature lapse rate */

/* Operating Conditions - Material Air */
#define presOper 101325 /* Operating Pressure Pa - Internal Solver Pressure. This is ...
    the pressure specified at 0m and for this you can use lowest mast pressure reading */
#define tempOper 288.16 /* Operating Temperature - Internal Solver Standard ...
    Tempearture. This is the temperature based from the lowest measurement height on ...
    the mast. But can be left as the standard value */
#define densOper 1.0800 /* Problem density */
#define Cp 1006.43
#define beta 0.0032
#define viscosity 1.7894e-05

/* Initilization */
/*  Due to HAGL variations and Fluent not being able to compute cell distance before ...
    initialiazing we have to manually set the initialiaze values. These are used for z ...
    values lower than maxZInit, afterwards it returns to the inlt profile values */
#define maxZInit 1000.0   /* Height before using init values from inlet profiles */
#define initVelocity 10.0 /* y velocity */
```

```
#define initK 2.0          /* k */
#define initEpsilon 2.0        /* epsilon */

double linearInterpolation(double y);
/* ********************* Profiles ****************************** */

/* ********************** Inlet Velocity ********************* */
DEFINE_PROFILE(inletVelocityStable,t,i)
{
    real x[ND_ND];
    real z;
    real phiM;
    face_t f;

  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      z = x[2] + z0 - offset;
      if (z > ablHeight){
      z = ablHeight;
      }
      phiM = 1.0 + 5.0*(z/Lin);
      F_PROFILE(f, t, i) = (uStar/vonKarman)*(log(z/z0) +phiM -1.0);
    }
  end_f_loop(f,t)
}


/* ********************** Inlet Temperature *********************
The site values for temperature are in potential temperature. This is converted back to ...
    standard temperature via the operating pressure of Fluent. */
 DEFINE_PROFILE(inletTemperatureStable,t,i)
{
     real x[ND_ND];
     real z;
     real phiM, potenTemp, pressure, zAMSL;
     face_t f;

  begin_f_loop(f,t)
     {
       F_CENTROID(x,f,t);
      z = x[2] + z0 - offset;
      if (z > ablHeight){
      z = ablHeight;
      }
       zAMSL = z + siteElevation;
       phiM = 1.0 + 5.0*(z/Lin);
       potenTemp = T0 + (Tstar/vonKarman)*(log(z/z0) +phiM -1.0);
       pressure = presOper*pow(tempOper/(tempOper+Lb*zAMSL),(-g*M)/(R*Lb));
       F_PROFILE(f,t,i) = potenTemp/(pow(presOper/pressure,0.286));
     }
   end_f_loop(f,t)
}


/* ********************** Inlet k ********************* */
DEFINE_PROFILE(inlet_k_Stable,t,i)
{
     real x[ND_ND];
     real z;
     real phiE,phiM;
     face_t f;

  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      z = x[2] + z0 - offset;
      if (z > ablHeight){
      z = ablHeight;
      }
      phiM = 1.0 + 5.0*(z/Lin);
      phiE = phiM-z/Lin;
      F_PROFILE(f,t,i)  = (pow(uStar,2.0)/sqrt(Cmu))*pow(phiE/phiM,0.5);
    }
  end_f_loop(f,t)
}

/* ********************** Inlet epsilon ********************* */
DEFINE_PROFILE(inlet_e_Stable,t,i)
{
    real x[ND_ND];
    real z;
    real phiE, phiM;
    face_t f;

  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      z = x[2] + z0 - offset;
      if (z > ablHeight){
      z = ablHeight;
      }
      phiM = 1.0 + 5.0*(z/Lin);
      phiE = phiM-z/Lin;
      F_PROFILE(f,t,i) = phiE*pow(uStar,3.0)/(vonKarman*z);
    }
  end_f_loop(f,t)
}


/* ********************** Walls ********************* */

/*  ********************** Wall Roughness ********************* */
/* Use this if you are using the ABL log law wall function */
DEFINE_PROFILE(wallRoughness,t,i)
{
  real x[ND_ND];
```

```
  face_t f;
  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      F_PROFILE(f,t,i) = z0; /* Use this if you are using the ABL log law wall function */
    }
  end_f_loop(f,t)
}

/* Modified wall roughness */
DEFINE_PROFILE(wallRoughnessModified,t,i)
{
  real x[ND_ND];
  face_t f;
  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      F_PROFILE(f,t,i) = 9.793*z0/Cs;
    }
  end_f_loop(f,t)
}

/* ********************** Wall Temperature ********************* */
DEFINE_PROFILE(wallTemperatureStable,t,i)
{
  real x[ND_ND];
  face_t f;
  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      F_PROFILE(f,t,i) = T0;
    }
  end_f_loop(f,t)
}

/* ************************ Cell Wall Distance ************************ */
/* To Use:  Define a UDS with Flux Function = none, no Inlet Diffusion
      Add Material Property "UDS Diffusivity"; defined-per-uds: constant, Coefficient = ...
          1 [kg/ms]
      Add Source Terms for User Scalars in the cell zone: Source Term = 1
      Set Boundary Conditions for User Scalar: Specified Value = 0 on all boundaries to ...
          which the distance should be computed (boundary lower in the attached sample ...
          case); Specified Flux = 0 on all other boundaries.
      Define a User-Defined Memory Location in which the UDF stores the computed distance
      Hook to define_excecute_at_end */

DEFINE_EXECUTE_AT_END(computeSelectedWallDistance)
{
  Domain *d=Get_Domain(1);
  Thread *t;
  cell_t c;
  real wallPhi, gradWallPhi, wallDistance;

  /* Check if UDM and UDS exist */
  if (N_UDM < 12 || N_UDS < 1) {
    Message0("\n  Error: No UDM or no UDS defined! Abort UDF execution.\n");
    return;
   }

  /* Loop over all threads and cells to compute the wall distance */
  thread_loop_c(t,d)
  {
    begin_c_loop(c,t)
    {
      /* Retrieve wallPhi from UDS-0 */
      wallPhi = C_UDSI(c,t,0);
      /* Compute magnitude of gradient of wallPhi */
      gradWallPhi = NV_MAG(C_UDSI_G(c,t,0));
      /* Compute local wall distance */
      wallDistance = -gradWallPhi + sqrt(MAX(gradWallPhi*gradWallPhi + 2*wallPhi, 0));

      /* Store local wall distance in UDM-0 */
      C_UDMI(c,t,0) = wallDistance; /* Call C_UDMI(c,t,0) to retrieve the wall distance */
    }
    end_c_loop(c,t)
  }
}

/* ******************* Sources ******************* */

/*  *************** Corliolis Force ************** */
 DEFINE_SOURCE(Coriolis_X_source,c,t,dS,eqn)
 {
    real x[ND_ND];
    real source;
    real Lat, density;

    C_CENTROID(x,c,t);

    Lat = globalLat + (x[1] - offsetY)*9.0066*1e-6; /* Add the local lattitude change ...
        converted from m to degrees */
    density = C_R(c,t);

    source = 2.0*earthRot*sin(Lat * 3.1459/180)*density*C_V(c,t);
    dS[eqn] = 0.0;
    C_UDMI(c,t,1) = source;
    return source;
 }

 DEFINE_SOURCE(Coriolis_Y_source,c,t,dS,eqn)
 {
    real x[ND_ND];
    real source;
    real Lat, density;

    C_CENTROID(x,c,t);

```

```
    Lat = globalLat + (x[1] - offsetY)*9.0066*1e-6; /* Add the local lattitude change ...
        converted from m to degrees */
    density = C_R(c,t);

    source = -2.0*earthRot*sin(Lat * 3.1459/180)*density*C_U(c,t);
    dS[eqn] = 0.0;
    C_UDMI(c,t,2) = source;
    return source;
 }

/* *********************** k *************************** */
/* No energy eqaution is solved with this model*/
 DEFINE_SOURCE(k_source_DTU_Stable,c,t,dS,eqn)
 {
    real fSt, phiM, phiE, phiH, CkD, source, Gb, Sk, uStarLocal;
    real x[ND_ND];
    real z, L;
    C_CENTROID(x,c,t);
    z = C_UDMI(c,t,0) + z0;
    L = linearInterpolation(x[1]);
    if (z > ablHeight){
    z = ablHeight;
    }

    if (N_ITER > 5) {
    phiM = 1.0 + 5.0*(z/L);
    phiE = phiM-z/L;
    phiH = 1.0 + 5.0*(z/L);
    uStarLocal = pow(C_K(c,t),0.5)*pow(Cmu,0.25)*pow(phiM,0.25)*pow(phiE,-0.25);

    fSt = 2.0-(z/L) - 10.0*(z/L)*(1.0-2.0*(z/L) + 10.0*(z/L));
    CkD = pow(vonKarman,2.0)/(sigma_k*sqrt(Cmu));
    Gb = -C_MU_T(c,t)*pow(sqrt(C_U_G(c,t)[2]*C_U_G(c,t)[2] + ...
        C_V_G(c,t)[2]*C_V_G(c,t)[2]),2.0)*((z/L)/(sigma_theta))*(phiH/pow(phiM,2.0)); ...
        /* DTU Formulation */
    Sk = pow(uStarLocal,3.0)/(vonKarman*L)*(1.0 - (phiH)/(sigma_theta*phiM) - ...
        0.25*CkD*pow(phiM,-3.5)*pow(phiE,-1.5)*fSt);
    source = -densOper*Sk + Gb;
    }
    else {
     source = 0.0;
    }

    dS[eqn] = 0.0;
    C_UDMI(c,t,3) = Sk;
    C_UDMI(c,t,4) = Sk*vonKarman*z/pow(uStar,3.0);
    return source;
 }


/* *********************** Epsilon *********************
 Epsilon is a function of the gradients and to save these the solver needs to be in ...
     expert mode
 Issue: 'solve/set/expert' in the FLUENT window, and answer YES when it asks if you ...
     want to free temporary memory

 Standard Fluent buoyancy treatment for epsilon
 Checking advanced buoyancy treatment in the viscous model box adds in the formulation ...
     below
 Changes in the model is made by changing Ce3 according to the AM or DTU method
 Not checking the box sets Gb = 0, this term is then re added in by the sources below. ...
     Do not check the box in the viscous box! */
 DEFINE_SOURCE(epsilon_source_Fluent_Stable,c,t,dS,eqn)
 {
    real Gb, C3e, source;

    if (N_ITER > 5) {
    Gb = beta*g*C_MU_T(c,t)/PrTurb*C_T_G(c,t)[2]; /* Standard Fluent Gb formulation, ...
        C_MU_T = Turbulent Viscosity, PrTurb = Turbulent Prandtl number, C_T_G = ...
        [partial_T/partial_xi] */
    C3e = tanh(fabs(C_V(c,t)/C_U(c,t)));      /* Standard Fluent C3e formulation, C_V = ...
        v velocity, C_U = x velocity */
    source = Ce1*C_D(c,t)/C_K(c,t)*C3e*Gb;  /* C_D = epsilon, C_K = k */
    }
    else {
    source = 0.0; /* Only run this source after 15 iterations. The gradients can cuase ...
        divergence with an illposed initilization */
    }
     dS[eqn] = 0.0;
    C_UDMI(c,t,5) = source;
    return source;
 }


/* ALot & Masson
/* Epsilon source treatment based on an anylytical expression for Ce3 */
/* Only valid of -2.3 < z/L < 2 and also highly sensitive*/
 DEFINE_SOURCE(epsilon_source_AM_Stable,c,t,dS,eqn)
 {
  real x[ND_ND];
  real z, L;
  real Gb, C3e, source;
  real a0, a1, a2, a3, a4, a5;
  C_CENTROID(x,c,t);
  z = C_UDMI(c,t,0) + z0;
  L = linearInterpolation(x[1]);
  if (z > ablHeight){
  z = ablHeight;
  }

  if (N_ITER > 5 && z/L < 2.0) {
    if (z/L < 0.33) {
    a0 = 4.181;
    a1 = 33.994;
    a2 = -442.398;
    a3 = 2368.12;
    a4 = -6043.544;
```

```
      a5 = 5970.776;
      }
      else {
      a0 = 5.225;
      a1 = -5.269;
      a2 = 5.115;
      a3 = -2.406;
      a4 = 0.435;
      a5 = 0;
      }

      Gb = beta*g*C_MU_T(c,t)/PrTurb*C_T_G(c,t)[2];
      C3e = a0*pow((z/L),0) + a1*pow((z/L),1.0) + a2*pow((z/L),2.0) + a3*pow((z/L),3.0) + ...
          a4*pow((z/L),4.0) + a5*pow((z/L),5.0); /* AM C3e formulation */
      }
      else if (N_ITER > 5 && z/L ≥ 2.0){
      Gb = beta*g*C_MU_T(c,t)/PrTurb*C_T_G(c,t)[2];
      C3e = 2.858999999999999;
      }
      else{
      Gb = 0.0;
      C3e = 0.0;
      }

    dS[eqn] = 0;
    source = Ce1AM*C_D(c,t)/C_K(c,t)*C3e*Gb;
    C_UDMI(c,t,6) = source;
    C_UDMI(c,t,7) = C3e;
    C_UDMI(c,t,8) = Gb*vonKarmanAM*z/pow(uStar,3.0);
    return source;
    }

  /* 2 - This uses the DTU Gb formulation and is run without a temperature eqaution */
  DEFINE_SOURCE(epsilon_source_AM_Stable_2,c,t,dS,eqn)
   {
    real x[ND_ND];
    real z, L;
    real Gb, C3e, phiM, phiH, source;
    real a0, a1, a2, a3, a4, a5;
    C_CENTROID(x,c,t);
    z = C_UDMI(c,t,0) + z0;
    L = linearInterpolation(x[1]);

    if (z > ablHeight){
    z = ablHeight;
    }
    phiM = 1.0 + 5.0*(z/L);
    phiH = 1.0 + 5.0*(z/L);

    if (N_ITER > 5 && z/L < 2.0) {
      if (z/L < 0.33) {
      a0 = 4.181;
      a1 = 33.994;
      a2 = -442.398;
      a3 = 2368.12;
      a4 = -6043.544;
      a5 = 5970.776;
      }
      else {
      a0 = 5.225;
      a1 = -5.269;
      a2 = 5.115;
      a3 = -2.406;
      a4 = 0.435;
      a5 = 0;
      }

      Gb = C_MU_T(c,t)*pow(sqrt(C_U_G(c,t)[2]*C_U_G(c,t)[2] + ...
          C_V_G(c,t)[2]*C_V_G(c,t)[2]),2.0)*((z/L)/(sigma_theta))*(phiH/pow(phiM,2.0));
      C3e = a0*pow((z/L),0) + a1*pow((z/L),1.0) + a2*pow((z/L),2.0) + a3*pow((z/L),3.0) + ...
          a4*pow((z/L),4.0) + a5*pow((z/L),5.0); /* AM C3e formulation */
      }
      else if (N_ITER > 5 && z/L ≥ 2.0){
      Gb = C_MU_T(c,t)*pow(sqrt(C_U_G(c,t)[2]*C_U_G(c,t)[2] + ...
          C_V_G(c,t)[2]*C_V_G(c,t)[2]),2.0)*((z/L)/(sigma_theta))*(phiH/pow(phiM,2.0));
      C3e = 2.858999999999999;
      }
      else{
      Gb = 0.0;
      C3e = 0.0;
      }

    dS[eqn] = 0;
    source = Ce1AM*C_D(c,t)/C_K(c,t)*C3e*Gb;
    C_UDMI(c,t,6) = source;
    C_UDMI(c,t,7) = C3e;
    C_UDMI(c,t,8) = Gb*vonKarmanAM*z/pow(uStar,3.0);
    return source;
    }

  /* DTU
    Epsilon source treatment based on an anylytical expression for Ce3 */
   DEFINE_SOURCE(epsilon_source_DTU_Stable,c,t,dS,eqn)
   {
      real x[ND_ND];
      real z, L;
      real Gb, C3e, source;
      real phiM, phiH, phiE, fe;
      C_CENTROID(x,c,t);
      z = C_UDMI(c,t,0) + z0;
      L = linearInterpolation(x[1]);

      if (z > ablHeight){
      z = ablHeight;
      }
```

```
    if (N_ITER > 5) {
    phiM = 1.0 + 5.0*(z/L);
    phiE = phiM-z/L;
    phiH = 1.0 + 5.0*(z/L);
    fe = pow(phiM,-2.5)*(2.0*phiM-1.0);

    Gb = -C_MU_T(c,t)*pow(sqrt(C_U_G(c,t)[2]*C_U_G(c,t)[2] + ...
        C_V_G(c,t)[2]*C_V_G(c,t)[2]),2.0)*((z/L)/(sigma_theta))*(phiH/pow(phiM,2.0)); ...
        /* DTU Formulation */
    C3e = sigma_theta/(z/L)*(phiM/phiH)*(Ce1*phiM-Ce2*phiE+(Ce2-Ce1)*pow(phiE,-0.5)*fe);
    source = Ce1*C_D(c,t)/C_K(c,t)*C3e*Gb;
    }
    else {
    C3e = 0.0;
    Gb = 0.0;
    source = 0.0;
    }

    C_UDMI(c,t,9) = source;
    C_UDMI(c,t,10) = C3e;
    C_UDMI(c,t,11) = Gb*vonKarman*z/pow(uStar,3.0);
    dS[eqn] = 0.0;
    return source;
 }

 /* ************************* Initilization ************************** */

 DEFINE_INIT(initStable,d)
 {
    cell_t c;
    Thread *t;
    real x[ND_ND];
    real phiM, phiE, phiH, pressure, potenTemp, z, zAMSL, L ;
    /* loop over all cell threads in the domain */
    thread_loop_c(t,d)
      {
      /* loop over all cells */
      begin_c_loop_all(c,t)
          {
      C_CENTROID(x,c,t);
      z = x[2] + z0;
      L = linearInterpolation(x[1]);
      if (z > ablHeight){
      z = ablHeight;
      }

      if (z > maxZInit){
      phiM = 1.0 + 5.0*(z/L);
      phiE = phiM-z/L;
      phiH = 1.0 + 5.0*(z/L);
      C_U(c,t) = 0.0; /*x velocity */
      C_V(c,t) = (uStar/vonKarman)*(log(z/z0) +phiM -1.0); /* y velocity */
      C_W(c,t) = 0.0; /* z velocity */
      /* C_T(c,t) = potenTemp/(pow(presOper/pressure,0.286)); */ /* Temperature */
      C_K(c,t) =  (pow(uStar,2.0)/sqrt(Cmu))*pow(phiE/phiM,0.5); /* k */
      C_D(c,t) = phiE*pow(uStar,3.0)/(vonKarman*z); /* epsilon */
      C_P(c,t) = 0.0; /*Pressure*/
              }
      else{
      C_U(c,t) = 0.0;
      C_V(c,t) = initVelocity;
      C_W(c,t) = 0.0;
      C_K(c,t) = initK;
      C_D(c,t) = initEpsilon;
      C_P(c,t) = 0.0;
          }
        }
         end_c_loop_all(c,t)
        }
}


 /* ************************* Wall Functions ************************** */

/* Designed around u/uStar = 1/K*log(z/z0) ref: Improved k-e model and wall function ...
    formulation for the RANS simulation of ABL flows, Parente et al
    Removes the need for multiplying z0 by 9.73/Cs and can thus use roughness lengths ...
        directly from ABL modelling with first cell height = 2*z0*/

  DEFINE_WALL_FUNCTIONS(ABL_logLaw, f, t, c0, t0, wf_ret, yPlus, Emod)
 {
    real ustar_ground, E_prime, yPlus_prime, zp, dx_mag, wf_value;
    real mu=C_MU_L(c0,t0);
    real xf[ND_ND];
    real xc[ND_ND];
    real dx[ND_ND];

    F_CENTROID(xf, f, t);
    C_CENTROID(xc, c0,t0);

    dx[0] = xc[0] - xf[0];
    dx[1] = xc[1] - xf[1];
    dx[2] = xc[2] - xf[2];
    dx_mag = NV_MAG(dx);
    zp = dx_mag;

    ustar_ground = pow(C_K(c0,t0),0.5)*pow(Cmu, 0.25);
    E_prime = (mu/densOper)/(z0*ustar_ground);
    yPlus_prime = (zp+z0)*ustar_ground/(mu/densOper);

    switch (wf_ret)
      {
      case UPLUS_LAM:
         wf_value = yPlus;
         break;
      case UPLUS_TRB:
         wf_value = log(E_prime*yPlus_prime)/vonKarman;
```

```
            /*wf_value = log(Emod*yPlus)/vonKarman; Standard Fluent*/
            break;
         case DUPLUS_LAM:
            wf_value = 1.0;
            break;
         case DUPLUS_TRB:
            wf_value =
            break;
         case D2UPLUS_TRB:
            wf_value = -1.0/(vonKarman*yPlus_prime*yPlus_prime);
            break;
         default:
            printf("Wall function return value unavailable\n");
         }
       return wf_value;
   }


   /* ************************* Interpolation ************************** */
   /* Currently does linear interpolation, Must be run with 180degree inlet location. ...
       This function can be expanded in future to bilinear (or more) to include more ...
       mast/WRF locations */
double linearInterpolation(double y)
{
      double L;
         if (y > mastLocation){
         L = Lmast;
         }
         else{
         L = (Lin*(mastLocation - y) + Lmast*(y - offsetY))/(mastLocation - offsetY); /* ...
             Local L */
         }

      return L;
}
```