Master of Science Thesis

CFD simulation of atmospheric wind gusts

A CFD simulation of three different gust models and its first effects on a NACAA 4415 airfoil

Vincent Vandecauter

August 17, 2011

Aerodynamics

Faculty of Aerospace Engineering

TU Delft

Delft University of Technology

CFD simulation of atmospheric wind gusts

A CFD simulation of three different gust models and its first effects on a NACAA 4415 airfoil

Master of Science Thesis

For obtaining the degree of Master of Science in Aerospace Engineering
at Delft University of Technology

Vincent Vandecauter

August 17, 2011

Faculty of Aerospace Engineering · Delft University of Technology

Delft University of Technology

DELFT UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF AERODYNAMICS

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance the thesis entitled **"CFD simulation of atmospheric wind gusts"** by **Vincent Vandecauter** in fulfillment of the requirements for the degree of **Master of Science**.

Dated: August 17, 2011

Supervisors:

Prof. Dr. ir. drs. H. Bijl

Dr. ir. A.H. van Zuijlen

Dr. ir. W.A.A.M Bierbooms

ir. T.P Scholcz

Preface

In September 2005 I enthusiastically began the Aerospace Engineering program at the University of Technology in Delft. Within no time the three years of Bachelor were passed and I had to choose a master track. This was a crucial choice since it would determine my specialization. Eventually, I chose Aerodynamics to be my main subject for the next two years. I've always been fascinated by race cars where I wanted to improve my chances of starting a professional career in the automobile-industry. Studying *fluid-structure interaction* at the Aerodynamics department would certainly seem to fit what I was trying to achieve.

Soon I began to realize that the beauty of aerodynamics also came at a cost. The difficulty of the masters course was significantly more challenging than that of the bachelors and as a consequence it became clear to me that aerodynamics is one of the most complex branches in physics. The numerous topics associated with the subject, ranging from hypersonic flows, turbulence, boundary layers to gas dynamics, experimental methods and others tested extensively my capabilities in problem solving particularly when using complex mathematical techniques and applying them in computer programming language. Honestly, maybe I aimed a sight to high above my capabilities when I chose aerodynamics as my 'partner in crime'. Nevertheless, it was a fascinating period where I really learned a lot at the Delft Aerodynamic Department.

The choice of the subject for my master's thesis came from two personal desires. In the first place I wanted to delve into an applied subject with some connections to the industry. Secondly, I wanted to try out the use of CFD since, for me, this was quit unfamiliar territory. The investigation into the possibilities of simulating wind gusts using CFD seemed a perfect match.

In the first place I would like to thank my parents for their unlimited support in all areas. Without them I would not even have the opportunities of studying at the TU Delft. I must also thank Sander for his help, patience and clear explanations of abstract topics. I'd also like to thank Prof. H. Bijl and the members of the jury for their time, efforts and contributions. Finally, I also want to mention a few friends; Olivier, Tom and Willem for the six fantastic years that they shared with me in Delft where their friendship was always of much support.

Summary

Since the last decade the world energy supply is shifting from fossil- to renewable energy resources. Wind energy is one of the main resources of this new type of energy. However, a lot of work still needs to be done on the efficiency improvements. This mainly results in increased rotor diameters. These large wind turbines are highly subjected to the vagaries of the atmosphere. Therefore a lot of research is being performed to the influence of atmospheric conditions on the aerodynamic and structural behavior of wind turbines. Typical atmospheric conditions that are relevant for wind turbine design are gusts, vertical velocity shear, direction change and turbulence. The highly unsteady character of these atmospheric conditions make most of available empirical models inadequate for predicting the flow behavior around wind turbine parts. CFD can be a good alternative. However, wind engineering is a relatively new part in CFD and a lot of investigations still need to be done.

This Msc thesis investigates the possibilities of simulating atmospheric gusts using CFD. Therefore two different models both suggested by the International Electrical Commission are used. The first model is the J.Mann model which is widely used in wind engineering. It defines a 3D, divergence-free, turbulent velocity field with an atmospheric character. The possibilities of using this velocity field for CFD simulations are investigated.
The second model is a (1-cos)-function that describes the horizontal velocity profile of a single, 1D cosine-shaped gust. This function will be used to generate such a single gust in a CFD domain. Three different approaches are investigated. First, an unsteady inlet boundary condition is used to impose an inlet velocity according to the velocity function. The second approach makes use of a convergent-divergent duct generated by a vertical wall inflow to create a flow acceleration according to the velocity function. In the third approach the analytical solutions of a vortex are used. A cosine-shaped gust is created by superposition of vortices.

The results from the J.Mann model showed to much dissipation and more complex CFD computations are advised. For the cosine gust model the use of a duct and the use of vortices showed best results. The latter is used to investigate its first effects on a NACA 4415 wind turbine airfoil.

Table of Contents

List of Figures

List of Tables

Nomenclature

Abbrevations

ABL	Atmospheric Boundary Layer
BEM	Blade Element Method
BEM	Blade Element Momentum
CFD	Computational Fluid Dynamics
CFL	Courant-Friedrichs-Lewy number
CGDuct	Cosine Gust Duct
CGM	Cosine Gust Model
DES	Detached Eddy Simulation
DNS	Direct Numerical Simulation
EDC	Extreme Direction Change
EOG	Extreme Operating Gusts
ETM	Extreme Turbulence Model
EWC	Extreme Wind Conditions
EWM	Extreme Wind speed Model
EWS	Extreme Wind Shear
HAWT	Horizontal Axis Wind Turbine
Hz	Hertz
IEC	International Electronical Commision
LES	Large Eddy Simulation
MSc	Master of Science
NS	Navier Stokes
NTM	Normal Turbulence Model
NWC	Normal Wind Conditions
NWP	Normal Wind Profile
PDE	Partial Differential Equation
RANS	Reynolds Averaged Navier Stokes
VGM	Vortex Gust Model

Greek symbols

Δt	Time step	[s]

ϵ_2	2^{nd} order numerical dissipation coefficient	[-]
Γ	Lapse rate	$[^\circ C/m]$
ν	Viscosity	$[m^2/s]$
ρ	Density	[kg/ m^3]
k	von Karman's constant	[-]

Roman symbols

$\bar{u}$	Mean x-velocity component	[m/s]
c	Chord	[m]
$C_L(\alpha)$	Lift coefficient	[-]
du	velocity disturbance	[m/s]
$F_{viscous}$	Viscous force	[N]
L	Lift	[N]
l	Length scale	[m]
L_g	Gust domain	[m]
N_x	Number of cells in x-direction	[-]
Re	Reynolds number	[-]
U^*	Friction velocity	[m/s]
V_g	Gust velocity	[m/s]
W_g	Gust amplitude	[m/s]
z	Altitude	[m]
z_0	Surface roughness length	[m]
z_r	Reference velocity	[m/s]
e	Energy	[Joule]
p	Pressure	[Pa]
Q	Energy addition	[Joule]
q	Dynamic pressure	[kg/ms^2]
t	Time	[s]
u	x-velocity component	[m/s]
V	Velocity	[m/s]
v	y-velocity component	[m/s]
W	Work	[Joule]
w	z-velocity component	[m/s]

Chapter 1

Introduction

For the last two decades it became clear that renewable resources will play a more dominant role in the future energy suppliers. Global warming and the drop in oil supply lead to an increase in the demand for wind energy. Although, first wind turbines are more than 40 years old, the demand for more efficient and powerful turbines increased only last decade, mainly resulting in an increase in turbine size. The prediction of the aerodynamic behavior then becomes a challenging task because of the complexity of its aerodynamics. Large wind turbines can have diameters up to 120m and as a result the turbine blades are exposed to different wind regimes of the atmospheric boundary layer. In particular, wind turbine blades may experience large changes in angle of attack associated with sudden large gusts, change in wind directions, atmospheric boundary layer effects or interaction with the unsteady wake shed from the tower support. These changes in aerodynamic behavior will have important effects on the structural performances and power performances of the wind turbine. Nowadays, computers are powerful enough to help improving wind turbine design by calculating the flow around the turbines using computational fluid dynamics (CFD) .
Last years a lot of research is done to the aerodynamics of wind turbines. The National Renewable Energy Laboratory (NREL) in the US and the National Laboratory for Sustainable Energy (Risø) in Denmark are specialized in wind turbine aerodynamics and have made important and useful investigations and experiments. The primer [3] has made unsteady aerodynamic experiments and investigations to the three-dimensional dynamic stall response to wind turbine operation conditions and the influence of the tower. The latter [13] made simulations - based on BEM method (empirical models) - of the wind shear and turbulence impact on the power performance of wind turbines.
Empirical models are very popular to make fast flow estimations and performance predictions. The drawback of these models is the need for empirical data, mainly provided by experiments. Therefore empirical models are not suitable for the prediction of unsteady phenomena which are hard to simulate with experiments. It are these phenomena that are important to improve efficiencies. CFD simulations - which are based on first principles - can be a solution for the prediction of unsteady flows and discard the need for experimental data. Currently, flow predictions over wind turbines based on CFD simulations are not yet

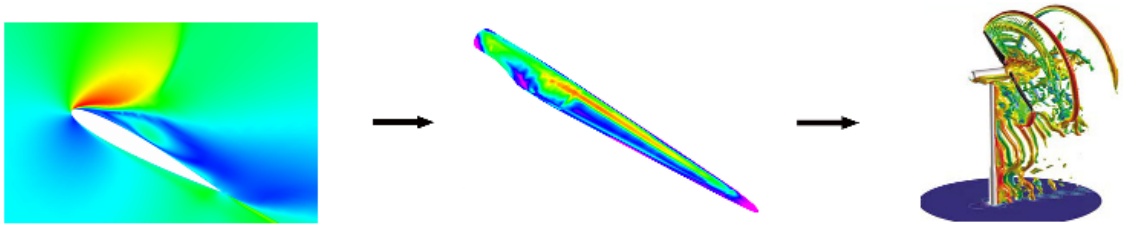


Figure 1.1: Investigation approach

widely available. Mainly due to high computational cost of accurate predictions. Although they are expensive, in the future the demand for CFD simulations will increase.
An important unsteady phenomenon to which wind turbines are often exposed are wind gusts. The effects of gusts on wind turbines is one of the main contributors that influence the design. Especially for large turbines, fatigue stresses - induced by gusts - on the structures of the tower and blades are enormous. Even effects on the gearbox should be considered. Besides fatigue problems, wind gusts also affect the power performances of wind turbines. It may not be a surprise that CFD simulations of wind turbines exposed to a gust are gaining interests.

Before the start of this master thesis the research question was "What is the influence of a wind gust on a wind turbine using CFD?". Together with this first research question some other important preliminary questions rise up. Three important key words can be found in this statement that make this research question complex; 'gust'(1), 'wind turbine'(2) and 'CFD'(3).

(1) What is a wind gust? How does it look like? How can it be simulated or modeled?

(2) When making CFD simulations it is important to start with simple cases and gradually make the configuration more complex until the complete set-up represents the real situation. Translating this to the current mission statement would result in the order of investigation presented in figure 1. First, gust effects on the basic aerodynamic characteristics of a 2D airfoil should be investigated. Then by changing to a wind turbine blade one can look to the 3D effects created by the shape of the blade, including tip effects. Finally, atmospheric effects on an entire 3D wind turbine can be modeled.

(3) An optimum should be found between accuracy and computational cost. How does the mesh look like? What flow scales should be captured? What simulation techniques should be used? How are the boundary conditions chosen? How does the model behave when convecting through a CFD domain? Is the gust simulated by the CFD simulation realistic?

During the preliminary literature study and even during the research itself the mission statement was changed. It is impossible to reveal the gust effects on a full 3D wind turbine by

CFD simulations within one MSc thesis. Therefore the mission statement developed to the title of this MSc thesis:

A CFD simulation of three different gust models and its effects on a NACA 4415 airfoil

In this MSc thesis three different gust models are tested with a CFD simulation using the *Numeca FineHexa* flow solver. The three gust models differ fundamentally in the way they construct the gust. The first gust model is an existing three-dimensional turbulent velocity field, widely known as the J.Mann model. The second model tries to generate a one-dimensional gust, based on a cosine-shape. The last model simulates a two-dimensional gust with vortices in the flow. There is investigated how the gusts behave when convecting through a CFD domain, what the issues are and what the influence is on a NACA 4415 airfoil.
This MSc thesis is divided into six main chapters of which this introduction is the first one. In chapter two some relevant background information is given to make the reader confident with different gust types and how they influence wind turbine performances. The third chapter points out the fundamental equations of gust modeling and the role of CFD. Next, also the basic concepts of CFD are explained. The fourth chapter explains for each model separately how it is implemented in the flow solver and shows the results of its convection through the domain. Also a chapter is dedicated to the gust effects on a NACA 4415 airfoil. The final chapter ends this report with general conclusions and recommendations for further investigations.

Chapter 2

Atmospheric wind gusts

In the introduction was stated that three main topics dominate this investigation; 'gust', 'wind turbines' and 'CFD'. In fact, this chapter formulates an answer to the first topic. So, before getting to the essence of this Msc thesis we take a step back and take a wider look to wind gusts in general. In this chapter the concept 'wind gust' is analysed in more detail. In the first place, we take a look to what a wind gust is, what kind of types there exist and which types are of interest with respect to wind turbines. Besides answers to these basic questions also some relative topics are drawn in and explained briefly. Finally, also two types of gust models are proposed. This chapter will help to understand what kind of problems show up during the actual investigation and why certain choices are made.

2.1 Wind characteristics

2.1.1 General definitions and characteristics

Global wind which occurs on every place on earth is moving air caused by pressure differences in the air-layer around the earth. When different regions on earth - like the poles and the equator - are uneaven heated by solar radiation, they will interact by creating an airflow between them restoring the equilibrium. Day and night alternations, ocean currents, presence of mountains and valleys, season variations cause variations in the air circulations. The circular motion of the earth and the Coriolis effect make it even more complicated but this is out of the scope. Wind circulations can be divided into different categories. There is a distinction between atmospheric motions varying in space and in time. Next, for both categories the variations take place on different scale levels.

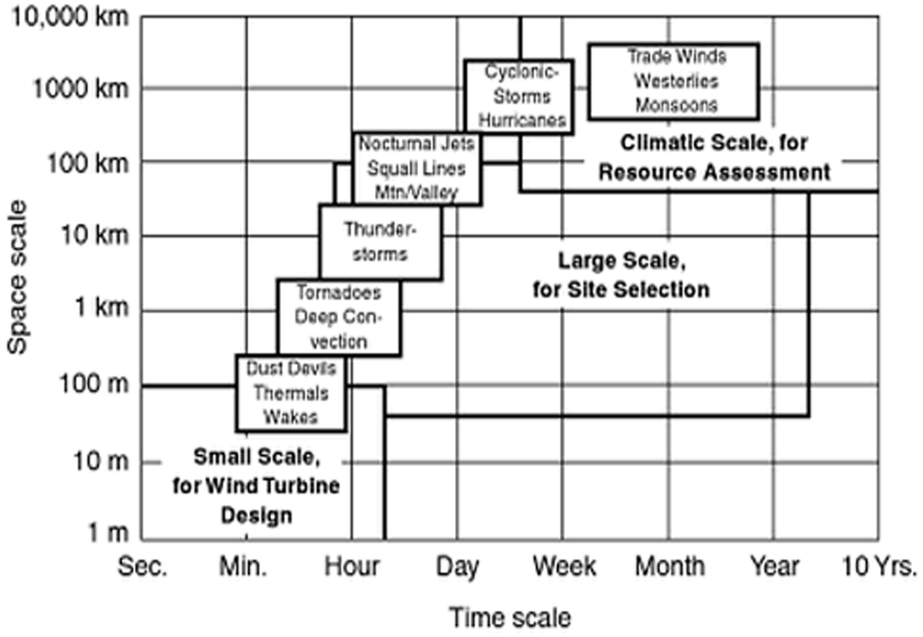


Figure 2.1: Time- and spatial-scales of atmospheric motions (Spera, 1994)

Variations in space

On the one hand there are extreme large-scale wind circulations which are known as the climate winds like the Trade and Mosoons winds. Within these large-scale winds there are smaller variations dedicated by mountains, variations in land and sea mass, variation in vegetation. More locally the small-scale wind circulations show up caused by small variations in topography and obstacles in the landscape as trees and buildings. These obstacles are responsible for earth's atmospheric boundary layer (ABL) which will be described in more detail in section 2.2. In figure 2.1 a schematic overview of the different time and space variations of atmospheric motions as applied to wind energy is presented. It shows that the small scales, let's say between 1m and 100m, of the ABL are relevant for wind turbine design. Within these length-scales new usefull disctinctions can be made which are explained in subsection 2.1.2.

Variations in time

Also in time a difference can be made between large- and small-scales. The largest wind scales are larger than one-year and therefore named the inter-anual variations. They are very hard to predict because it takes many years of data collection before patterns can be discovered. The most famous inter-annual variations are El Nino and the North Atlantic Oscilations. Smaller wind scales are generaly well known as the season winds like the Hurricane-season in the U.S.. The diurnal variations are even smaller and known as the wind variations during day and night. Especially in the mountains the diurnal pattern can differ significantly. In figure 2.2 the wind speed variation during 24 hours is shown. On still shorter time-scales of

minutes and seconds gusts and turbulence will cause the variations in wind speed and wind direction. An overview of different wind speed peaks at different scales is presented in figure 2.3. Use is made of wind spectrum in terms of frequencies. Time-scales ranging from 10s up to 10min are relevant for this research.

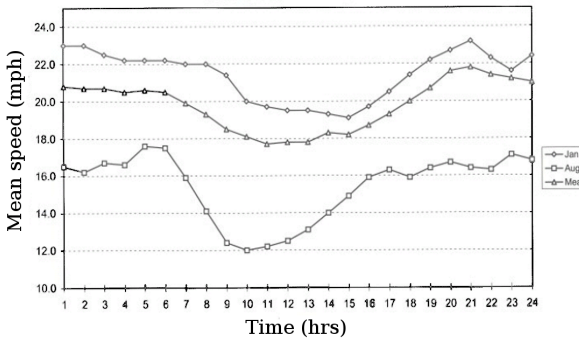


Figure 2.2: Diurnal mean wind speeds vermont - NRG sites 1997-1999

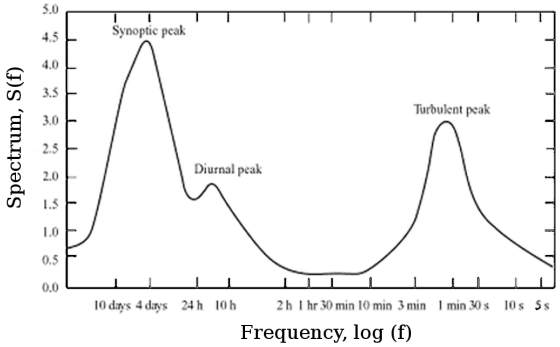


Figure 2.3: Wind spectrum 'Farm Brookhaven' based on work by van der Hoven (1957)

2.1.2 Turbulence and wind gusts

It was already mentioned in the previous paragraph that turbulence and gusts are the smallest-scale circulations in the atmosphere. Turbulence and gusts are short-term wind variations mainly created by two causes. On the one hand by friction with the earth's surface through small irregularities like trees, hills, lakes and buildings; on the other hand by small thermal differences which can cause hot air to rise and cool air to decline. Most of the time these two phenomena are interconnected.
Turbulence is a complex process. However, for this subject it can be thought of a mean wind flow determined by a season during a small interval of about a few hours during the day, superimposed by small variations over time intervals of 10 minutes or less. It is general accepted in wind engineering that variations in wind speed over a time interval from less than a second to 10 minutes and having a stochastic character can be determined as turbulence. Ten-minute averages are typically determined using a sampling rate of about one to ten Hz. [11]
A gust is then defined as a single discrete event within a turbulent wind field. It is characterized by amplitude, rise time, maximum gust variation and lapse time. See figure 2.5. Notice the small negative gust before the highest peak. According to [1] this negative gust has a mean magnitude of about 25% of the main gust. In figure 2.4 a typical time series of a turbulent wind field is shown at a sampling rate of 8 seconds.
So, it is important to understand that the small turbulent scales of the ABL are of great importance for wind turbines. When looking to the scales at wind turbine level, these gust-

scales are not small anymore. On the contrary, 1m to 10m flow scales for a wind turbine are rather large. However, an important difference can be made again between a full 3D wind turbine and a 2D blade section. The effects of a 10m-gust and 1m-gust on a full 3D wind turbine will be different than on a 2D blade section. The smallest turbulent scales (˜ 0.1m) are not considered in this investigation.

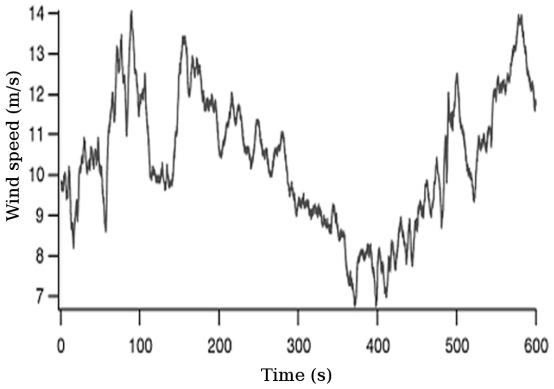


Figure 2.4: Sampled wind data (8Hz)

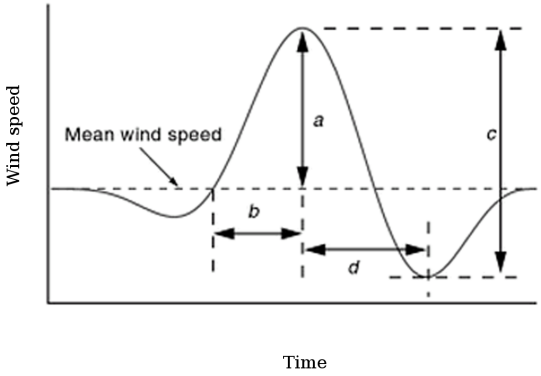


Figure 2.5: Illustration of a discrete gust event

2.1.3 Wind characteristics and wind turbines

During the design of a wind farm or a single wind turbine all wind scales will have an influence in a particular stage of the design process. In the very beginning, the large-scale wind characteristics will help to decide whether a specific site is suited for having wind turbines. On the one hand, the anual average wind speed should be high enough to produce a minimum amount of electricity, on the other hand the wind turbines should be capable of surviving a hurricane or severe Siberian wind storms. Large atlases describing the wind behaviour with power spectral density functions and data from anemometers are available. Figure 2.6 is an example of such a map.
For the design of the wind turbine itself the small wind scales are of great importance. As can be noticed from figure 2.3, the turbulent peak is located on a time scale between 1s and 10 minutes. Outside this domain, the fluctuations in wind speed are not relevant anymore for the structural and power performances of the turbine. Turbulence is then defined as the largest wind speed peaks of 10 seconds during a wind speed measurement of 10 minutes. Turbulence and gusts have a large influence on the structural design since they determine - together with some other forces - the ultimate and fatigue loads. All regulations and design conditions that wind turbines should met are gathered in the International Electronical Commision (IEC) [1]. Besides structure, turbulence also has an influence on the aerodynamic forces of a wind turbine. Especially the variation in the transition line on the blades highly influences the power performance.

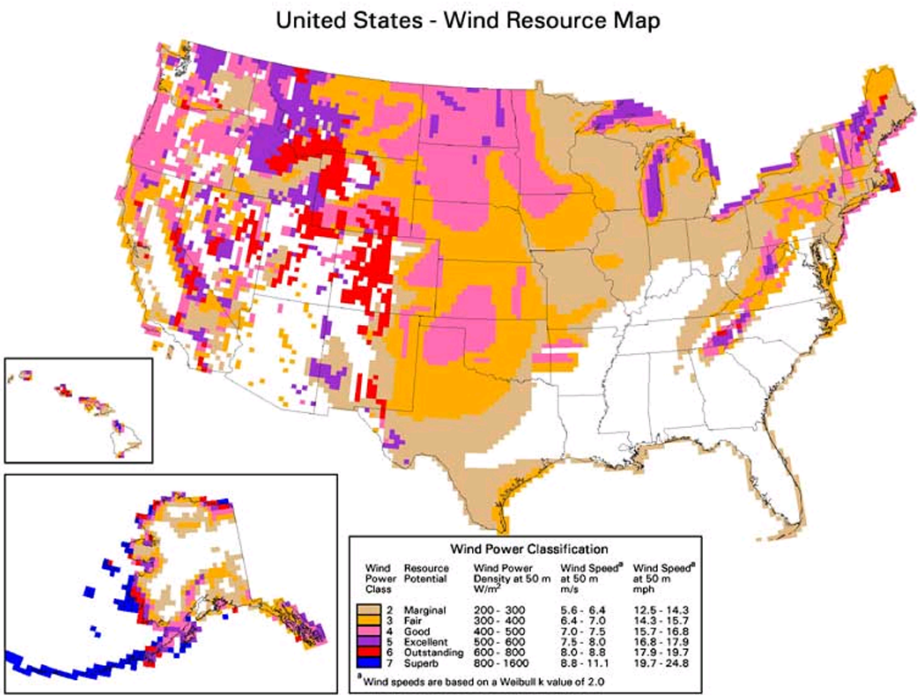


Figure 2.6: Wind resource map - U.S.

2.2 Atmospheric boundary layer

In the previous section we mainly talked about the origin of wind, the different wind scales and the relevance for wind turbines. It was also shortly mentioned that there is a relation between earth surface irregularities, turbulence and gusts. In fact this phenomenon is part of a larger story entitled the Atmospheric Bounadary Layer. The ABL describes the variation in horizontal wind speed and temperature with increasing altitude. This layer results from the friction between the horizontal wind flow and earth's surface. In wind energy the ABL is an important parameter; it directly influences the productivity of a wind turbine of a certain height and it also strongly influences the lifetime of the rotor blades. These two effects result from the change in wind speed in vertical direction. In the following paragraphs some relevant topics are shorthly clarified since they characterise the ABL.

Lapse rate

The lapse rate (Γ) is the rate at which temperature decreases with increasing altitude and can be simply derived from Newton's law applicated to a dry ideal air element in the gravitational atmosphere. The full complete derivation can be found in [11]. The international standard

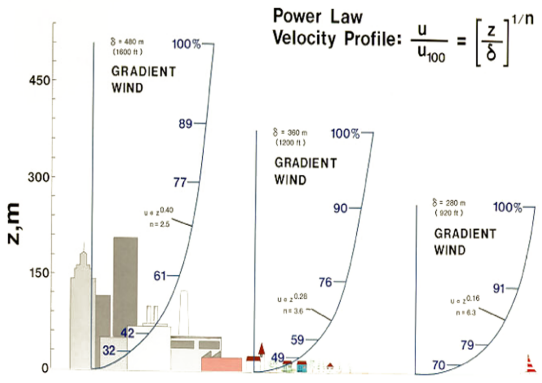


Figure 2.7: Visualization of ABL

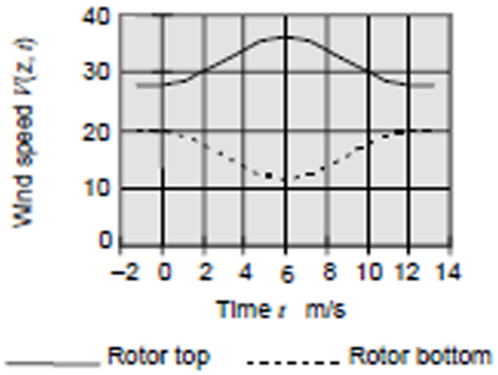


Figure 2.8: Example of wind speed at rotor and bottom ([1])

atmospheric lapse rate, based on meteorological data has been defined as:

$$\Gamma = \frac{0.0066°C}{1m} = \frac{0.66°C}{100m} \tag{2.1}$$

Velocity profile

In wind engineering two methods are widely used to describe the variation in horizontal wind speed with the altitude. A first method described here is the log law, a method based on the combination of theoretical and empirical research. The expression is given by Wortman (1982).

log law:

$$U(z) = \frac{U^*}{k} ln\left(\frac{z}{z_0}\right) \tag{2.2}$$

where U^* is the friction velocity, $k = 0.04$ (von Karman's constant), z is the altitude and z_0 is the surface roughness length. In table 2.1 some surface roughness lengths for various terrain types are given.

While te previous model is a logaritmic profile, the second model is a power law estimation. This model is very simple and can be used when no information about the surface roughness is available.

power law:

$$\frac{U(z)}{U(z_r)} = \left(\frac{z}{z_r}\right)^{\alpha} \tag{2.3}$$

Table 2.1: Values of surface roughness lengths for various terrain types

Terrain Description	$z_0(mm)$
Very smooth, ice	0.01
Calm open sea	0.20
Blown sea	0.50
Snow surface	3.00
Lawn grass	8.00
Rough pasture	10.00
Fallow field	30.00
Crops	50.00
Few tress	100.00
Many trees, hedges, few buildings	250.00
Forest and woodlands	500.00
Suburbs	1500
City centers with tall buildings	3000.00

where $U(z)$ is the wind speed at altitude z, $U(z_r)$ is the reference wind speed at height z and α is the exponent. This is quite complex since α is influenced by many parameters as elevation, time of the day, seasons, terrain, wind speed, temperature, etc. Some popular estimations for α are:

- α as a function of velocity and altitude (1978):

$$\alpha = \frac{0.37 - 0.088ln(U_{ref})}{1 - 0.088ln(\frac{z_{ref}}{10})}) \tag{2.4}$$

- α as a function of surface roughness (1975):

$$\alpha = 0.096log_{10}z_0 + 0.016(log_{10}z_0)^2 + 0.24 \tag{2.5}$$

  for $0.001m < z_0 < 10m$.

2.3 Gust models

It is clear that gusts are important phenomena for the design and performances of wind turbines. In this section some different types of gust models are briefly described. The reader should pay attention here and realise that these are general models describing a velocity profile. The models suggest how a gust should look like, not how they are generated.

2.3.1 IEC standards

The first model is the one used by the IEC standards [1] a document that describes the certification and testing conditions of wind turbines. Besides safety, electrics, structures, etc

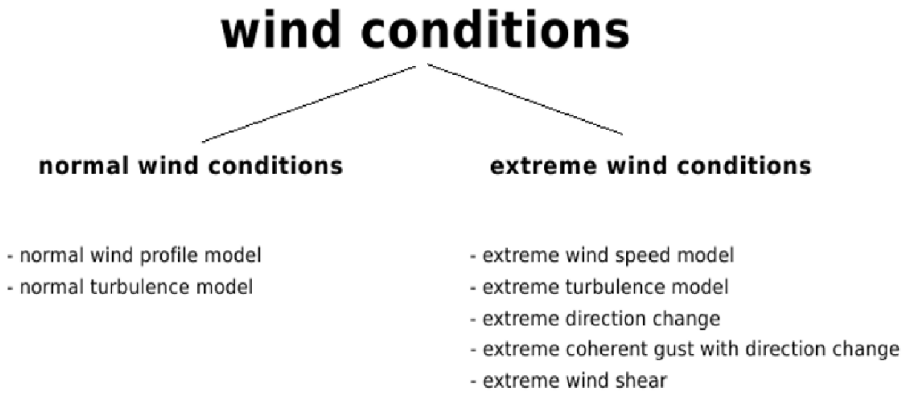


Figure 2.9: Wind conditions described by [1].

it also contains an interesting section about the different wind conditions that wind turbines should withstand. According to IEC a wind condition includes a constant mean flow combined - or more correct - superimposed with either a varying deterministic gust profile - like a $(1 - cos)$-shaped gust - or with turbulence. Turbulence denotes random variations in the wind velocity from 10 minutes averages. The turbulence includes the effects of wind speed, shears and direction and allow rotational sampling through varying shears. IEC suggests two turbulence models, namely the *J.Mann model* which is described further in this section and the *Kaimal and exponential coherence model* (not used in this report).

A distinction is made between normal wind conditions (NWC) and extreme wind conditions . Sometimes a distinction is made between a one-year-recurrence and a 50-year-recurrence. The first condition includes a normal wind speed distribution - referred to as a normal wind profile model (NWP) - and normal turbulence model (NTM) . The latter gives representative values of the turbulence standard deviation depending on wind speed and turbulence intensity. The extreme wind condition includes an extreme wind speed model (EWM) , extreme operating gusts (EOG) , extreme turbulence model , extreme direction change, extreme coherent gust with direction change (ECD) and extreme wind shear (EWS) . A schematic overview of all conditions is shown in figure 2.9. All detailed descriptions and formulas for the models can be found in [1]. For this investigation of gusts only the EOG is relevant, nevertheless, it can be usefull in the future to extend to other wind conditions. Figure 2.10 presents a gust according to IEC's EOG model for a mean wind speed of 25 m/s and a rotor diameter of 42m.

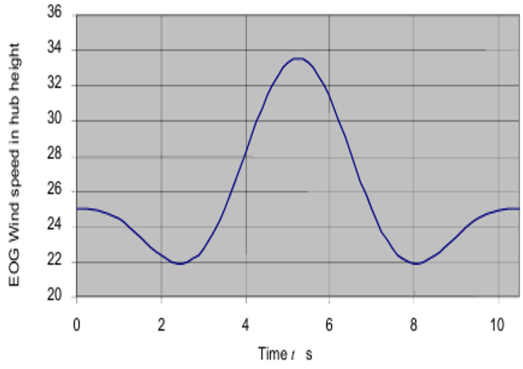


Figure 2.10: Example of extreme gust [1]

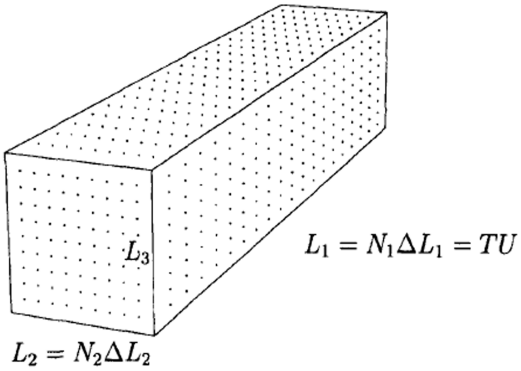


Figure 2.11: Visualization of J.Mann output domain [10]

2.3.2 Jacob Mann model

In [10] a more detailed simulation is proposed. Instead of describing a single gust Jacob Mann found an efficient algorithm to simulate a complete turbulent, atmospheric wind field. He started from the need to have a three-dimensional field of either one, two or three components of the wind velocity to make dynamic load calculations on wind turbines or bridges. The most 'correct' method would be making a CFD simulation with the lower boundary having a roughness corresponding to earth's surface. The flow will then develop into a boundary layer generating an atmospheric velocity profile. However, the computational cost of such simulation would be enormous. The domain dimensions need to be large and turbulent scales ranging from small to large need to be captured to obtain realistic results. Only DNS or LES are suited for these calculations but require nowadays still to much CPU time.
Instead empirical information can be used which Mann did. Mann's method is build on the model of the spectral tensor for atmospheric surface layer turbulence at high wind speeds [9]. The model is a widely used and respected tool in wind engineering.

2.4 Preliminary conclusion

The atmospheric boundary layer has a large impact on the design and performance of wind turbines. Nowadays, to improve wind energy's efficiency the size of wind turbines is increasing up to more than 100m diameter. The higher the wind turbine, the larger the effect of the ABL on the wind turbine. Basically, two effects are relevant which both due to this atmospheric boundary layer.
The first phenomenon is the horizontal wind gust resulting from turbulence in the ABL. Generaly spoken, gusts are the highest wind speed peaks of a 10 minute measurement. An extreme operating gust is characterized by increasing the mean wind speed up to 30% within a time interval of 10s. The length scales range from 1m to 10m. Typical values of turbulence

intensity are within the range of 0.1 and 0.5. This horizontal varying gust is interesting when investigating 2D blade profiles.
Besides the impact of gusts, also the other side effects created by the ABL are important. Vertical shear, turbulence and gust-direction-change are in fact three dimensional phenomena which will affect more the complete three-dimensional wind turbine rather than a blade profile. Both atmospheric phenomena are described in the IEC-standards, the official regulations for design and testing. For all wind and gust conditions that were mentioned in this conclusion also a sort of mathematical model is given. These models describe the velocity profile which is not a physical flow model containing velocity, pressure, temperature and density. For the horizontal gust a (1-cosine)-function is proposed, while for the atmospheric turbulence and the shear effects the J.Mann model is proposed. The two gust types are be summarized in 2.2.

	Horizontal gust	**Gust atmospheric turbulence**
Dimension	2D	3D
Relevance	blade profile	full wind turbine
Length scale	1m∼10m	10m∼100m
Time scale	∼1s	1s∼10s
Proposed model	Cosine Gust Model	J.Mann model

Table 2.2: Relevant gust types and scales

Chapter 3

Gust modeling

3.1 Introduction

The previous chapter ended with a suggestion for two types of gust models. Remember that these models only describe a velocity profile. To investigate the effects of gusts on a wind turbine or a blade section the flow velocity needs to be translated to a force or other physical effect. In the first section of this chapter the relation between velocity and its effect on a blade section (airfoil) is described by analytical equations. After, a section is given about the computational fluid dynamics used to simulate the gust using the gust models.

3.2 Analytical equations

In general the lift (L) is calculated by the following formula

$$L = C_{L(\alpha)} \frac{1}{2} \rho V^2 c \tag{3.1}$$

and depends on the lift coefficient ($C_{L(\alpha)}$) , density (ρ), velocity (V) and chord (c) . Since the velocity is determined by a gust model which depends on time (t), the following expression for the horizontal and vertical gust component respectively can be found as:

Horizontal gust

$$L(t) = C_L \frac{1}{2} \rho (u + du_{(t)})^2 c \tag{3.2}$$

with

$$C_L = f(\alpha) \tag{3.3}$$

and

Vertical gust

$$L(t) = C_L \frac{1}{2} \rho (u + du_{(t)})^2 c \tag{3.4}$$

with

$$C_L = f(\alpha + \frac{dv_{(t)}}{u}) \tag{3.5}$$

For gusts with a relative large gust length (L_g) w.r.t the airfoil chord the lift will be only a function of the dynamic pressure (q) . For gusts with a relative small gust length also the variation in effective angle of attack will affect the lift coefficient and the lift. In wind engineering Blade Element Momentum theory is often used to estimate the aerodynamic forces on a rotor blade. By discretizing the blade and calculating the aerodynamic forces for each section apart, a resulting force for the whole blade can be determined. The theory uses both axial and angular momentum balances to determine the flow and the resulting forces at the blade. But, because of its assumptions, this analytical approach has several drawbacks. The major shortcomings result from the discretization, the influence of the flow-far-field and the neglected fluid structure interaction. Besides that, also the influence of unsteady phenomena as dynamic stall and vortex shedding is neglected. However, over the past several corrections were found to compensate some of the assumptions. CFD computations can be a solution. The use of first principles make it is possible to obtain a physical correct solution according to the flow assumptions. Moreover, CFD is able to capture the coupling between gust, flow and airfoil. Many shortcomings of an analytical approximation are then not existing anymore.

3.3 Computational fluid dynamics

It was already mentioned in the objectives of the thesis that CFD is used to simulate the wind gust and the flow around the airfoil. Computational fluid dynamics is a quit new discipline that solves the Navier Stokes equations (NS) in a numerical way on a discretized domain. The increase of computer power during the last two decades of the twentieth century made CFD a promising tool for aerodynamicists. The beauty of CFD - in contrast to the classical, analytical methods - is that it has no restrictions regarding physical or geometrical approximations. CFD can solve problems that are impossible to solve by analytical solutions. The Navier Stokes equations are a set of non-linear partial differential equations (PDE) describing the physical dynamics of a fluid by means of fluid velocity and pressure. These relations between pressure (p) and velocity (V) dependent on time (t) and space (x,y,z). It contains three momentum equations, one mass conservation law and one energy conservation law, all shown from equation 3.6 to equation 3.10.

Conservation of mass:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \underline{V}) = 0 \tag{3.6}$$

Momentum equation:

$$\frac{\partial \rho u}{\partial t} + \nabla \cdot (\rho u \underline{V}) = -\frac{\partial p}{\partial x} + \rho f_x + F_{x_{viscous}} \tag{3.7}$$

$$\frac{\partial \rho v}{\partial t} + \nabla \cdot (\rho v \underline{V}) = -\frac{\partial p}{\partial y} + \rho f_y + F_{y_{viscous}} \tag{3.8}$$

$$\frac{\partial \rho w}{\partial t} + \nabla \cdot (\rho w \underline{V}) = -\frac{\partial p}{\partial z} + \rho f_z + F_{z_{viscous}} \tag{3.9}$$

Energy equation:

$$\frac{\partial}{\partial t}\left[\rho\left(e + \frac{V^2}{2}\right)\right] + \nabla \cdot \left[\rho\left(e + \frac{V^2}{2}\right)\underline{V}\right] = \rho \dot{q} - \nabla \cdot (p\underline{V}) + \rho(\underline{f} \cdot \underline{V}) + \dot{Q}_{viscous} + \dot{W}_{viscous} \tag{3.10}$$

Together with CFD some additional concepts show up. It is to complex to explain all details in this thesis. Reference [4] gives a good summary about the theory behind CFD. Here only the most important concepts are shortly mentioned.
Before the NS equations can be solved numerically, the fluid domain needs to be discretized by a mesh. The choice of the mesh is of great importance for the end result and therefore care should be taken about that. It highly depends on the type of problem, the required accuracy and the expected result. The Numeca flow solver uses a finite volume method to solve the spatial discretization.
Where grids take care of the spatial discretization of the problem, also a way to discretize time is needed when the problem is unsteady. Therefore mathematical time-discretization schemes are developed. In this investigation the flow solver uses a second-order implicit time scheme to solve the discretization.
Besides adapting the spatial- and time discretization to the problem type, also the governing equations can be modified to simplify the calculations and to reduce the computational costs. This results in different CFD methods which are explained briefly in the following section.

3.3.1 CFD simulation techniques

Solving the full Navier Stokes equations is a very time consuming task. In the past different techniques are developed to reduce the CPU cost of a CFD simulation. A classical measure is to simplify the equations while keeping the results accurate enough. Notice that therefore a prediction of the end result is needed to simplify the equations in a correct way. Especially to model turbulent structures in a flow a lot of techniques are developed where turbulence models are introduced. Although the theory behind turbulence is out of the scope of this thesis a small overview of the most widely used methods is given. For further investigations in the future to the effect of atmospheric turbulence on wind turbines, turbulent flow will become important. Below, the four most widespread types of approach are discussed, ordered after level of detail contained in the solution.

DNS stands for Direct Numerical Simulation. No turbulence models are used to solve the Navier Stokes equation. Using DNS all fluctuations in the flow are completely resolved. In theory, this is the perfect method to solve the dynamics of a flow. In reality the computational cost of DNS is way to high to be efficient. It is basically only used for fundamental research and only in combination with simple geometries and low Reynolds numbers. According to [2] the CPU time for a DNS scales as

$$\text{CPU time} \propto \mathrm{N_x N_t} \sim \left(\frac{\mathrm{T}}{\mathrm{l/u}}\right)\left(\frac{\mathrm{L_{box}}}{\mathrm{l}}\right)^3 \mathrm{Re}^3 \tag{3.11}$$

In figure 3.1 Davidson's estimated computer times are presented for a simulation in a periodic cube at a computing rate of 1 teraflop. The total run time is taken to be five eddy turn-over times, $T = 5\,(l/u)$, and the resolution is $\Delta x \approx 2\eta$. Note that $Re = ul/\nu$ where l is the integral scale.

LES stands for Large Eddy Simulation and is a half way-house between DNS and methods using turbulence models. Large eddies extract energy from the mean flow while the small-scale isotropic turbulence extracts energy from dissipation on molecular level. The latter one is then modeled by sub grid turbulence models. In engineering applications most of the time only the large scale eddies are of great interest. They dominate the transfer of momentum, heat and chemical pollutants. Since 99% of the computation effort of a DNS goes to computing the small-to-intermediate scales, LES is a much more attractive proposition than DNS. However, LES also has a high CPU cost; in order to capture all eddies high grid resolutions and small time steps are required.

RANS stands for Reynolds Averaged Navier Stokes equations. When using RANS only the largest and slowest variations in the flow are completely resolved. The small-scale turbulence is modeled by a turbulence model. When only very large scales are of great importance, this approach is very popular because of its low CPU cost. However, for boundary layers, transitions and wake-details, LES is more suitable than RANS. Important is still the choice of the appropriate model. It might also be the most difficult part and it depends on the type of

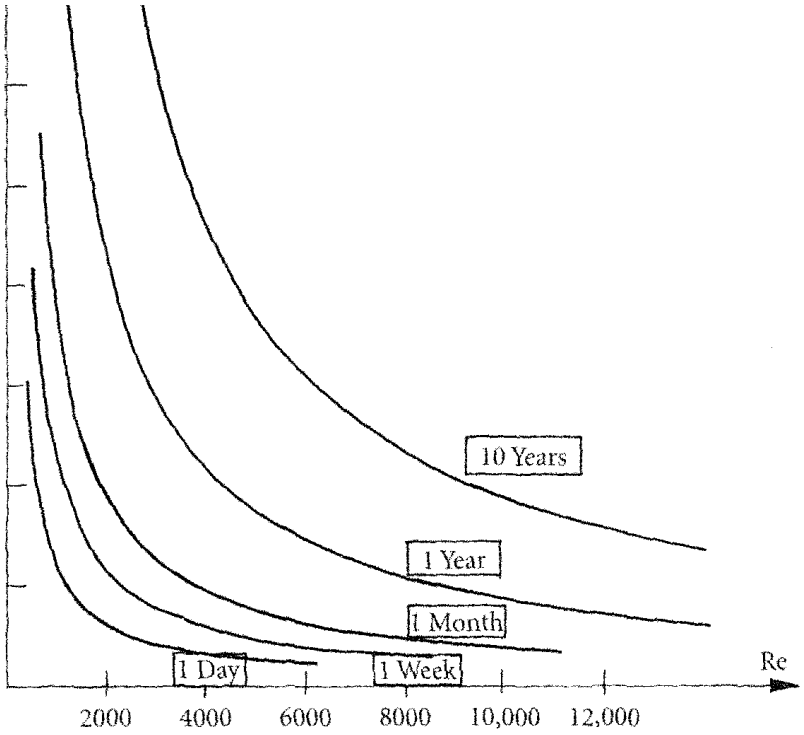


Figure 3.1: Estimated computation times for DNS.

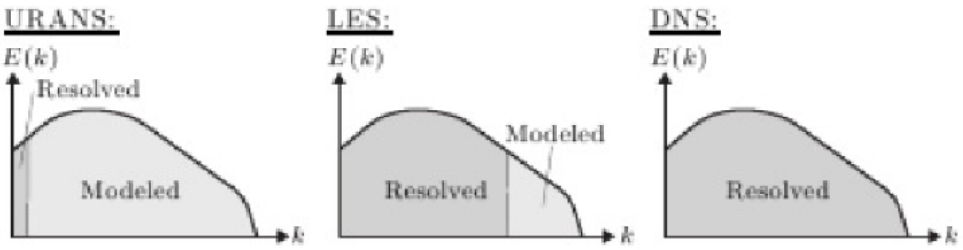


Figure 3.2: Modeled and resolved scales for different turbulent modeling strategies.

flow, angle of attack, Reynolds number and on what is expected to occur. Basically two types of models can be distinguished. There are models based on the manner how Reynolds stresses are expressed (algebraic models). The Reynolds stress tensor is then computed using an assumption that relates the Reynolds stress tensor to the velocity gradients and the turbulent viscosity. Other models are based on how velocity and length scales are described. The latter is commonly referred to as turbulent viscosity models and can be divided in three distinct levels: zero-, one- and two-equation models. In [14] and [12] some turbulence models are evaluated with respect to wind turbine aerodynamics. The following turbulence are suitable for wind turbine aerodynamics but highly depend on the flow conditions:

one-equation models follow the same approach as the previous class but they extend the model by using a transport equation to solve for the turbulent kinetic energy.

- Spalart-Allmaras

two-equation models remove the need to assume a certain length- and time-scale by using two transport equations to solve for them.

- $k - \epsilon$ turbulence model (ke)
- Renormalization group $k - \epsilon$ turbulence model (RNGke)
- Steady-state two-equation transition model based on SST

The accuracy of the latter three CFD approaches discussed above can be visualized by presenting the eddies as a energy spectrum. The methods RANS, LES and DNS can schematically be characterized by comparing the range of wave numbers they resolve. Reference [2] presents them in figure 3.2.

EULER The most famous simplification of the Navier Stokes equations are the Euler equations. The simplification is not made from a turbulence point of view. This set of equations

neglects external forces, viscosity and heat conduction terms and is mostly used to make a fast, rough first simulation. The mass, momentum and energy equation are respectively:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \underline{V}) = 0 \tag{3.12}$$

$$\frac{\partial \rho u}{\partial t} + \nabla \cdot (\rho u \underline{V}) + \frac{\partial p}{\partial x} = 0 \tag{3.13}$$

$$\frac{\partial}{\partial t}\left[\rho\left(e + \frac{V^2}{2}\right)\right] + \nabla \cdot \left[\left(\rho\left(e + \frac{V^2}{2}\right) + p\right)\underline{V}\right] = 0 \tag{3.14}$$

During this thesis, all calculations were done by using the Euler equations. Since this is only, let's say a 'fundamental' investigation of the CFD simulation of gusts, it is more suited to use Euler equations. Especially when new configurations need to be tested or new methods need to be debugged the need for a fast and robust solver is a benefit.

Chapter 4

Gust models and results

4.1 Introduction

In this chapter the actual investigation is presented. Three different gust models are tested. First, the possibility of using the J.Mann model is investigated. The J.Mann model exists of velocity-data which form an atmospheric coherent turbulence field. It physically seems to be the most correct model to investigate the effects of the atmospheric boundary layer on a complete 3-D wind turbine. Secondly, the cosine-shaped gust is used to simulate a single 1D gust which is already being used in the aerospace industry. This model is a simplification of an atmospheric gust. Different approaches will be used to model the cosine gust. Finally, a gust model based on the analytical solution of vortices is tested. The aim of this approach is to obtain a cosine-shaped gust by using a combination of vortices. Let's summarize in a small overview.

1. J.Mann Model (3D)
2. Cosine Gust Model (CGM) (2D)
    - Cosine Gust Model without source term (CGNoSource)
    - Cosine Gust Model with source term (CGSource)
    - Duct Gust Model (CGDuct)
3. Vortex Gust Model (VGM) (2D)

In the upcoming sections a model description, implementation, CFD set-up and discussion of the results for all models is given.
During the investigation the following requirements should be taken into account:

- The flow entering the CFD domain will change and develop according to the physical laws dictated by the NS equations.
- A gust approaching the position of an airfoil should still be physically correct. Moreover, it should be possible to verify this.
- A gust should travel with a realistic convection speed downstream the domain.
- There should be enough space around the airfoil to investigate the aerodynamic effects of a gust on the airfoil.

When these requirements are fulfilled, the effects on an airfoil can be investigated. This will be done in the fifth chapter.

4.2 Jacob Mann Model

4.2.1 Model description and implementation

The first model used in this investigation is the Jacob Mann model. The idea of using this model was based on the work of R. Hadgi in [6]. In this paper R. Hadgi also tried to simulate turbulent wind to investigate the effects on a wind turbine blade. The main problem of his approach was the use of pure random velocity as an unsteady inlet boundary condition. The absence of a coherent structure in the horizontal and vertical dimension resulted in a rapidly dissipating signal. The random velocity fluctuations cancel each other out when entering the CFD domain. The J.Mann model might solve this problem.

Jacob Mann developed an algorithm to simulate turbulent, atmospheric wind fields that can be used for load calculations on wind turbines and bridges. It is based on his previous work [9] where he derived a spectral tensor for atmospheric surface layer turbulence. The problem of random fluctuations that R. Hadgi faced might be less severe when a coherent structure of fluctuations is used instead of random fluctuations.
Since programming the J.Mann model is a very time consuming job, use is made of a Mann-wind field generator, made by D. Veldkamp in corporation with Vestas (wind turbine company). The inputs for the Mann-wind field generator are: rotor diameter (domain size), Mann's shear parameter (Γ), domain resolution (width and height), time step (Δt) , total number of time steps and mean wind velocity ($\bar{u}$). The output is a 3-D, divergence-free velocity field formed by three data files containing velocity data for the three velocity components respectively. Each line of the output-file is data for a new time step resulting in a third dimension. When these data are combined, a "velocity box" can be constructed. In figure 4.1 a typical Mann velocity field is visualized.

The first advantage over R. Hadgi's model is the absence of randomness; the velocity in x-, y- and z-direction are correlated to each other, so there should be no direct cancellation of the signal. Secondly, the generated velocity field is divergence free, so mass is conserved. This makes the input signal suitable for a CFD calculation. Next, also the nature of the model is highly auspicious. The model corresponds to real atmospheric turbulence which means it includes several atmospheric effects like, gusts and wind shear effects.

To impose the synthetic turbulence from Mann on the mean velocity at the inlet, some changes are made to the source code of Numeca. In `CartesianSubsonicInlets.C` an extra section is added that first searchs for a `projectname.MannU`, `projectname.MannV`, `projectname.MannW` containing the velocity disturbances for components u, v and w respectively. Then, the data on the first line is imposed on the mean velocity in the cell centers of the inlet boundary face. Each time step the following line of data is used. In appendix A the extra source code is attached.
The three J.Mann-data files should be constructed before the actual simulation. By running a small precursor simulation the cell center coordinates of the inlet boundary face are stored in a data file (`projectname_coordinates_centres.dat`) in the same order as maintained by the solver. The coordinates are imported in Matlab together with velocity data from the

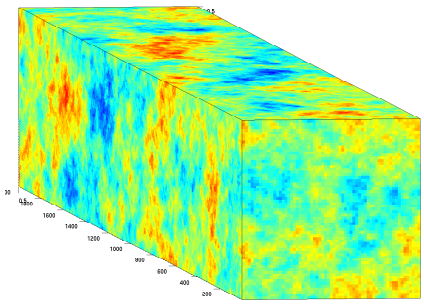


Figure 4.1: Example of Mann velocity box, 2000 time steps

Mann-wind field generator. By interpolating the velocity data over the cell centers a data file (`projectname.MannU`, `projectname.MannV`, `projectname.MannW`) can be constructed containing the data in the correct order. Now the actual simulation can be performed.

4.2.2 CFD set-up

As a starting point for the CFD set-up reference [5] is used. In this paper L. Gilling investigates the effect of resolving inflow turbulence in detached eddy simulations (DES) of airfoil flows. To generate the synthetic turbulence for the inflow also the J.Mann model is used.
In this Msc thesis a similar investigation is made preparatory to using the model to analyse its effects on an airfoil or wind turbine. We are not interested in the turbulence decay caused by viscosity but in the behavior of the coherent velocity structures propagating through the domain. The Euler equations are used to investigate the numerical dissipation of these velocity structures. The reader should pay attention to the following differences with respect to Gilling's study:

- In [5] an in-house flow solver - EllipSys3D - is used, not Numeca.
- A LES flow model is used with k-ω subgrid model, this is not possible in our in-house version of Numeca. This study is restricted to Euler equations.
- The numerical parameters are set to match their experimental data. In this study no experimental data is available.

Domain The flow domain consists of 64 x 64 x 256 cubic cells, instead of 64 x 64 x 1280 cubic cells used by L. Gilling. The smaller domain for initial simulations is needed to reduce CPU cost; if needed it can be extended afterwards. By maintaining a general cell dimension (Δx,Δy) of 0.015625m the outer domain dimensions become 1m x 1m x 4m. The mean velocity is constant and oriented along the x-axis. The synthetic turbulence is superimposed to the constant mean velocity and set as an inflow boundary condition. In figure 4.2 the

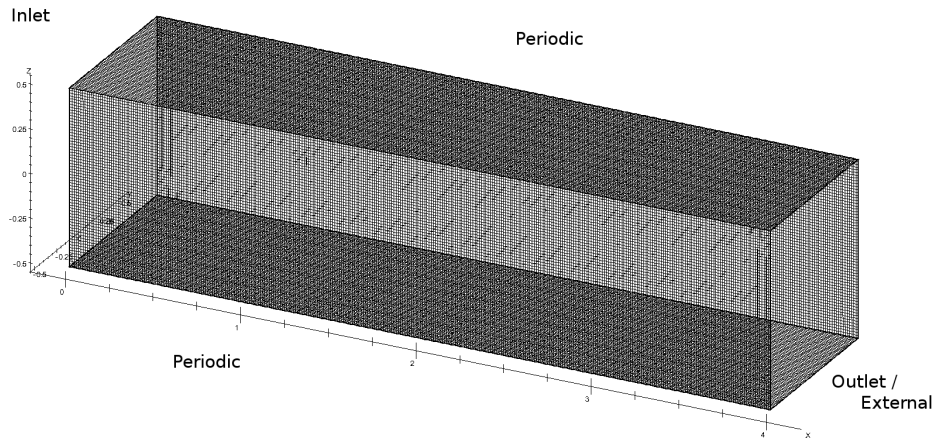


Figure 4.2: CFD domain with 1310720 cells, $\Delta x = \Delta y = 0.015625m$

domain is shown which is made by *fineHexa* and used for the CFD simulation. The boundary conditions are also shown in the figure.

Flow Model The flow model is set to *Euler* since viscous effects are not relevant for this investigation. The advantages of using these flow equations is its simplicity which leads to fast computations.
In case the full Navier Stokes are used, the flow solver uses the cell nodes, whereas Euler uses cell centers. In that case the node-coordinates (`projectname_coordinates_nodes.dat`) should be stored and new Mann-data files need to be created by Matlab in order to impose inlet velocity on the nodes instead of the cell centers.

Numerical scheme All calculations are made with a 2^{nd} order central discretization scheme which is the default scheme in Numeca.

Courant number and time-step For unsteady simulations it is important to take the Courant number into account. The Courant-condition is necessary for convergence when solving certain partial differential equations (flow equations). When explicit time-marching schemes are used the time step must be less than the time for a disturbance to travel adjacent grid points, otherwise the simulation will produce incorrect results. For a one-dimensional case, the Courant-condition is given by

$$\text{Courant} = \frac{\bar{U}\Delta t}{\Delta x} < 1 \tag{4.1}$$

where U is the mean velocity, Δt is the time step and Δx is the domain resolution (cell size). Assuming $\bar{U} = 10m/s$, $\Delta x = 0.015625m$ and $CFL = 0.7$ leads to a time step of $0.001s$.

CFD complications Before ending up with a good CFD simulation a lot of testing - sometimes with try and error - is needed to find an optimal configuration. It is a bit useless to explain all intermediate test runs. Instead in figure 4.3 a schematic overview is presented of the main issues faced before achieving a final simulation. In this section the main issues are discussed together with some measures that are - or could be - taken as a solution. Not all measures were actually applied since some of them were not possible to perform within the available time. In figure 4.3 a number is attached to each measure that refers to the discussion below. The discussion can be seen as an advise for further investigations. Note that this are not proven solutions.
The main problem that showed up when the synthetic turbulence was convecting through the flow domain was dissipation. Of course a certain dissipation is justified since turbulence decays physically in space and time. Besides that, there can also be numerical dissipation caused by the numerical procedure solving the flow equations. The boundary conditions can also lead to faster dissipation than is physically correct. It is always difficult to determine what the contributions are to the total dissipation of the synthetic turbulence. Therefore some of the felons can be reduced by some simple measures:

(1) To reduce the effects of the boundaries on the dissipation of the turbulent flow, periodic boundaries should be used in combination with a periodic synthetic turbulent flow. Velocity that leaves the domain on the left side will enter the domain on the right side and vice versa. When solid or external boundaries are used, the flow will adapt to those boundary conditions resulting in a faster dissipation of the turbulent signal.

(2) To reduce the contribution of the numerical dissipation the 2^{nd} artificial dissipation coefficient (ϵ_2) can be lowered from 1.0 to 0.2. This was advised by the Numeca developers.

A second important issue is the convergence of the solution. Some measures can speed up the convergence rate and lead to faster and more accurate solutions.

(3) In Numeca there is the possibility to calculate on multi grids. Multigrid method is a technique to yield a better convergence rate and is based on the fact that the convergence rate is a function of the error field frequency. The field frequency is the gradient of the error between two nodes. By using a hierarchy of discretizations (from coarse to fine) the convergence can be accelerated. During the study it was discovered that the multi grid function does not work in combination with the synthetic turbulence inlet and parallel computing.

(4) Instead of using a multi grid method to calculate the residual the Jacobian-Free-Newton-Krylov function can be used. It reduces the amount of iterations needed for convergence remarkably. The main draw back is the need for memory to store the precondition matrix of the system of equations. For a 4GB RAM processor the amount of cells

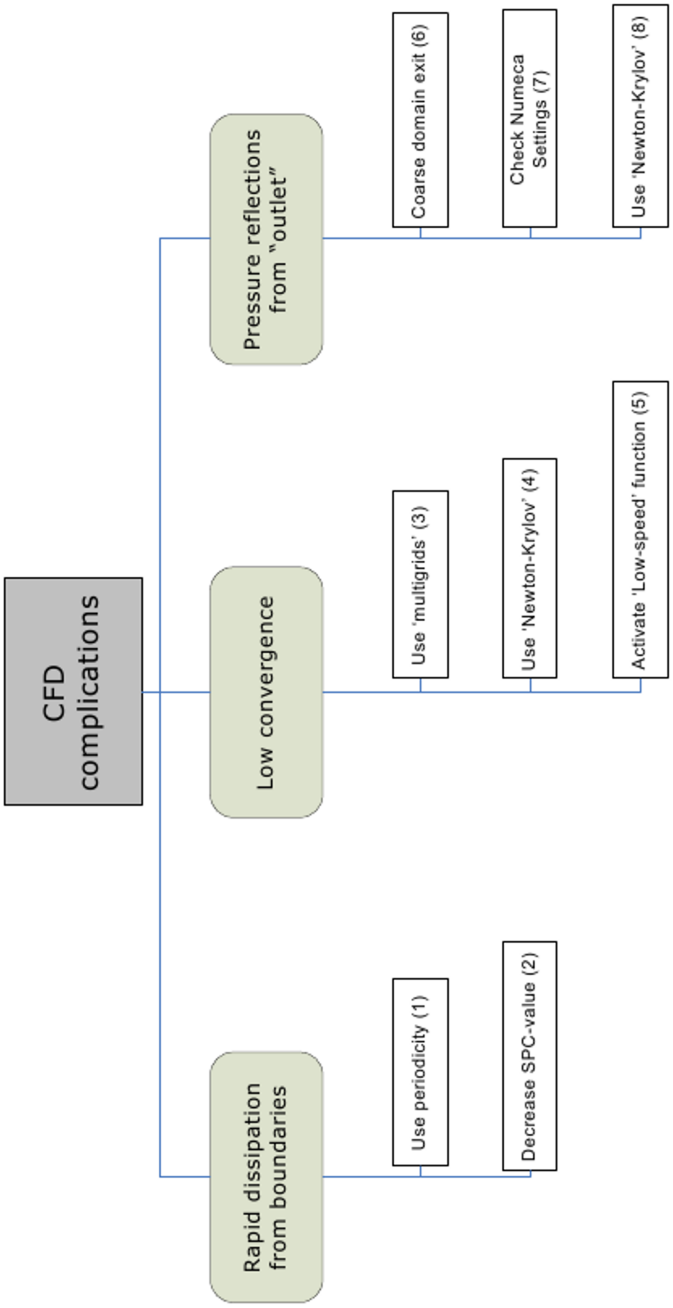


Figure 4.3: CFD complications w.r.t Mann turbulence in Numeca

setting	Case 1	Case 2	Case 3	Case 4
Flow Model	Euler	Euler	Euler	Euler
Low Speed Function	no	yes	no	yes
Boundaries	I-P-E	I-P-E	I-P-E	I-P-E
Multigrid	no	no	no	no
Δt/ #t	0.001s/1000	0.001s/1000	0.001s/1000	0.001s/1000
ϵ_2	1.0	1.0	0.2	0.2

* I-P-E = Inlet-Periodic-External

Table 4.1: CFD set-up for the test cases - J.Mann-Model

should be limited to 125000 or less. More details about the integration of this method in Numeca can be found in [8].

(5) As was already stated before the Numeca fineHexa flow solver is a compressible flow solver. For simulation at low velocity the *Low Speed Function* can be activated so a preconditioner is used. The preconditioner reduces the condition number of the matrix of the original system of equations and leads to a higher rate of convergence. More details about the preconditioner in Numeca can be found in the manual [7].

A third knotty phenomenon that showed up were pressure disturbances reflecting from the outlet. Often for short domains this effect shows up. Sometimes, it can lead to back flow or velocity disturbances and result in useless solutions.

(6) By making the domain longer and/or end the flow domain with a coarse part, the flow is dissipated artificially. Also the reflections from the outlet damp out.

4.2.3 Results and discussion

Four cases were calculated all containing different CFD settings. In table 4.1 an overview is shown of the settings for each case. For all calculations only the Euler flow equations are used. This makes calculations fast and since we are not interested in viscosity- or boundary effects, they are allowed for this type of flow calculations. In fact, two different parameters and its combination are investigated, namely the influence of the preconditioner (Low Speed Function) and the reduction of the artificial numerical dissipation coefficient. Case 1 is in fact a basic case were only default settings are used, case 2 uses the Low Speed Function, case 3 uses a lower artificial dissipation coefficient and case 4 combines both settings.

In figure 4.4 to 4.7 the flow domains are shown with cutting planes at certain intervals visualizing the u-component of the velocity field after 0.2s. The green color represents the mean velocity which was also stated as an initial condition to 10m/s. At the inlet the velocity disturbances according to the J.Mann model are entering the domain which range from -3m/s(blue) to 3m/s(red). By observing the cutting planes it is possible to see how the atmospheric turbulence travels through the domain. This can give a first idea of the dissipation.
It is expected that a mean velocity of 10m/s containing disturbances should have traveled approximately 2m after 0.2s. For all four cases can be observed that already after 1m only some weak disturbances are present, while at 2m almost no disturbances can be noticed. Shortly said, the turbulence has almost dissipated completely by traveling two meters. When comparing the cases to each other only two differences can be noticed; case 1 and 3 are equal as well as case 2 and 4. This indicates that a preconditioner influences the result remarkably, while the effect of reducing the artificial dissipation coefficient (ϵ_2) is only limited. It should be noticed that activating the Low Speed Function also changes the numerical dissipation coefficient besides only using a preconditioner.
In figure 4.8 and 4.9 the turbulent kinetic energy through the CFD domain is compared with the original Mann data used for input. For every cell face the turbulent energy from the fluctuations is calculated and presented in a graph. The graph gives an idea about how fast the energy decays compared to its original input data. Again can be observed that case 1 and 3 and case 2 and 4 have the same result. It confirms the previous conclusions; the preconditioning shows its effect while the dissipation reduction does not.
Figure 4.10 and 4.11 show the decay of kinetic energy from a Lagrangian point of view. The kinetic energy of only one moving-plane is calculated at different time steps. In [5] the log-scale of this graph is compared to the log-scale of the experimental results. Unfortunately these data are not available. However, from figure 4.10 can be observed that already after 40ms the kinetic energy has dropped drastically and after 1s almost no fluctuations are left.

For the J.Mann model this CFD configuration is not optimal to simulate gust or atmospheric conditions in a realistic manner. The distance of only 1m where the velocity structures are present is too short to investigate its effects on an airfoil. Even then, the velocity is dissipating too fast in order to represent gusts that are physically correct. For further investigations concerning this gust model more complex CFD computations are required, for example by using a precursor simulation in combination with Large Eddy Simulations. More information is given in [5]. Such further investigations might be useful, especially when full 3D wind turbines are analyzed. Due to time limitations further investigations using the J.Mann model are stopped and some other approaches are proposed in the following sections of this report.

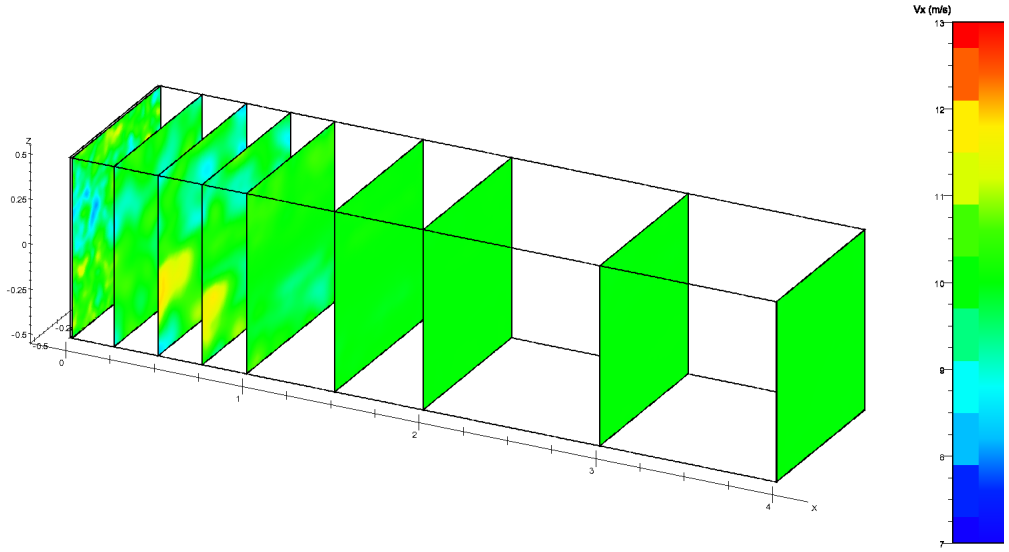


Figure 4.4: Mann model case 1, V_x-component after 0.2s

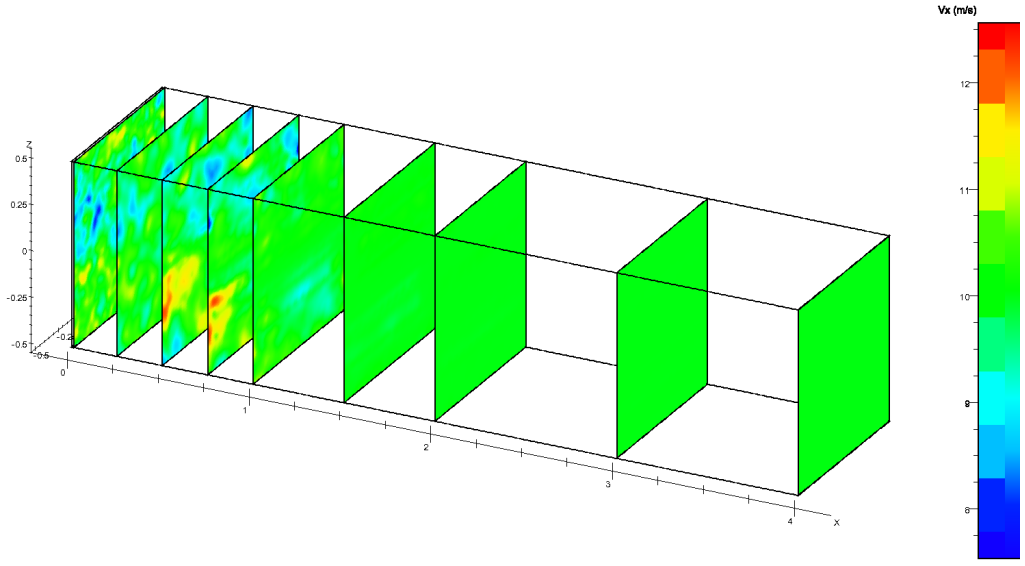


Figure 4.5: Mann model case 2, V_x-component after 0.2s

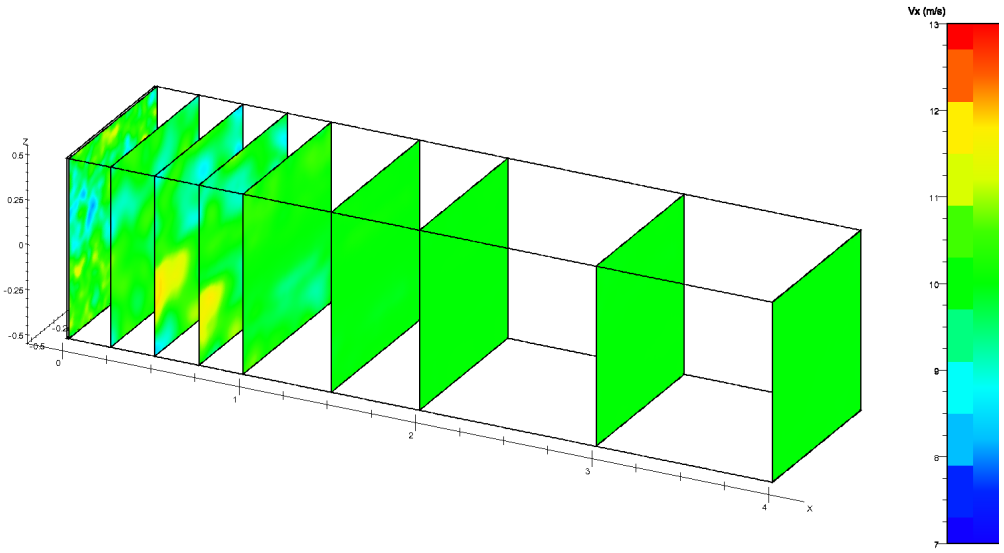


Figure 4.6: Mann model case 3, V_x-component after 0.2s

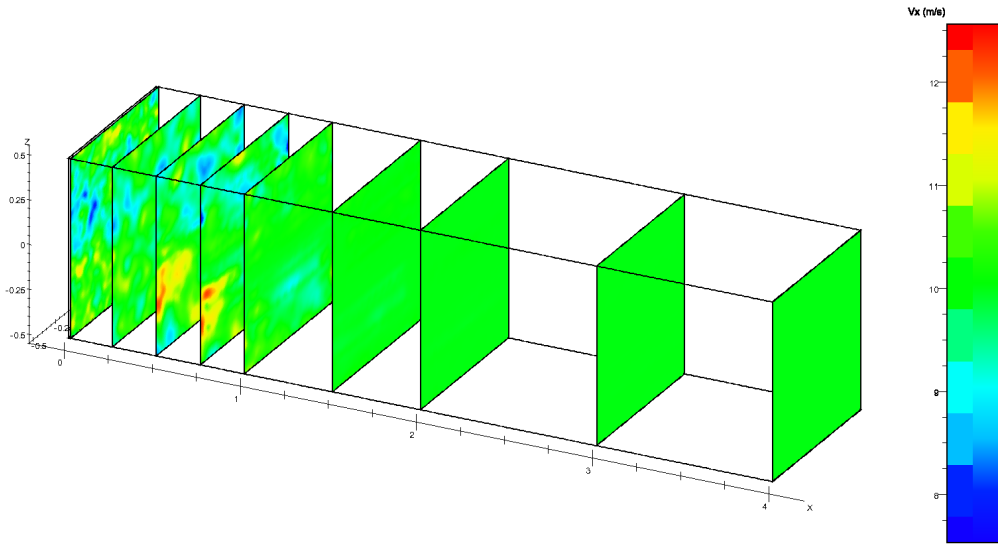


Figure 4.7: Mann model case 4, V_x-component after 0.2s

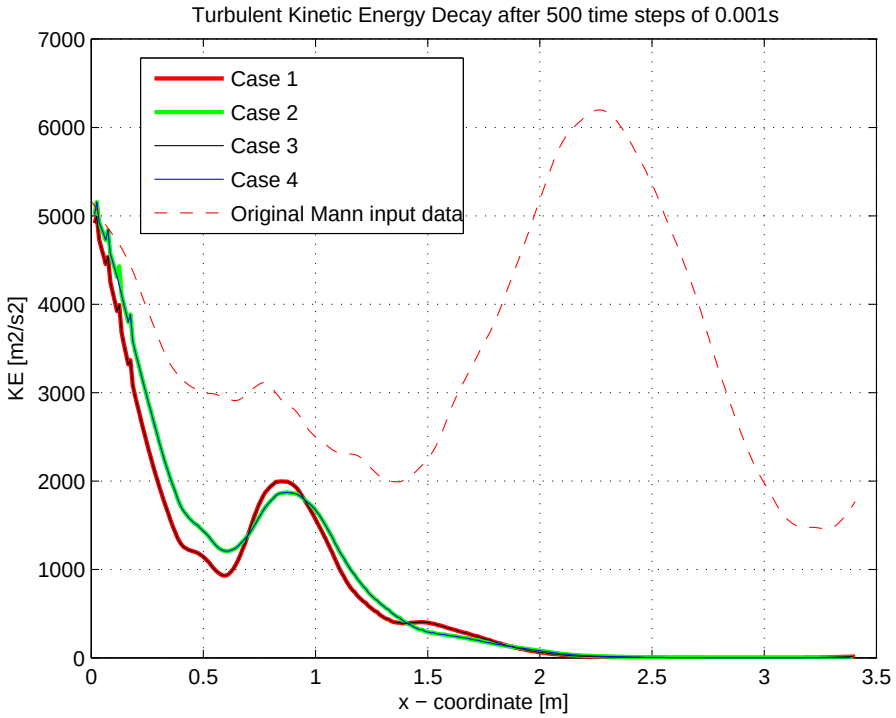


Figure 4.8: Comparison turbulent energy decay after 500 time-steps, $\Delta t = 0.001s$

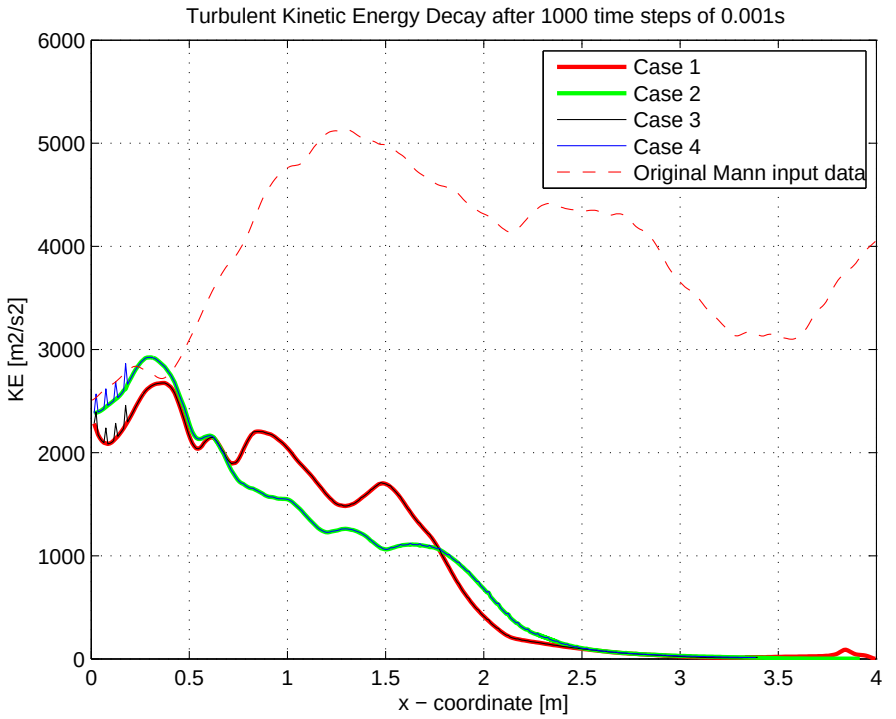


Figure 4.9: Comparison turbulent energy decay after 1000 time-steps, $\Delta t = 0.001s$

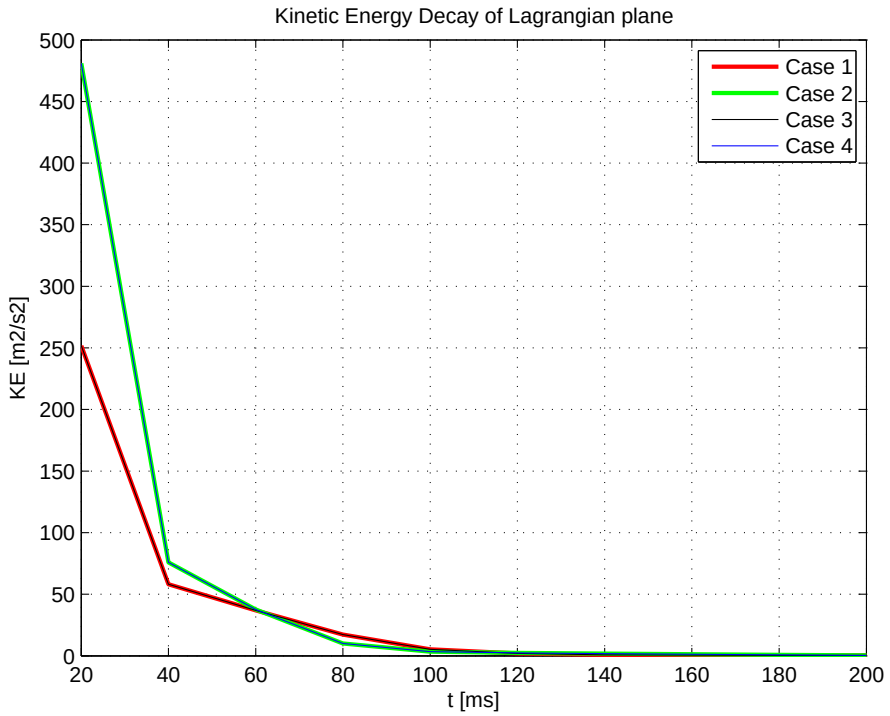


Figure 4.10: Energy decay of a Lagrangian plane

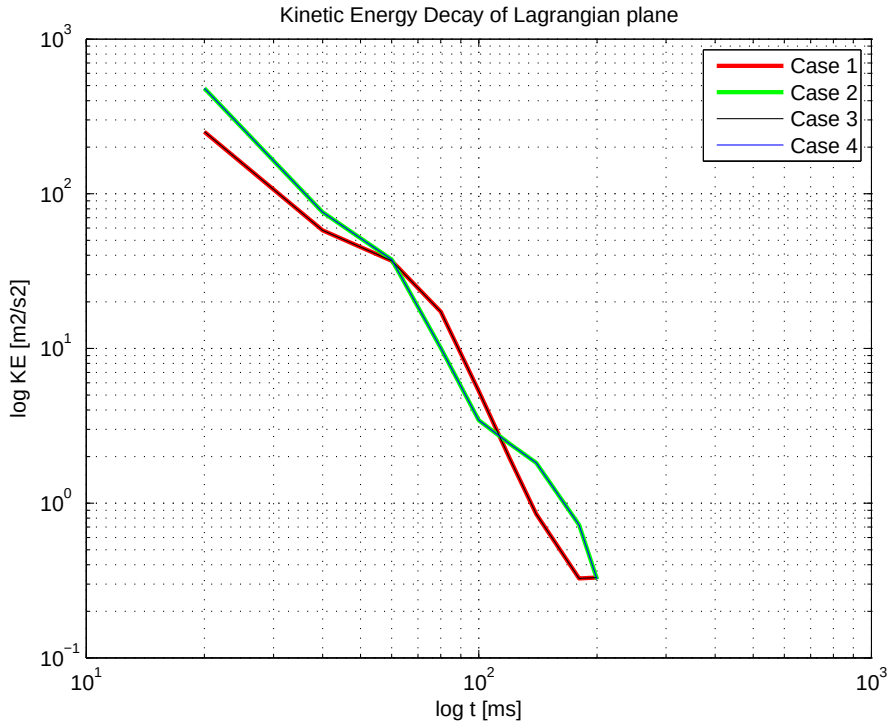


Figure 4.11: Energy decay of a Lagrangian plane log-scaled

4.3 Cosine Gust Model

4.3.1 Introduction

The J.Mann model discussed in the previous section did not work as expected and therefore more complex computations are needed. Unfortunately, due to time limitations another approach was advised; investigating another model. The model discussed in this section is named the Cosine-Gust-Model (CGM). Compared to the previous model the GCM is a simple model since it is nothing more than a function describing a velocity profile in the flow-direction. A keen aerospace engineer will recognize this model since it is based on a typical cosine-shape-gust. Cosine-shape-gusts are often used in aerospace to model wind gusts experienced by airplanes during flight. The IEC standards - international standards regarding the design and safety-requirements for wind turbines - also use this type of gust model to make load calculations. Therefore it can be assumed that the CGM used in this section is realistic, credible and accurate enough to use. The velocity profile that models an atmospheric gust is shown in figure 4.12 and is described by

$$du(x,t) = \frac{Wg}{2}\left[1 - \cos\left(2\pi(t - \frac{x}{V_g})(\frac{V_g}{L_g})\right)\right] \tag{4.2}$$

where W_g is the amplitude, V_g is the wave velocity and L_g is the gust length. In figure 4.12 this function is imposed on a mean velocity of 10m/s.

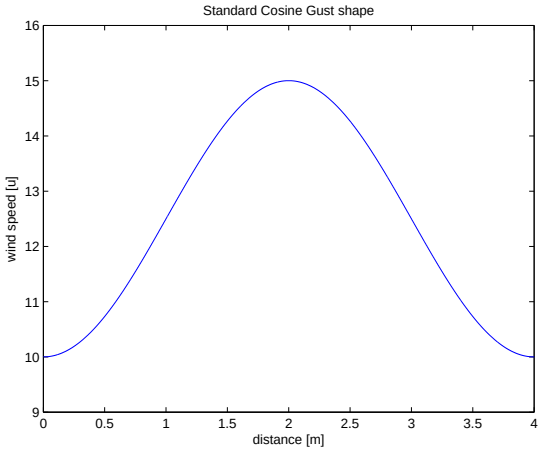


Figure 4.12: Standard cosine-gust-shape with $L_g = 4$ and $W_g = 5$.

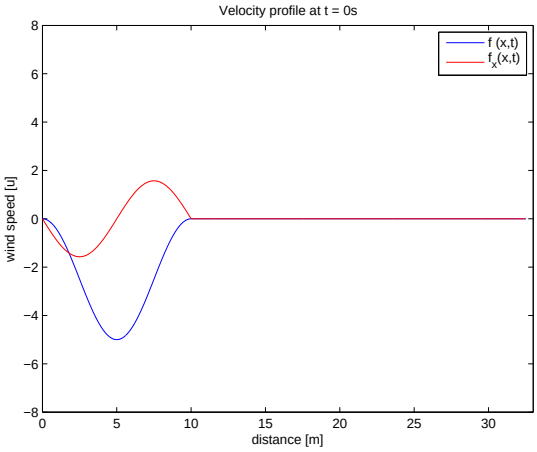


Figure 4.13: Derivative of equation 4.2

The CGM deals with some of the issues that show up when simulating a gust using CFD. There are three important main questions that should be investigated. How will the gust

propagate through the CFD domain? How will the gust-shape look like when it finally approaches the airfoil? What will be the response of the airfoil to the gust?

The nature of this type of model already shows some benefits. First of all the simplicity of this model makes it possible to observe easily the behavior of the model when propagating through the domain. No turbulence or kinetic energy has to be analyzed. Next, the model describes the velocity as a function of the x-position (x) and time (t). The benefit now is that on every position and at every time step the velocity - and also the gust shape - is known. If the flow still has the same character as described by the model when it reaches the airfoil, then it is certain that the airfoil will experience a realistic gust. This in contrast to the J.Mann model or R. Hadgi's approach, were it is impossible to verify if the airfoil experiences a realistic gust. These advantages make it worth to use this model to simulate a gust through a CFD domain.

There are several manners to introduce a velocity profile such as described by the CGM into a CFD simulation. In this section three different approaches are discussed. The first implementation (CGNoSource) is done by creating an external velocity data file according to equation 4.2 and imposing this to the inlet boundary face. This is the same way of implementation as was done with the J.Mann model. The second method (CGSource) is similar but a source term is added to the Navier Stokes equation before solving the flow. In fact it is an improvement of the first implementation. In the third method (CGDuct) the walls parallel to the flow are creating a convergent-divergent duct. Then, the mass conservation law will accelerate and decelerate the flow according to the desired velocity profile. By moving the throat the gust can travel through the domain with a desired gust velocity (V_g). In the upcoming subsections the implementation, CFD-setup and the results of the three approaches are unveiled separately.

Setting	Case 1	Case 2
Grid resolution	200 x 8 cells, $\Delta x = \Delta y = 0.25$	500 x 20 cells, $\Delta x = \Delta y = 0.1$
Time-step Δt	0.01s	0.005s
Flow Model	Unsteady Euler with Low Speed Function (default values)	
Inlet	`projectname.cosGust`	
Initial Velocity	$V_x = 10$, $V_y = 0$	
Initial Pressure	101300Pa	
Initial Temperature	293K	
External Velocity	$V_x = 10$, $V_y = 0$	
External Pressure	101300Pa	
External Temperature	293K	
Multigrid	no	
Numerical scheme	Central 2^{nd} order scalar Jameson dissipation scheme	
# iterations per time-step	50	
L_g	10m	
W_g	3.5m/s	
V_g	10m/s	

Table 4.2: CFD set-up for the test cases - CGNoSource

4.3.2 Unsteady inflow boundary condition: CGNoSource

Model description and implementation

The first approach, referred to as `CGNoSource`, is by imposing the velocity profile according to equation 4.2 on the inlet boundary. In this way new velocity information according to the velocity profile is imposed on the inlet face of the domain for every time-step. This is easily implemented in the source code (`CartesianSubsonicInlets.C`) of the Numeca flow solver. The code is included in appendix A.3. The program checks the existence of a `projectname.cosGustSource`-file that contains the parameters (L_g, W_g and V_g) for the CGM. If the file exists then the free-inlet velocity u_0 will be imposed by a velocity disturbance du calculated by equation 4.2.

CFD set-up

For testing this gust model a 2D computation is preferred because only a variation in the flow direction is expected. Two test cases are computed, having a coarse and fine mesh. The domain for both cases has a length of 50m and a height of 2m. A long domain is chosen in order to capture the behavior of a 10m long gust completely. Moreover, the x-direction is more important than the y-direction because we are only interested in the flow direction of the gust shape. Case 1 is discretized in 200 x 8 cells ($\Delta x = \Delta y = 0.25m$) while case 2 is discretized in 500 x 20 cells ($\Delta x = \Delta y = 0.1m$). The upper- and lower walls are 'Euler solid walls'. This means no boundary layers are developed. Figure 4.14 shows the domain and its discretization. The relevant Numeca flow solver settings are summarized in table 4.2.

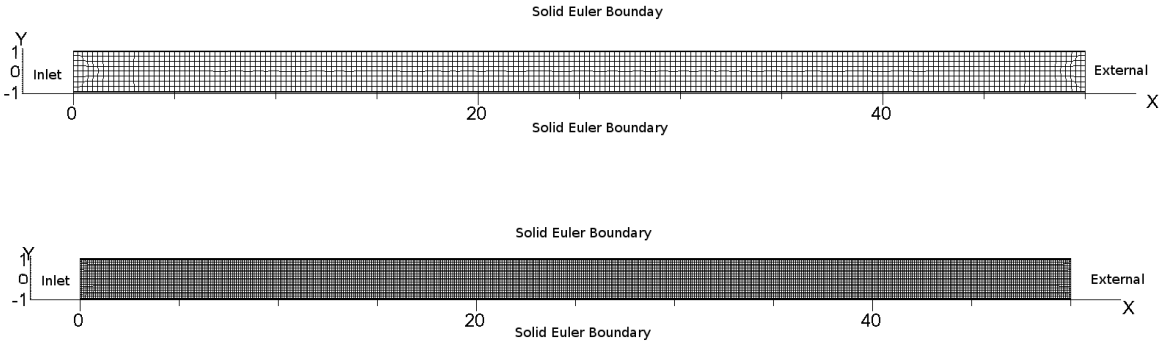


Figure 4.14: Coarse mesh: 200 x 8 cells, $\Delta x = \Delta y = 0.25m$; Fine mesh: 500 x 20 cells, $\Delta x = \Delta y = 0.1m$

Results and discussion

The inlet boundary face is fed by velocity data according to equation 4.2. It is expected to find this velocity profile again in the flow domain. The results of the CFD simulation are presented from figure 4.15 to 4.22 by visualizing the V_x-component and showing the velocity profile at y=0. The black dotted curve presents the CFD velocity profile. The red curve presents the expected theoretical velocity profile.

Already from the first plot, a velocity profile is shown that travels faster than expected. The gust is also longer than initially imposed. This behavior can be observed for all time steps. Already after two seconds the gust has left the domain of 50m. The results of case 2 show the same phenomenon. The fine mesh also destroys the vertical homogeneity of the horizontal flow. Shortly said, velocity and dimensions of the traveling disturbance are not as expected.

This behavior is caused by a violation of the conservation of mass. The conservation law for a 1D problem (only horizontal gust) is given by:

$$\frac{\partial u}{\partial x} = 0 \tag{4.3}$$

The unsteady inlet boundary condition used by this approach violates the conservation law ($\frac{\partial u}{\partial x} \neq 0$). The flow solver tries to restore the equilibrium. As a result the velocity disturbances imposed at the inlet are not propagating at a convection speed of 10m/s. Therefore it is not possible to simulate a correct horizontal gust using this approach.
On the other hand, the simulation of a vertical gust ($\frac{\partial v}{\partial y} = 0$ and $\frac{\partial v}{\partial x} \neq 0$) in a steady horizontal flow ($\frac{\partial u}{\partial x} = 0$) does not violate this conservation. Unfortunately, this is not what we want to achieve. The conservation law for a 2D problem (horizontal flow and vertical gust) is given by:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{4.4}$$

A possible solution for this problem is the addition of a source term in the Navier Stokes equations. The extra term will in fact compensate the mass violation made in the conservation law. This approach is tested in the following section.

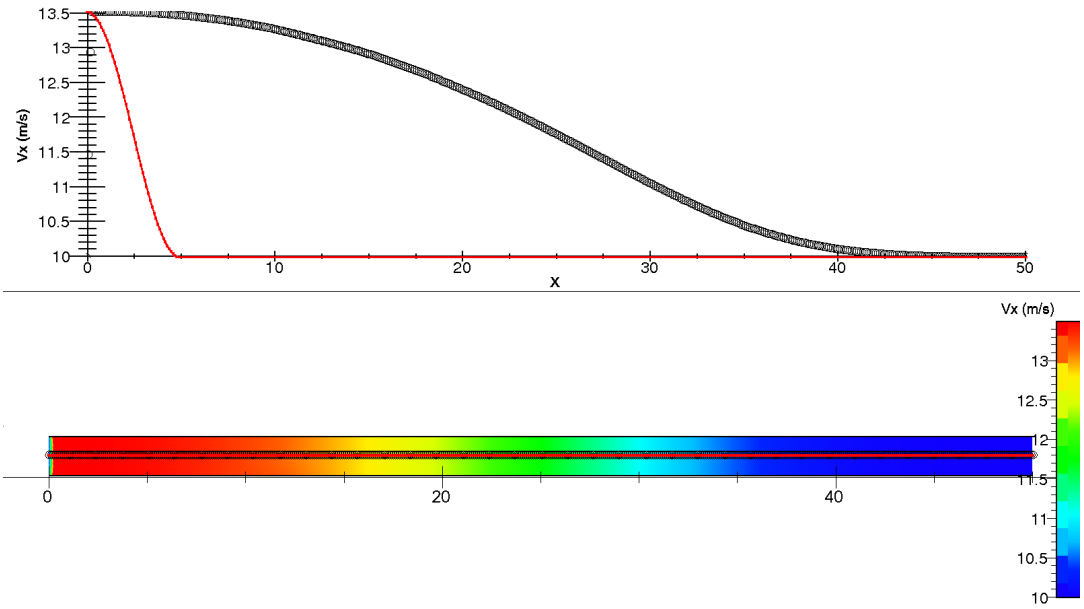


Figure 4.15: Case 1 at t=0.5s: V_x-component and velocity profile; o = CFD, □ = theoretical.

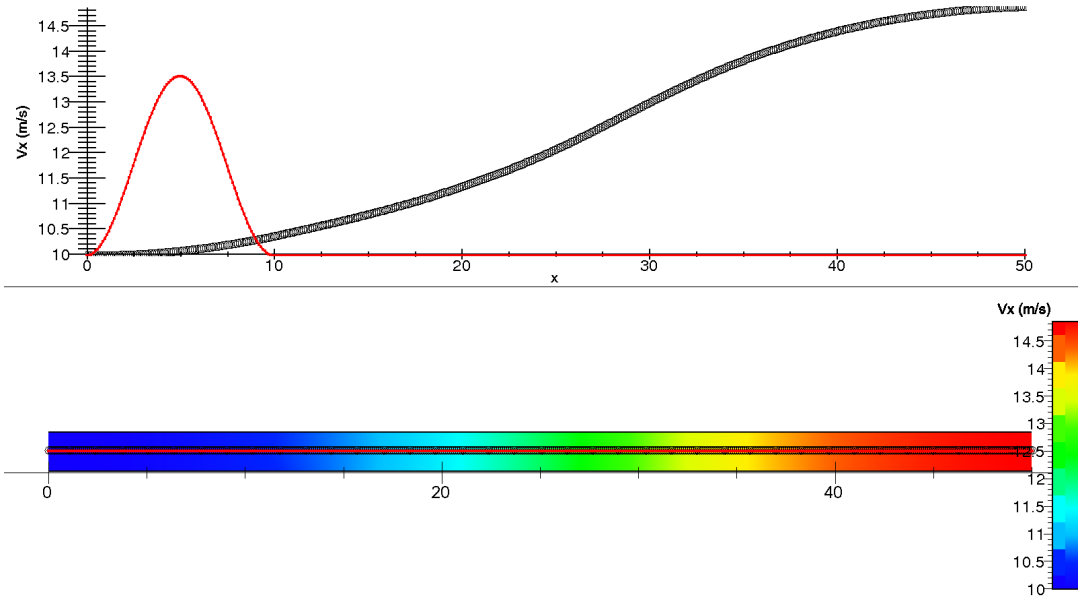


Figure 4.16: Case 1 at t=1s: V_x-component and velocity profile; o = CFD, □ = theoretical.

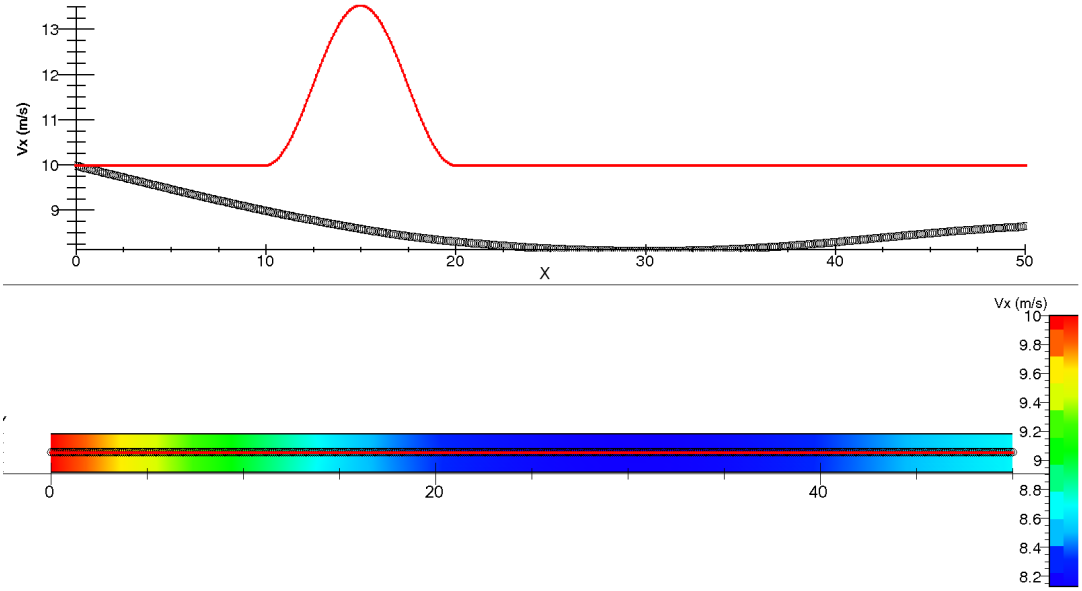


Figure 4.17: Case 1 at t=2s: V_x-component and velocity profile; o = CFD, □ = theoretical.

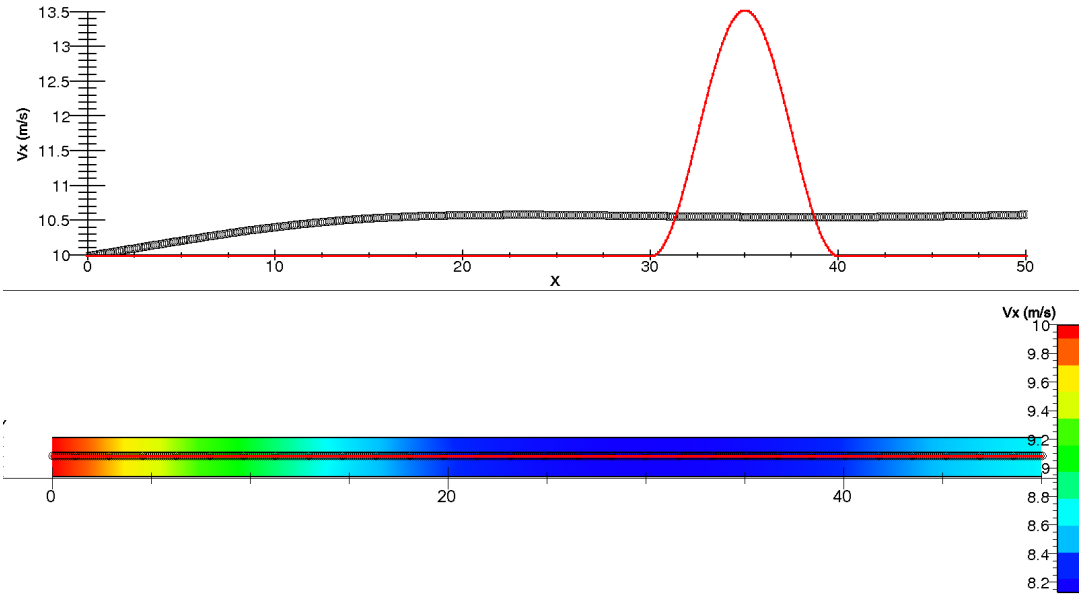


Figure 4.18: Case 1 at t=4s: V_x-component and velocity profile; o = CFD, □ = theoretical.

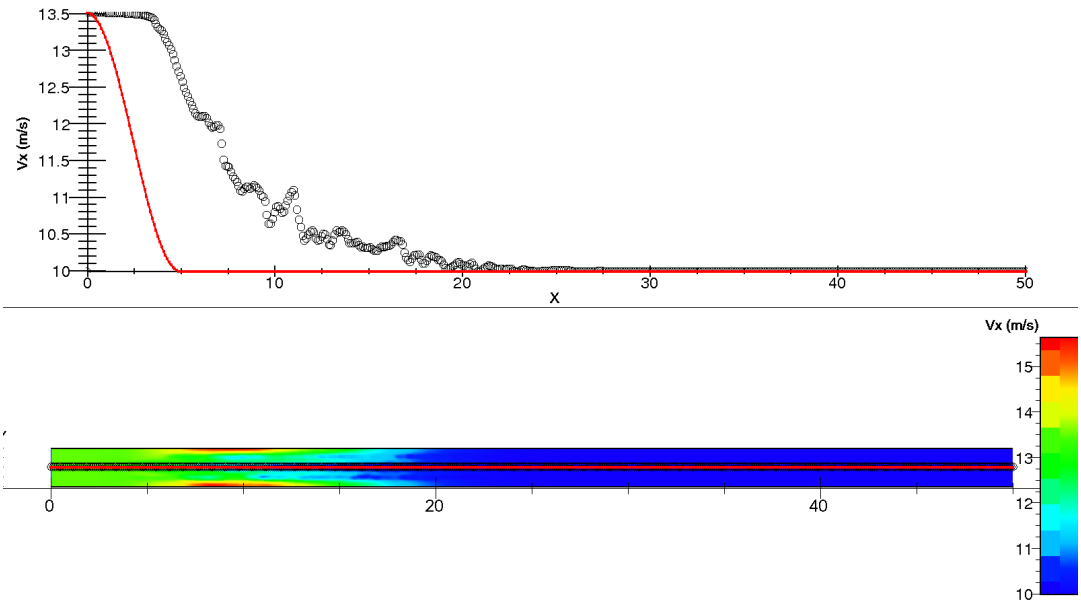


Figure 4.19: Case 2 at t=0.5s: V_x-component and velocity profile; o = CFD, □ = theoretical.

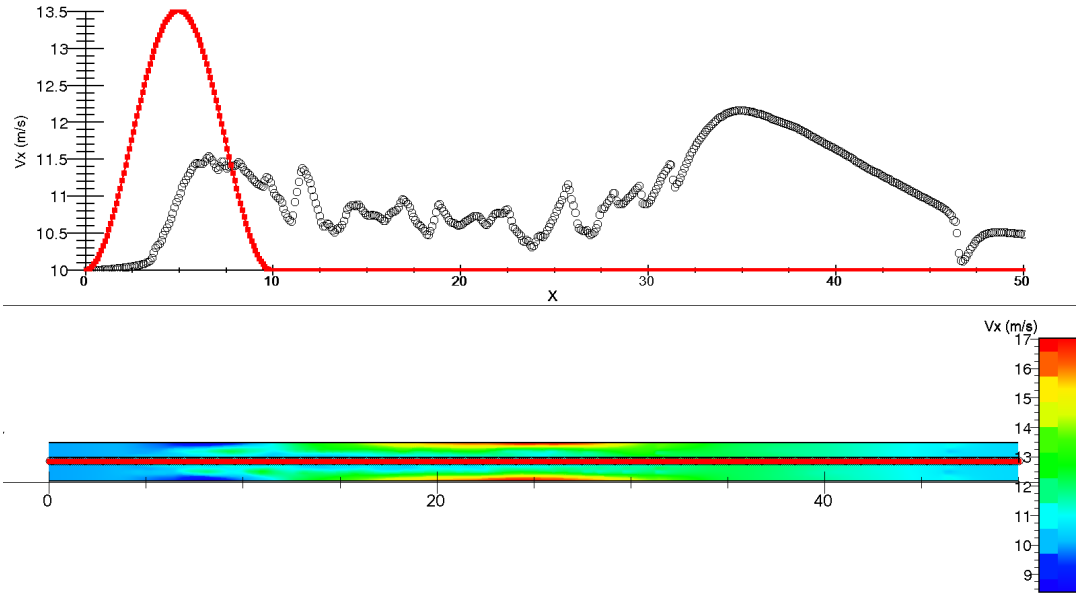


Figure 4.20: Case 2 at t=1s: V_x-component and velocity profile; o = CFD, □ = theoretical.

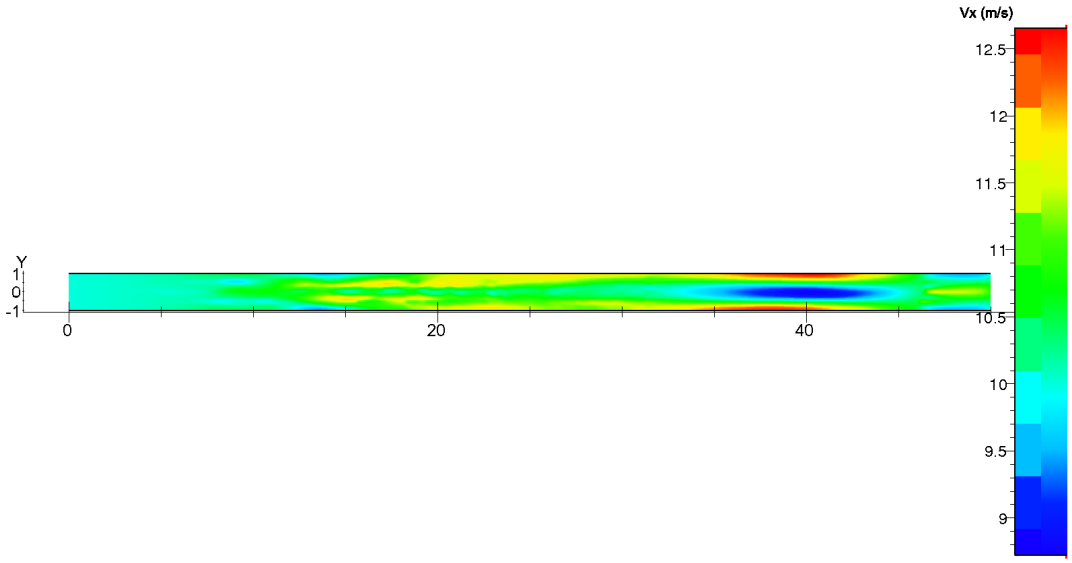


Figure 4.21: Case 2 at t=2s: V_x-component.

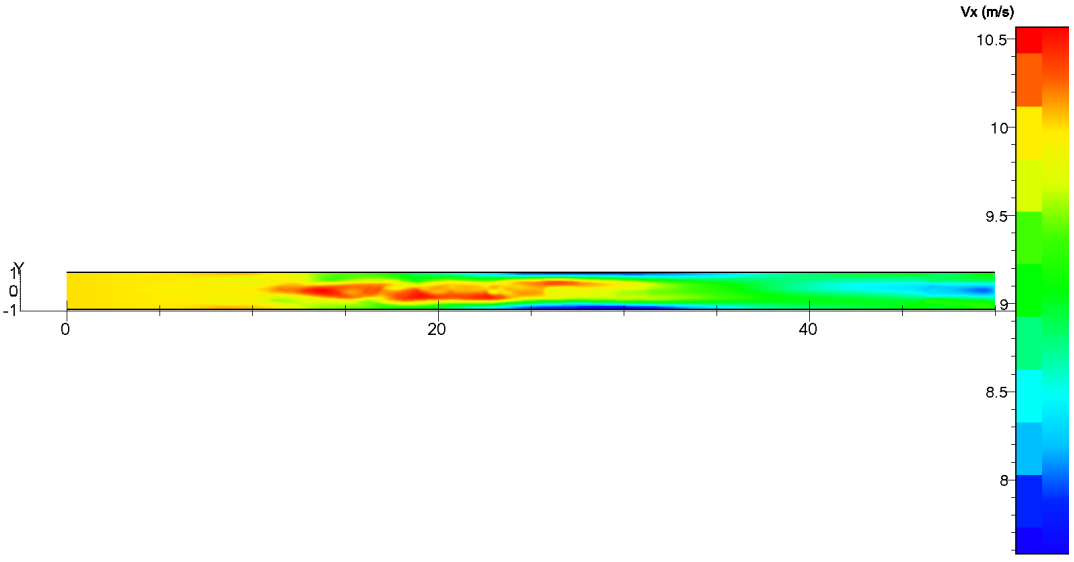


Figure 4.22: Case 2 at t=4s: V_x-component.

4.3.3 Unsteady inflow boundary condition with source term: CGSource

Model description and implementation

Now, `CGsource` will be extended since the results were not satisfactory. The problem that disturbances are not traveling at the convection speed of 10m/s - due to a violation of the conservation law - can be circumvented by adding a source term to the Navier Stokes term, hence the name `CGSource`.
The standard way to solve a flow is by estimating U and P such that the Navier Stokes equations with certain boundary- and initial conditions is satisfied. For the convenience of the reader, the Navier Stokes equations according to Euler are repeated here.

mass:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} = 0 \tag{4.5}$$

momentum in x-direction:

$$\frac{\partial \rho u}{\partial t} + \frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} + \frac{\partial p}{\partial x} = 0 \tag{4.6}$$

momentum in y-direction:

$$\frac{\partial \rho v}{\partial t} + \frac{\partial \rho uv}{\partial x} + \frac{\partial \rho v^2}{\partial y} + \frac{\partial p}{\partial y} = 0 \tag{4.7}$$

energy:

$$\frac{\partial \rho E}{\partial t} + \frac{\partial \rho Eu}{\partial x} + \frac{\partial \rho Ev}{\partial y} + \frac{\partial Ep}{\partial x} = 0 \tag{4.8}$$

where $E = e + \frac{V^2}{2}$.
Since there is already a function for the velocity, namely the CGM (equation 4.2), there is also already an exact solution for the Navier Stokes equations. By implementing this exact solution in the Navier Stokes equations, the equations will change. The extra term that will show up is the source term. Shortly said, the source term will ensure that solving the Navier Stokes equation will result in the velocity dictated by the CGM. The source term acts as a mass compensation for the violation of the conservation law in the previous approach.
Assuming

- $u = u_0 + f(x,t) = u_0 + \frac{W_g}{2}\left[1 - \cos\left(2\pi(t - \frac{x}{V_g})(\frac{V_g}{L_g})\right)\right]$
- $v = 0$

- $V = \sqrt{u^2 + v^2} = u$
- density (ρ) is constant

equations 4.5 to 4.8 can be modified as:

$$\rho \frac{\partial}{\partial x} f(x,t) = h_1(x,t) \tag{4.9}$$

$$\rho \frac{\partial}{\partial t} f(x,t) + \rho \frac{\partial}{\partial x} f^2(x,t) + \frac{\partial P}{\partial x} = h_2(x,t) \tag{4.10}$$

$$\frac{\rho}{2} \frac{\partial}{\partial t} f^2(x,t) + \rho e \frac{\partial}{\partial x} f(x,t) + \frac{\rho}{2} \frac{\partial}{\partial x} f^2(x,t) + \frac{\rho}{2} \frac{\partial}{\partial x} f^3(x,t) = h_3(x,t) \tag{4.11}$$

which are the expressions for the source terms h_1, h_2 and h_3. The further derivation of the source terms can be found in appendix C. Notice that 4.9 to 4.11 are the source terms for the mass, x-momentum and energy equation. The source term for the y-momentum equation is zero.
The source terms are added to the Numeca code in the void-function `DataStructureManager::computeTASourceTerm()` in `TimeAccurateDataManager.C`. See appendix A.3. For the condition *iMovingGrid_ == 0* the source terms 4.9, 4.10 and 4.11 are included and only activated if the data-file `projectname.cosGustSource` exists in the project directory. This file contains the parameters for the gust function. Now the following problem shows up. Since the gust is traveling, the source terms may only be introduced to cells that contain the gust on that specific moment in time. The undisturbed part of the domain should stay unaffected by the new source terms. Therefore the source terms h_1, h_2 and h_3 are implemented as a step-function.

$$\text{source term} = \begin{cases} 0 & \text{if } x < V_g t - L_g \\ h_{1,2,3}(x,t) & \text{if } V_g t - L_g < x < V_g t \\ 0 & \text{if } V_g t < x \end{cases} \tag{4.12}$$

Since the flow-problem is incompressible, only the mass and momentum equation will be used. There are only two flow parameters to solve (V and P), instead of three (V, P and ρ) for an incompressible flow-problem. The contribution of h_3 can be neglected. This was also tested and verified during the simulations.

CFD set-up

To make an optimal comparison between the new (`CGSource`) and the previous approach (`CGNoSource`) the same domain and mesh is used. Again two cases were tested. The Numeca flow solver settings summarized in table 4.3 are the same.

Setting	Case 1	Case 2
Grid resolution	200 x 8 cells, $\Delta x = \Delta y = 0.25$	500 x 20 cells, $\Delta x = \Delta y = 0.1$
Time-step Δt	0.01s	0.005s
Flow Model	Unsteady Euler with Low Speed Function (default values)	
Inlet	`projectname.cosGust`	
Initial Velocity	$V_x = 10$, $V_y = 0$	
Initial Pressure	101300Pa	
Initial Temperature	293K	
External Velocity	$V_x = 10$, $V_y = 0$	
External Pressure	101300Pa	
External Temperature	293K	
Multigrid	no	
Numerical scheme	Central 2^{nd} order scalar Jameson dissipation scheme	
# iterations per time-step	50	
L_g	10m	
W_g	3.5m/s	
V_g	10m/s	

Table 4.3: CFD set-up for the test cases - CGsource

Results and discussion

As was stated in the previous paragraph the gust is expected to cover only 10m when traveling through the domain. What still remains unsure is how the gust shape will look like within the 10m coverage. It is also not known yet how the transition between the parts of the domain were a source term is used and the parts where it is not used.

The first time-series of results for case 1 are found in figure 4.28 to 4.30. In contrast to the results from `CGNoSource` the gust travels at a correct convection speed of 10 m/s. Also a gust length of 10m is well preserved through the whole domain. Within the gust itself some additional disturbances are present which are not expected. These disturbances might be caused by the presence of extra source terms in the gust-part of the domain. Up- and downstream of the gust the domain is not influenced by the extra source terms. The solver tries to achieve an equilibrium between the different domain parts which results in a different velocity behavior.
For case 2 a finer grid is used. The results are different. The finer mesh seems to destroy the vertical homogeneity of the velocity which is harmful for the simulation of the horizontal cosine gust. No velocity profile is shown since unsteady structures seem to develop through the domain.

In general it can be concluded that the requirements are not fulfilled w.r.t simulating a correct, smooth horizontal gust in a CFD flow domain. Besides the fact that the cosine shape is not preserved also a second problem can be noticed. Splitting the domain into two types is not favorable to investigate the aerodynamic characteristics of an airfoil. The source term will influence the physical correctness of the NS equations; aerodynamic results would be useless. This could - partly - be solved by removing the source term when the gust is

approaching the airfoil. However, than again it is unsure whether the airfoil is exposed to a correct gust. Another approach is proposed in the next section.

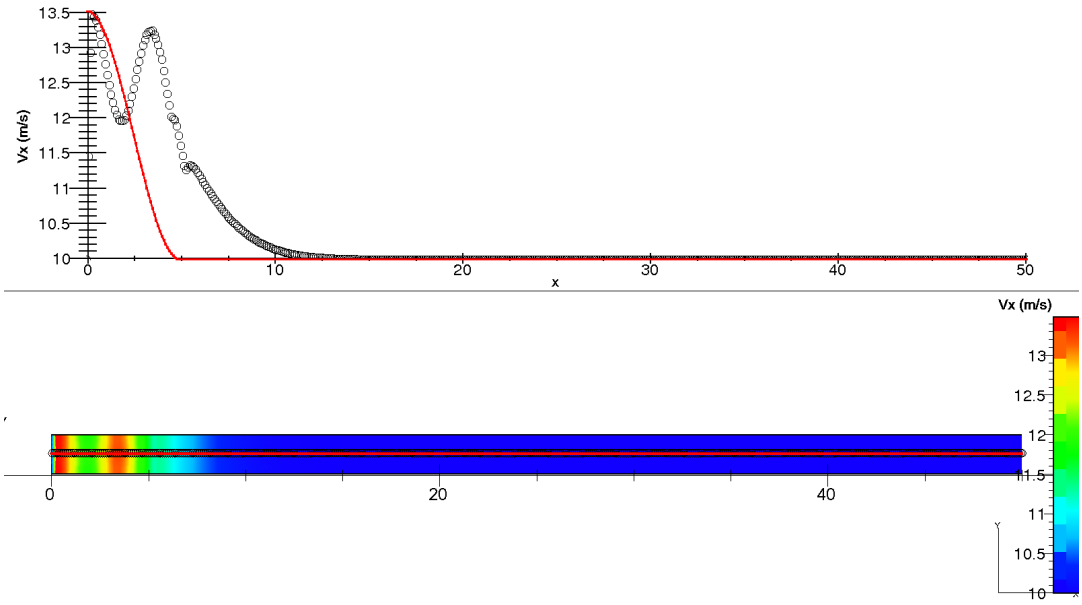


Figure 4.23: Case 1 at t=0.5s: V_x-component and velocity profile; o = CFD, □ = theoretical.

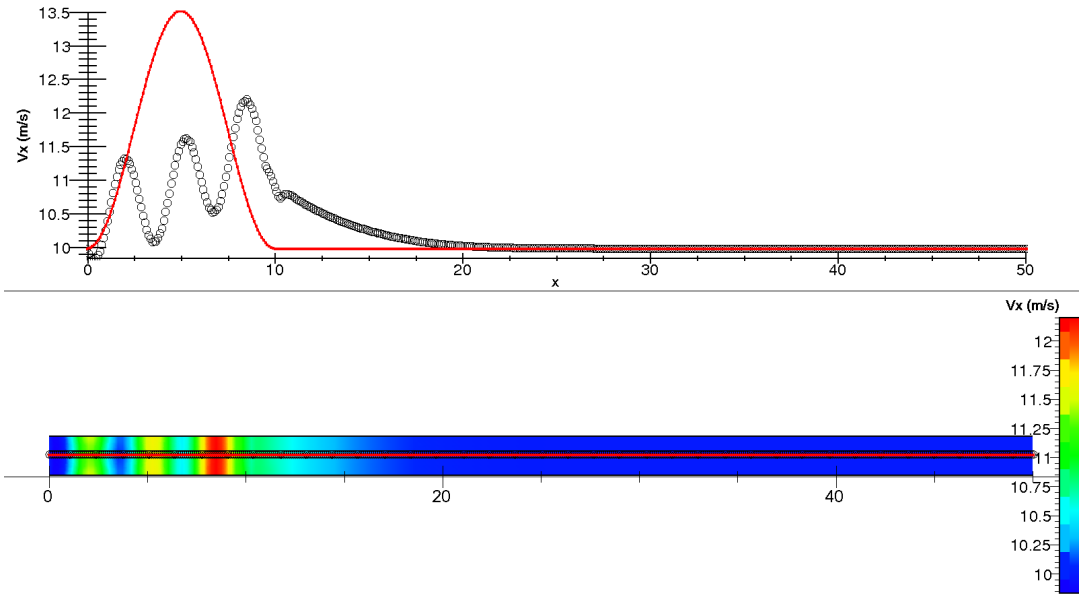


Figure 4.24: Case 1 at t=1s: V_x-component and velocity profile; o = CFD, □ = theoretical.

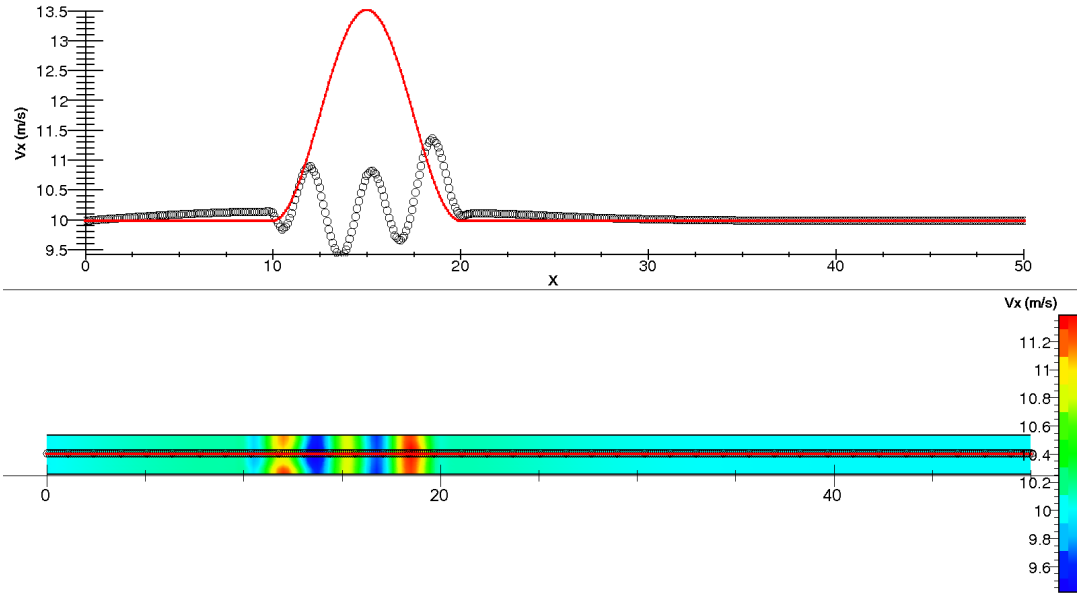


Figure 4.25: Case 1 at t=2.0s: V_x-component and velocity profile; o = CFD, □ = theoretical.

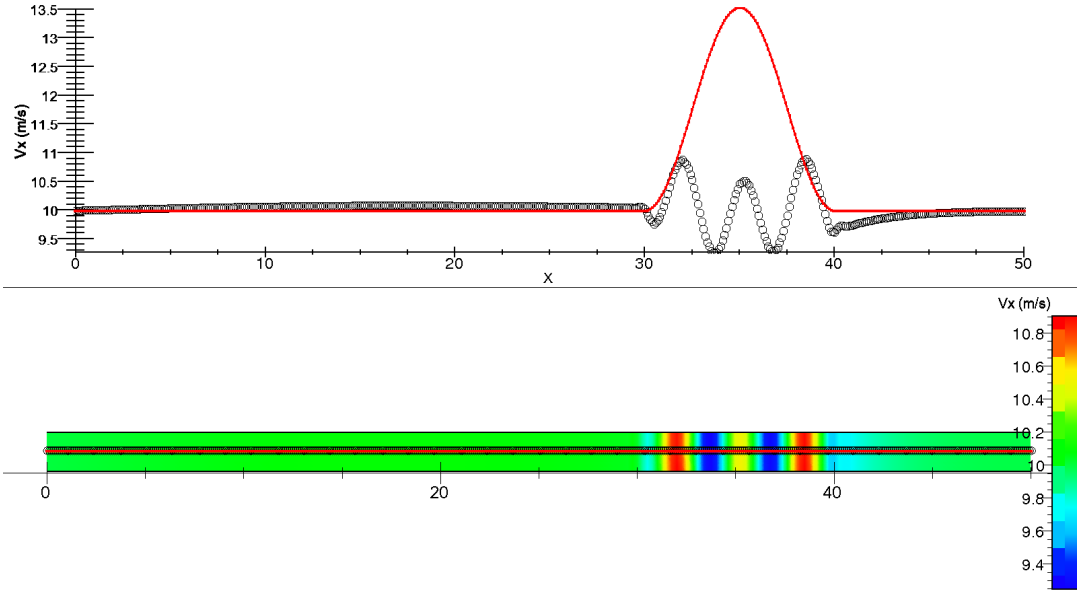


Figure 4.26: Case 1 at t=4.0s: V_x-component and velocity profile; o = CFD, □ = theoretical.

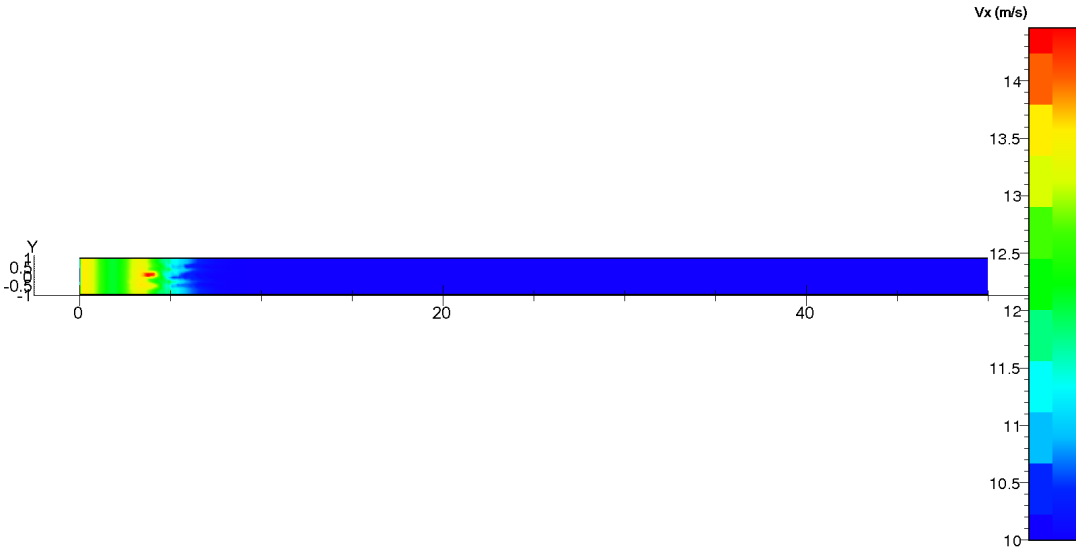


Figure 4.27: Case 2 at t=0.5s: V_x-component and velocity profile; o = CFD, □ = theoretical.

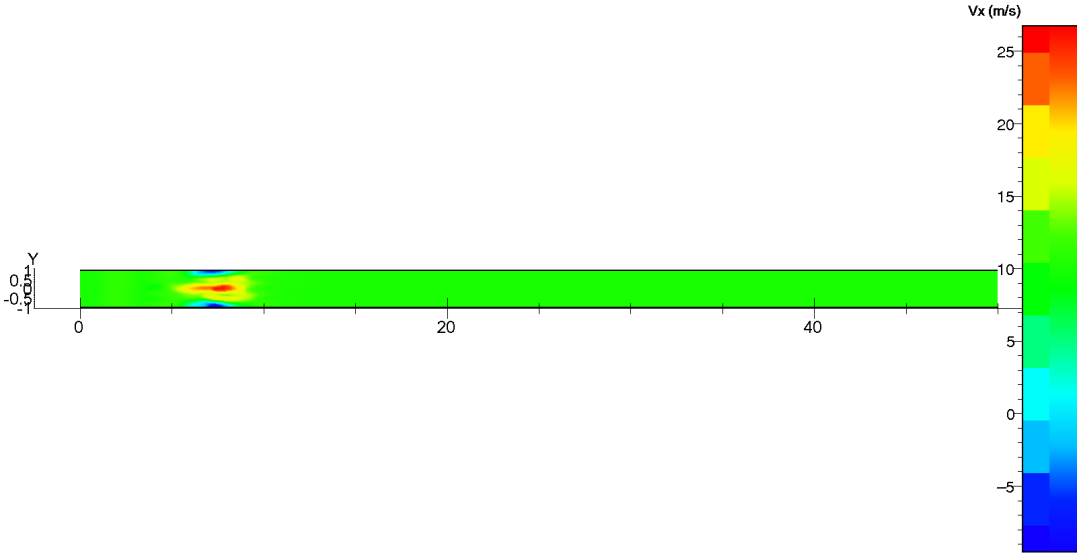


Figure 4.28: Case 2 at t=1.0s: V_x-component and velocity profile; o = CFD, □ = theoretical.

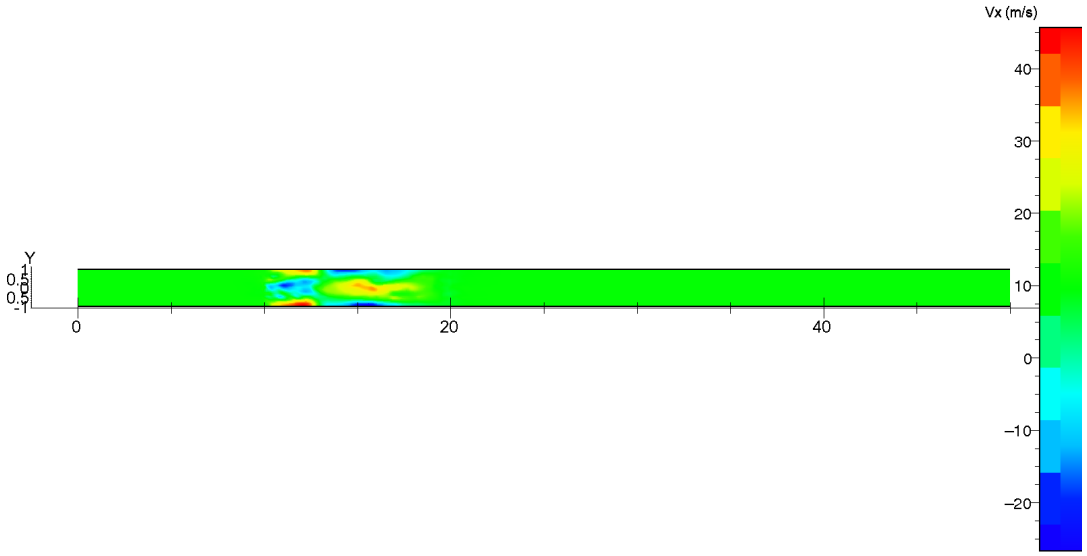


Figure 4.29: Case 2 at t=0.5s: V_x-component and velocity profile; o = CFD, □ = theoretical.

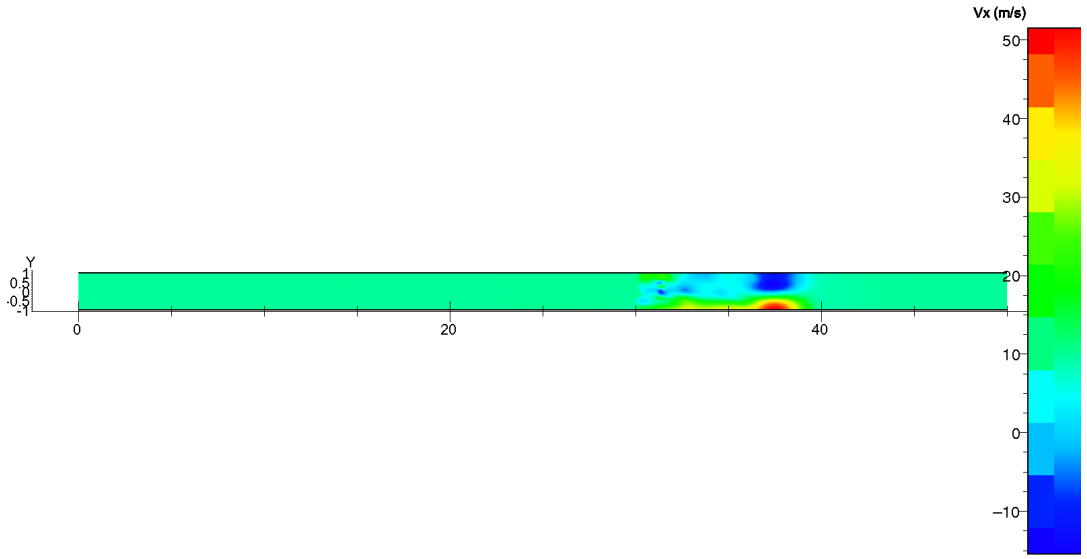


Figure 4.30: Case 2 at t=4.0s: V_x-component and velocity profile; o = CFD, □ = theoretical.

4.3.4 Cosine Gust Duct: CGDuct

Model description and implementation

The third method uses wind tunnel principles to introduce the gust, i.e. the domain features a convergent-divergent duct (throat). Since mass is conserved the velocity will increase gradually to a maximum when approaching the throat and again decrease gradually when passing the throat. Instead of deforming and moving the walls, a vertical inflow is imposed imitating the solid walls. Attention, the throat is now located behind the wall inflow, not at the location of maximum inflow velocity. Since an extra mass flow is then introduced at the wall inflow also an outflow is imposed behind the throat to keep mass conserved. The outflow replaces the function of the source term in the previous approach. Let us summarize; the inflow section will increase the horizontal velocity while the outflow section will decrease the horizontal velocity. The streamlines will follow the gradient of the wall inflow. In figure 4.31 a sketch is presented of the domain set-up.
This approach makes it also easy to transport the gust over the flow by giving it a velocity V_g. Of course the vertical inflow will interact and mix up with the horizontal flow near the boundaries. It is expected that these irregularities are not present in the middle between the two walls. The symmetry of the domain will create a horizontal flow in the middle.

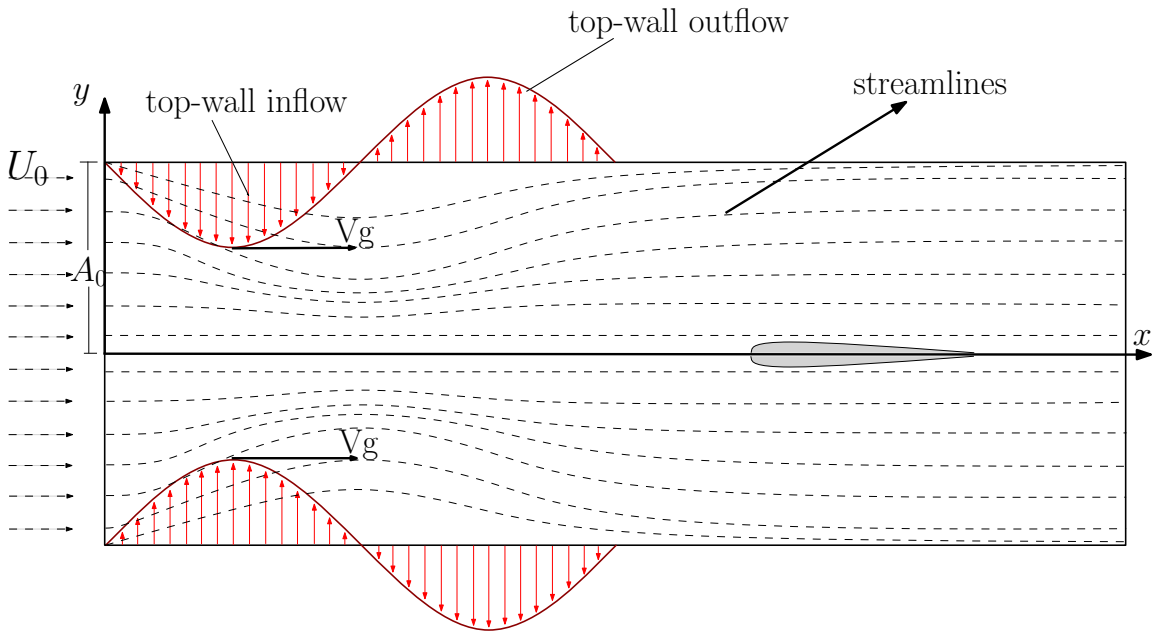


Figure 4.31: Sketch of Cosine Gust Duct

The derivation of the correct wall in- and outflow function starts from the conservation of

mass for an incompressible 2D problem:

$$\begin{aligned} \vec{\nabla}\vec{u} &= 0 \\ &\Leftrightarrow \\ \frac{du}{dx} + \frac{dv}{dy} &= 0 \\ &\Leftrightarrow \\ \frac{dv}{dy} &= -\frac{du(x,t)}{dx} \end{aligned} \tag{4.13}$$

and the velocity function given by equation 4.2:

$$u(x,t) = u_0 + \frac{Wg}{2}\left[1 - \cos\left(2\pi(t - \frac{x}{V_g})(\frac{V_g}{L_g})\right)\right] \tag{4.14}$$

After substituting equation 4.14 into equation 4.13 and integrating over the height y above the center line of the domain, a relation can be found between $v(x,t)$ at height y and $u(x,t)$.

$$\int_0^y \frac{dv}{dy}dy = -\int_0^y \frac{du(x,t)}{dx}dy \tag{4.15}$$

$$= -\frac{du(x,t)}{dx}y \tag{4.16}$$

$$v(x,y,t) = -\frac{W_g\pi}{L_g}\sin\left(2\pi(t - \frac{x}{V_g})(\frac{V_g}{L_g})\right)y \tag{4.17}$$

The wall inflow velocity at y=A_0 is then given by the following function:

$$v(x,t) = -\frac{W_g\pi}{L_g}A_0\sin\left(2\pi(t - \frac{x}{V_g})(\frac{V_g}{L_g})\right) \tag{4.18}$$

In figure 4.13 the difference between $u(x,t)$ (eq 4.2) and its derivative $u_x(x,t)$ (eq 4.17) is visualized.

To introduce a vertical inflow at the walls equation 4.17 needs to be implemented in the void-function `Slipwall::numUpdate()` of `VelocityWall.C`. For the condition *iMovingGrid_ == 0* an if-statement is added that checks the existence of the file `projectname.ductGust`.

This file contains the parameters for the function. The periodicity of the function is neutralized by implementing it as a step-function.

$$\_\text{wallVelocity} = \begin{cases} 0 & \text{if } x < V_g t \\ -\frac{W_g \pi}{L_g} A_0 \sin\left(2\pi(t - \frac{x}{V_g})(\frac{V_g}{L_g})\right) & \text{if } V_g t < x < V_g t + L_g \\ 0 & \text{if } V_g t + L_g < x < 0 \end{cases} \tag{4.19}$$

Now, only one wave of velocity is traveling along the boundaries.

CFD set-up

Six cases were set up to investigate the use of a duct to generate a cosine gust. An overview of the case names and their variables is summarized in table 4.4. For the test cases two different meshes are used. They are presented in figure 4.32 and 4.33. The flow solver settings are summarized in table 4.5 and are slightly different for the coarse and fine mesh.

Case name	L_g	W_g	V_g	Domain
Case 10-15-2	10	15	2	Coarse
Case 10-25-2	10	25	2	Coarse
Case 6-15-2	6	15	2	Coarse
Case 4-15-2	4	15	2	Coarse
Case 10-15-2f	10	15	2	Fine
Case 5-15-2f	5	15	2	Fine

Table 4.4: Overview test cases - CGDuct

Results and discussion

From figure 4.34 to 4.39 the results of all test cases are present. Figure 4.34 presents a snapshot after every 4 seconds to show clearly the gust development in time. During the first 8 seconds the adaptation time can be observed which is needed for the flow to adapt to the wall inflow. After 8s the flow downstream of the gust stays around 10.5m/s which is slightly higher than the mean flow of 10m/s. After 28s the gust shape is slightly disturbed by a velocity increment downstream of the gust but is restored afterwards. The created gust reaches an amplitude of maximum 13m/s (cfr. 13.5m/s theoretically). After the adaptation time the gust length stays around 20m. This is twice as much as the expected theoretical gust length of 10m.
An increment of the wall inflow velocity results in a higher gust amplitude, a longer gust length and a longer adaptation time. This is done in case 10-25-2 of which the results are presented in figure 4.35.
In case 6-15-2 the parameter L_g of the wall inflow function is decreased from 10 to 6. The results are presented in figure 4.36 after every 8s. In general similar behavior can be noticed as case 10-15-2. After a certain adaptation time the gust length stays around 20m. However,

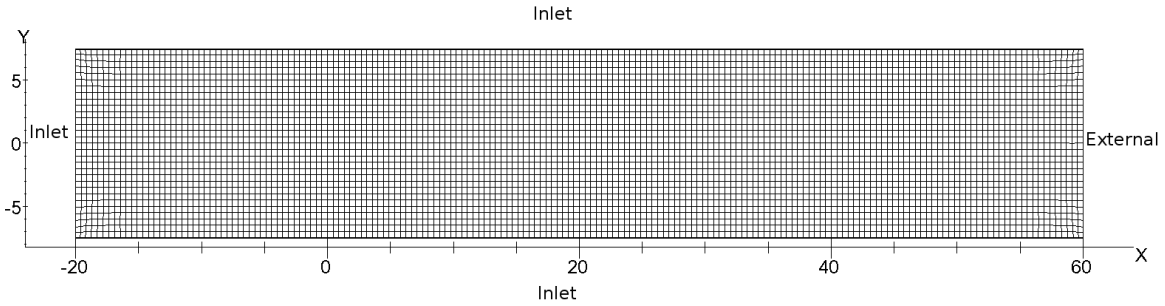


Figure 4.32: Coarse mesh; 180 x 30 cells, $\Delta x = 0.45, \Delta y = 0.5$

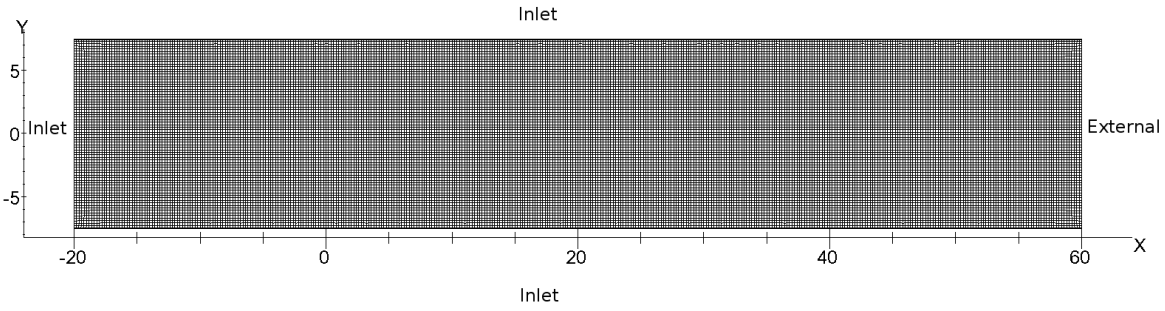


Figure 4.33: Fine mesh; 400 x 75 cells, $\Delta x = 0.2, \Delta y = 0.2$

Setting	Fine mesh	Coarse mesh
Grid resolution	180 x 30 cells	400 x 75 cells
	$\Delta x = 0.45$ $\Delta y = 0.5$	$\Delta x = 0.2$ $\Delta y = 0.2$
Time-step Δt	0.04s	0.0016s
Flow Model	Unsteady Euler with Low Speed Function (default values)	
Inlet	`projectname.ductGust`	
Initial Velocity	$V_x = 10$, $V_y = 0$	
Initial Pressure	101300Pa	
Initial Temperature	293K	
External Velocity	$V_x = 10$, $V_y = 0$	
External Pressure	101300Pa	
External Temperature	293K	
Multigrid	no	
Numerical scheme	Central 2^{nd} order scalar Jameson dissipation scheme	
# iterations per time-step	50	

Table 4.5: CFD set-up for the test cases - CGDuct

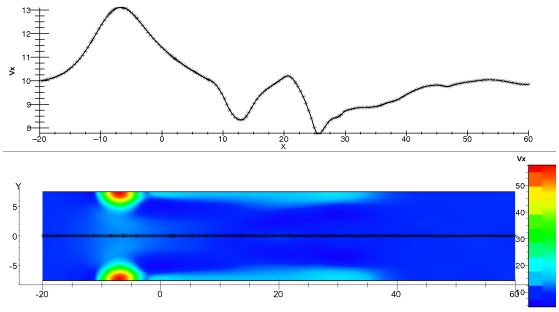
the gust amplitude has increased from 22m/s after 8s to 26m/s after 32s. It seems that a shorter wall-inlet does not decrease the gust length but increases the gust amplitude.
In figure 4.37 the results of even a shorter wall-inlet ($L_g = 4$) are presented. The velocity behavior is completely different. After the location of the duct the velocity drops to negative values. Near the wall inflow velocities above 100m/s are reached. The mesh is too coarse to deal which such high velocities. Therefore the results might not be correct. Case 10-15-2f uses the same inlet conditions as case 10-15-2 but on a finer mesh. The results are presented in figure 4.38. The gust generated near the duct has a length of 20m and an amplitude around 13.5m/s which was also observed in case 10-15-2. However, downstream of the gust a large velocity disturbance takes place. A finer mesh is beneficial for the development of the gust near the throat, but also amplifies additional disturbances. The development of additional disturbances when using a finer mesh can also be observed for case 5-15-2f in figure 4.39.

At first sight the use of a wall inflow to generate a cosine gust seems not perfect to create a cosine shaped gust. However, the approach shows some benefits:

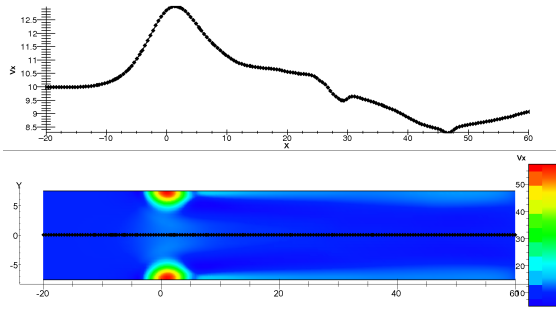
- The use of a wall outflow to restore the violation of mass conservation seems to work better than the use of source terms.
- The gust can travel at every desired velocity.
- The inflow velocity function can be replaced by other functions. More complex gust shapes can be easily implemented.
- The CFD set-up is relatively simple.
- No fine mesh is needed to simulate a gust. Grid refinement can be limited to the area around the airfoil.

Further investigations are advised by making an important improvement to the CFD set-up. This will be explained below.

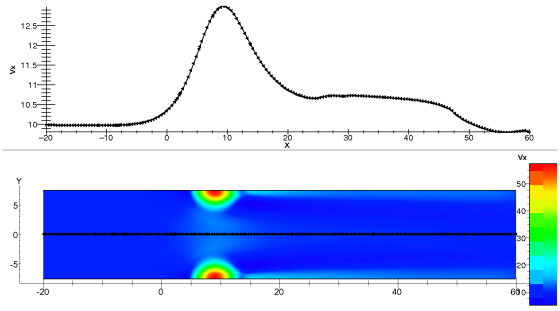
The CFD set-up for this investigation is equal to the sketch in figure 4.31. The simulation started at t=0 with a uniform flow (u_0) at the inlet and a wall inflow in the domain. During the simulation (t>0) the inflow travels downstream with velocity V_g along the wall adapting the uniform flow which results in a cosine gust. It is also possible to start the wall inflow entering the domain after the simulation is started. Instead of using a horizontal uniform inflow u_0 at the inlet the velocity relation (4.17) can be used. This will avoid the adaptation time that showed up in the results. Many side effects are then expected to disappear. Unfortunately, due to time limitations this improvement could not be made.
A second improvement could be the use of the z-direction. Instead of imposing the wall in- and outflow in the y-direction, this can also be imposed in the z-direction. The gust created by this inflow is expected to be the same. However, now the side effects of these wall flows are transfered to the third dimension which is not relevant for a 2D problem.



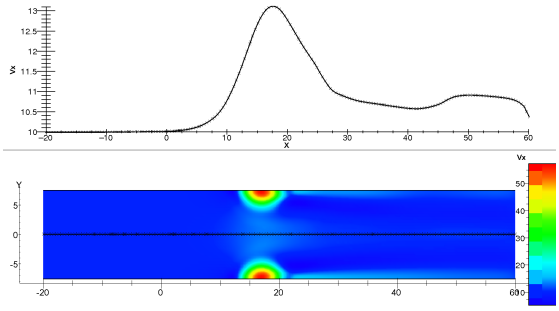
(a) 4s



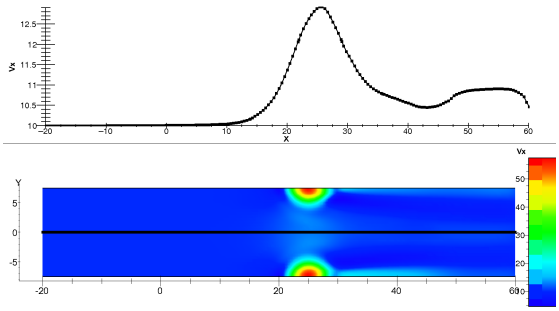
(b) 8s



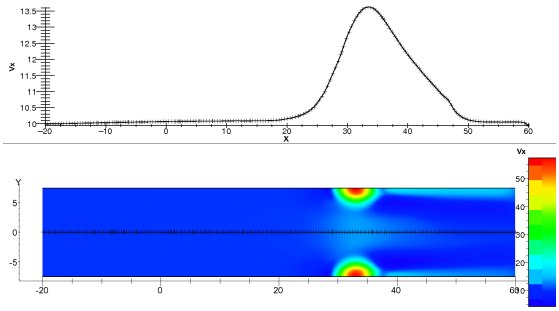
(c) 12s



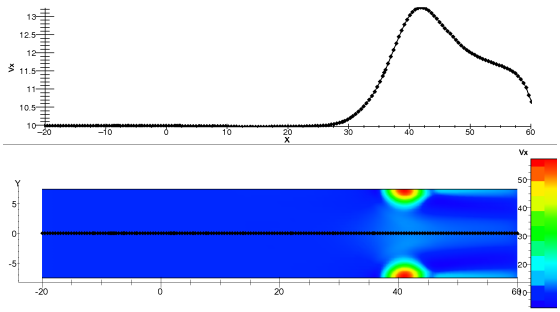
(d) 16s



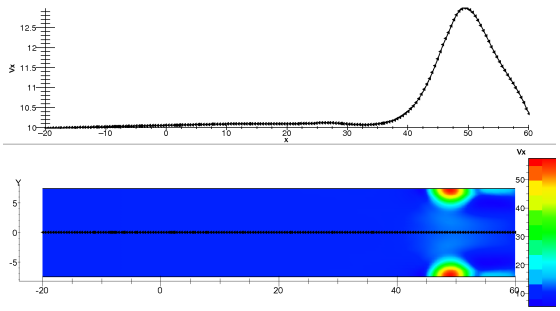
(e) 20s



(f) 24s

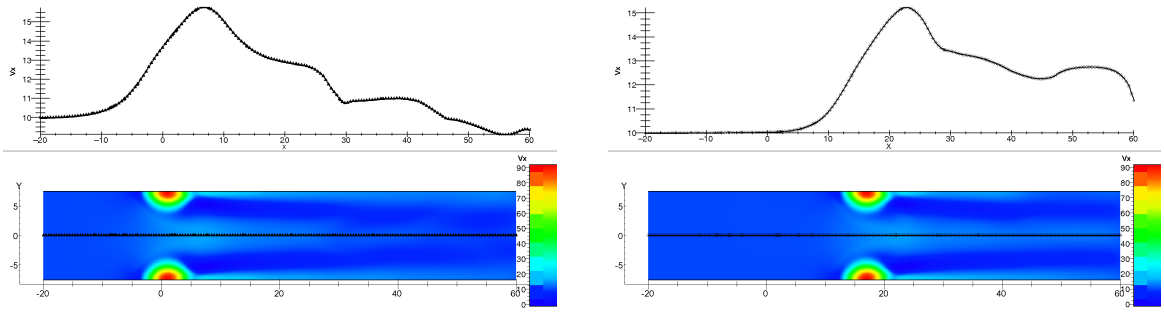


(g) 28s



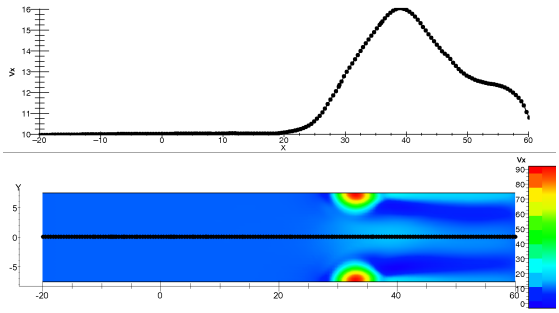
(h) 32s

Figure 4.34: Case 10-15-2: V_x-component and velocity profile

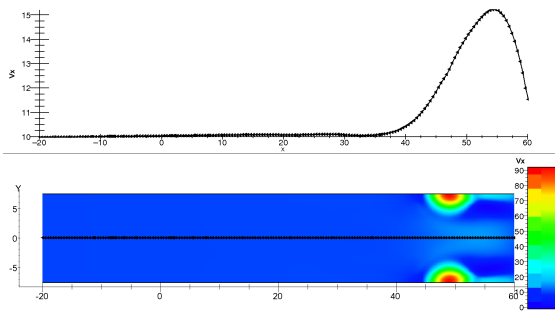


(a) 8s

(b) 16s

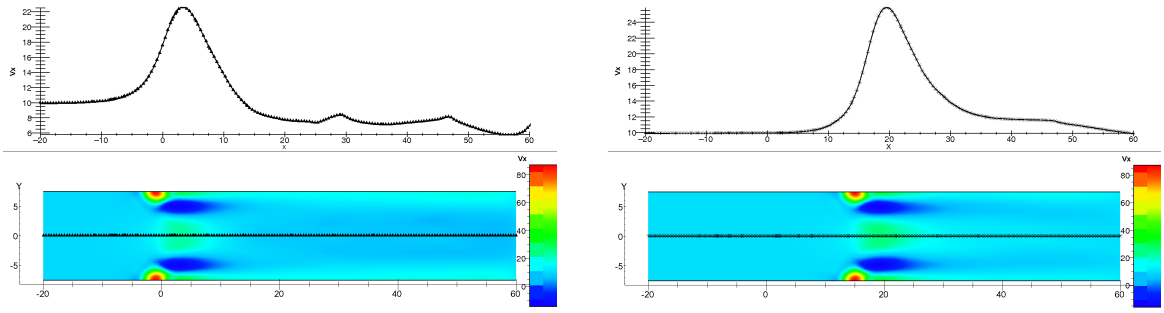


(c) 24s



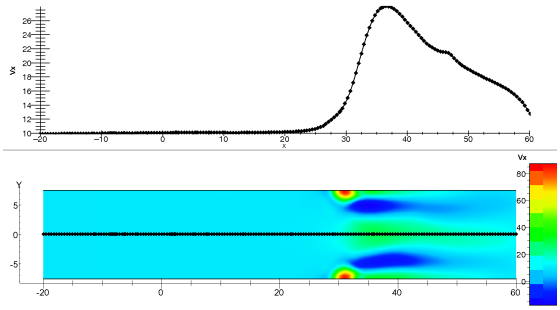
(d) 32s

Figure 4.35: Case 10-25-2: V_x-component and velocity profile

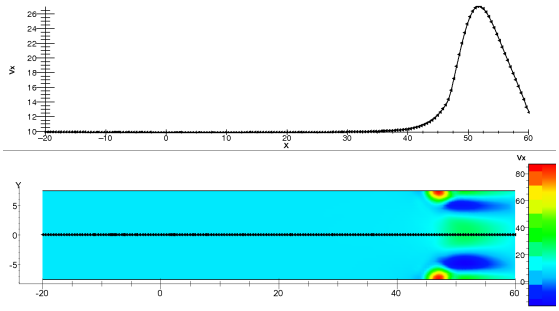


(a) 8s

(b) 16s

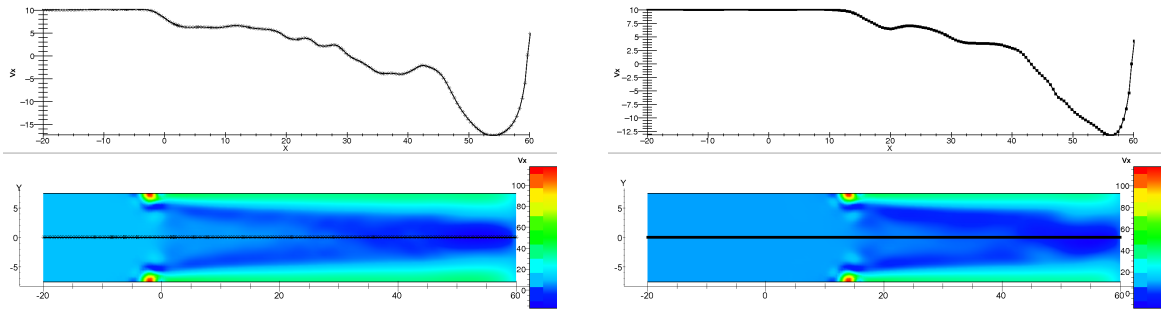


(c) 24s



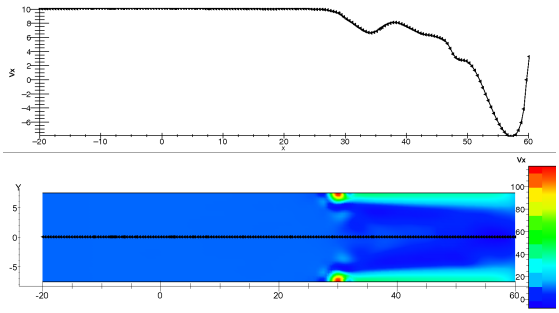
(d) 32s

Figure 4.36: Case 6-15-2: V_x-component and velocity profile

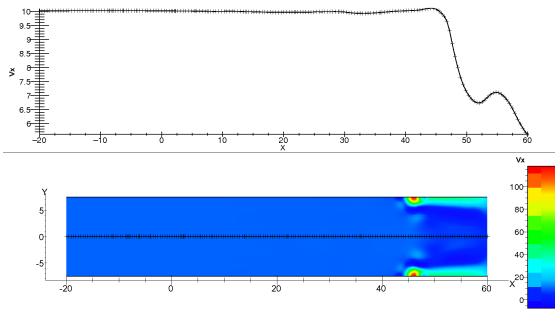


(a) 8s

(b) 16s

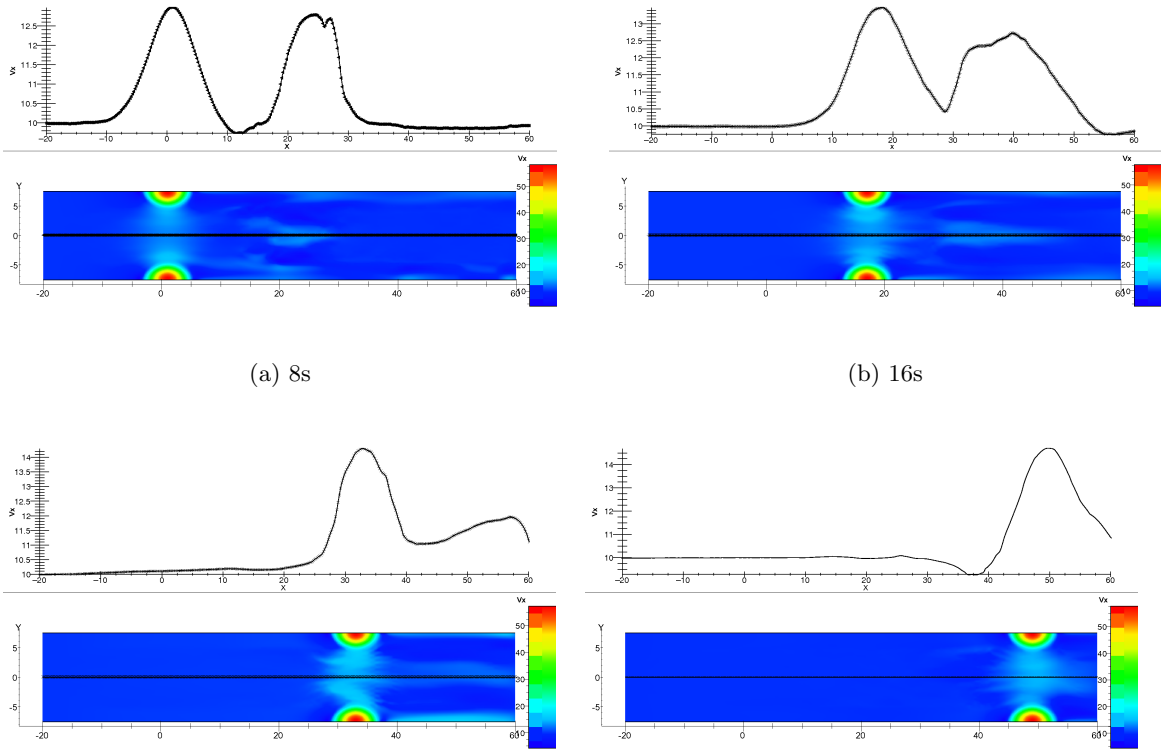


(c) 24s



(d) 32s

Figure 4.37: Case 4-15-2: V_x-component and velocity profile



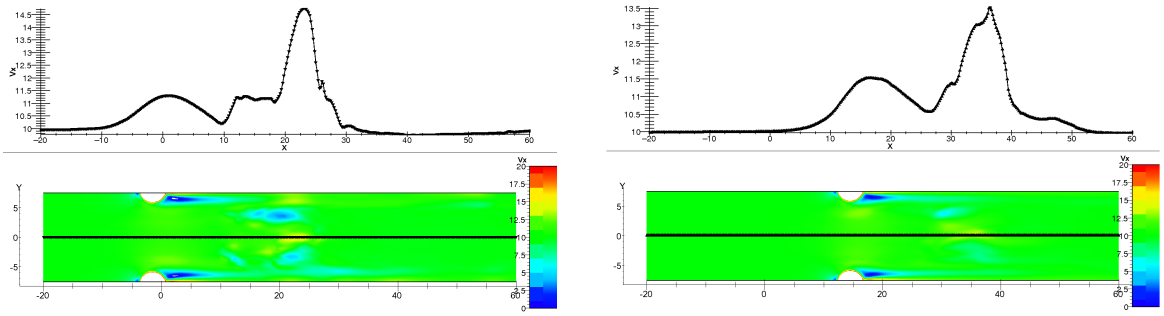
(a) 8s

(b) 16s

(c) 24s

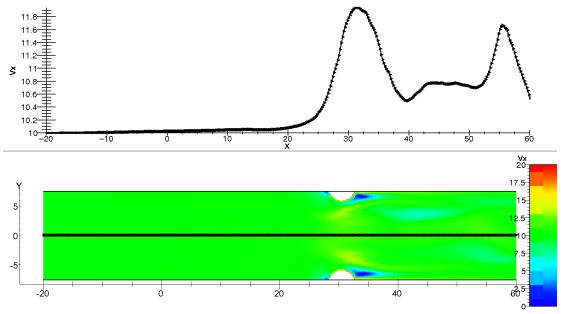
(d) 32s

Figure 4.38: Case 10-15-2f: V_x-component and velocity profile

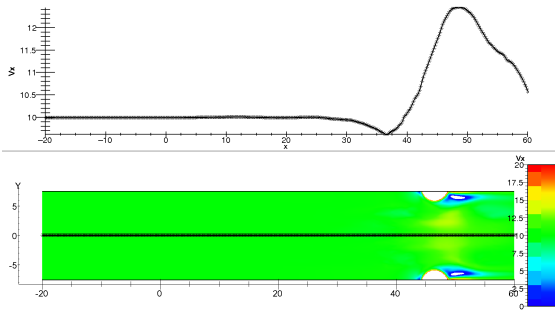


(a) 8s

(b) 16s



(c) 24s



(d) 32s

Figure 4.39: Case 5-15-2f: V_x-component and velocity profile

4.4 Vortex based Gust Model: VGM

4.4.1 Model description and implementation

The third gust model discussed and tested in this report is the Vortex Gust Model (VGM). It is based on the idea of generating a cosine-shaped gust with a 2D-vortex. In [15] an analytical solution of the Euler equations for a vortex is given. For a uniform flow, scaled to

$$(u_\infty, v_\infty, p_\infty, T_\infty) = (1m/s, 0m/s, 1Pa, 1K) \tag{4.20}$$

the perturbations are given by

$$u' = -\frac{\lambda_s}{2\pi}(y - Y)exp(\eta(1 - r^2)) \tag{4.21}$$

$$v' = \frac{\lambda_s}{2\pi}(x - X)exp(\eta(1 - r^2)) \tag{4.22}$$

$$T' = -\frac{(\gamma - 1)\lambda_s^2}{16\eta\gamma\pi^2}exp(2\eta(1 - r^2)) \tag{4.23}$$

where $r = \sqrt{(x - X)^2 + (y - Y)^2}$ is the distance to the vortex center (X, Y), λ is the vortex strength, η is a parameter determing the gradient of the solution which is stated to 1 and $\gamma = c_p/c_v = 1.4$. The vortex is assumed to be isentropic which states $p = \rho^\gamma$. Using the equation of state and scaling the specific gas constant R to 1J/kgK, results in $T = p/\rho$. The density can then be found as

$$\rho = (T_\infty + T')^{\frac{1}{\gamma - 1}} \tag{4.24}$$

By implementing the analytical solutions for the perturbations into the Numeca flow solver, a vortex can be solved along a uniform flow at the correct convection speed. In [15] S. Rhebergen already tested the implementation of the analytical equations 4.21 to 4.23 for a single vortex in the Numeca flow solver.
The scaled-equations 4.21 to 4.23 describe the perturbations for each position in space caused by a single vortex. The vortex is defined by a vortex center (X, Y), a vortex-size (η) and a vortex-strength (λ). Since a vortex satisfies the conservation of mass $\vec{\nabla} \cdot \vec{u} = 0$ also the summation of vortices satisfies the conservation of mass $\vec{\nabla} \cdot \sum_i \vec{u_i} = 0$ when perturbations are small.

For a summation of vortices traveling along the uniform flow u_0 equation 4.21 can be used to find a function for the flow velocity. This results in

$$u(\overrightarrow{x},t) = u_0 + u_0 \sum_{i}^{Nv} u_i'(\overrightarrow{x},t) \tag{4.25}$$

where Nv is the total number of vortices,

$$u_i'(\overrightarrow{x},t) = -\frac{\lambda_i}{2\pi}(y - Y_i)exp(\eta_i(1 - r_i^2)) \tag{4.26}$$

and

$$r_i(\overrightarrow{x},t) = \sqrt{(x - (X_i + u_0 t))^2 + (y - Y_i)^2} \tag{4.27}$$

The same can be found for $v(\overrightarrow{x},t)$ and $T(\overrightarrow{x},t)$.

The idea of using the characteristics of multiple vortices to construct a horizontal cosine-shaped gust resulted in the Vortex Based Gust model used in this investigation. However, up till now the behavior and the interaction between multiple vortices is unsure. Below is explained how a velocity field is generated by multiple gusts.

4.4.2 Wind field generation

The VGM uses a superposition of multiple vortices to generate a velocity field that corresponds to a horizontal cosine-shaped gust. Therefore the velocity profile of the velocity field should be equal to equation 4.2 and uniform along the vertical direction. In figure 4.40 a schematic representation is given of how a velocity field is constructed of multiple vortices.
To ensure a horizontal flow in the middle of the domain (along y=0) all vortices should be mirrored over the symmetry line y=0. As a result the vertical velocity components will compensate each other and a horizontal flow will remain. Further away from the symmetry line the vertical velocity disturbances will increase.
To create a velocity field that is uniform along the vertical direction a series of vortices are placed above each other. The strength of these vortices should increase linearly with the distance to the symmetry line. Superposition of the increased vorticity strengths will result in a uniform horizontal flow velocity (red vectors). Now, a sort of uniform gust column is created.
To increase the column width of the gust (L_g) the vortex size (η) can be increased. Multiple columns of vortices next to each other will also increase the column width. This is represented by the gray vortex columns in figure 4.40. By finding a suitable combination between vortex strength, vortex size and vortex columns a velocity profile equal to the cosine-gust can be found. Figure 4.41 shows the difference between a single vortex and a combination of multiple vortices. The black rectangle locates the upper part of the domain used to simulate the gust.
So summarized:

- The vortex strength should increase linearly with the distance to the centerline.
- An increased vortex strength will increase the gust length.

- Multiple columns of vortices will increase the gust length and gust shape.
- Maximum gust strength depends linearly on the vortex strength.

To facilitate finding a suitable combination of vortices Dr. ir. A.H. van Zuijlen developed a matlab-tool (`windfieldGenerator.m`). In appendix B the matlab code is included. The matlab tool uses equations 4.21, 4.22 and 4.23 to determine a resultant velocity- and temperature field according to a defined vortex configuration. The vortex information (X_i, Y_i, η_i and λ_i) is stored in a data-file (`projectname.windData`) which is used as inlet condition for a CFD computation using Numeca finehexa.

For this investigation a cosine gust is created with a length (L_g) of 10m and an amplitude of (W_g) of 3.5m/s. The results of the matlab-tool are shown in figure 4.42, 4.43 and 4.44. The velocity profile of the velocity field corresponds quit well to the theoretical profile. Notice the zero vertical velocity near the symmetry plane y=0 due to symmetrical vortices. The behavior of the gust - traveling through a CFD domain - is analyzed for three different mesh resolutions. To make an optimal comparison, the velocity profiles of the traveling gusts are compared to the theoretical profile and to each other. Results are discussed in subsection 4.4.4. Finally, also a cosine gust with a length of 1m is tested in a scaled domain. The results of the matlab-tool are also presented for this case in figure 4.45 to 4.47.

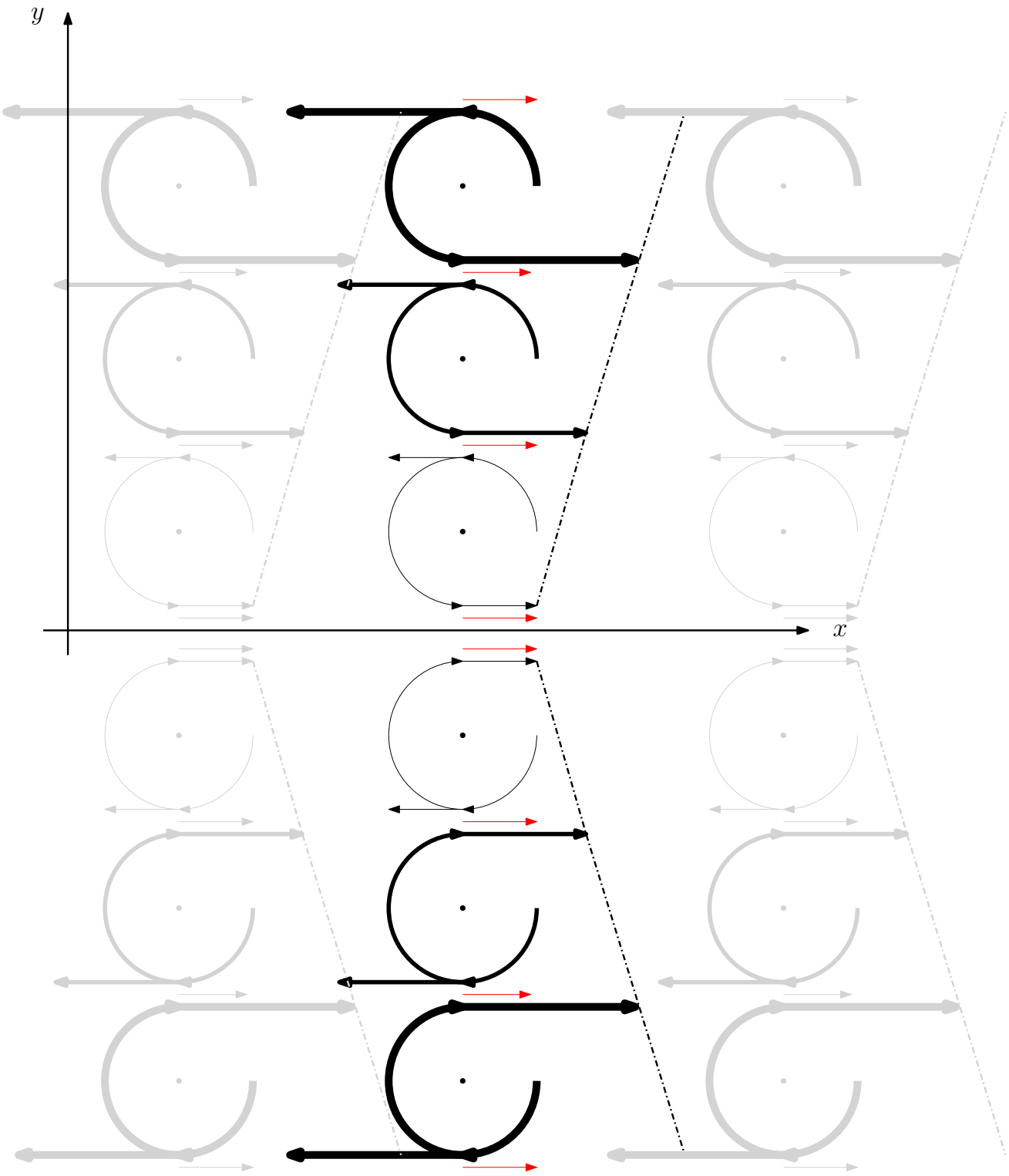


Figure 4.40: Schematic representation of vortices

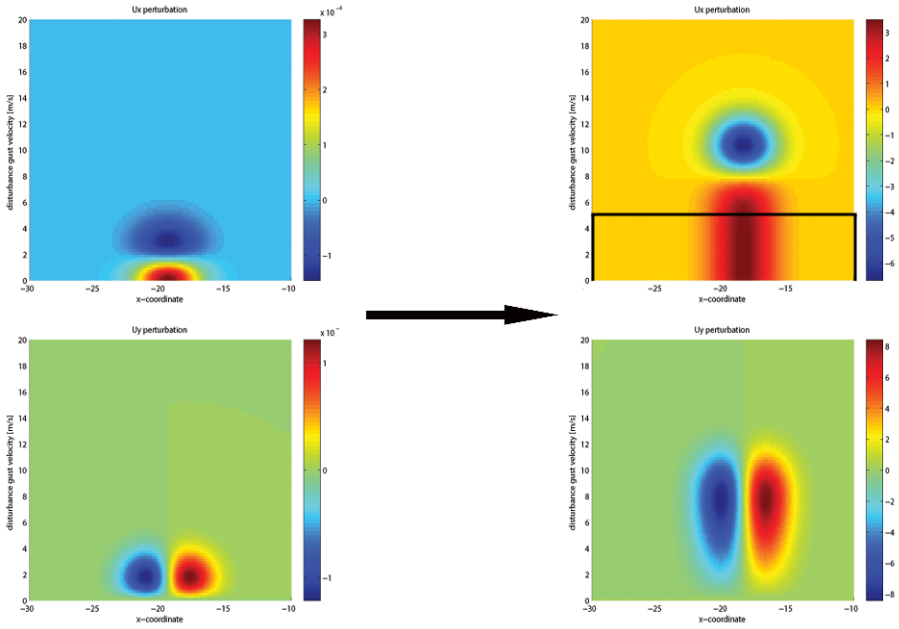


Figure 4.41: left: single vortex; right: combination of vortices with increased strength

4.4.3 CFD set-up

For this investigation four different simulations are made; three simulations for case 1 and one simulation for case 2. To analyze the influence of the mesh grid case 1 is simulated on a coarse-, fine and finest mesh. The length of the domain should be long enough to investigate the behavior of a 10 meters long gust. A domain height of 15m and a domain length of 20m is chosen. Figure 4.48 visualizes the coarse mesh. Besides the front side, also the upper- and lower wall are set as an inlet boundary condition. In this way, also the two walls are fed by vortices from the vorticity-field. The different mesh resolutions can be found in table 4.6. For case 2, a gust length of 1m is used. To make an optimal comparison with case 1 the CFD domain and the resolution need to be rescaled. For case 1 the behavior over a distance of twice the gust length is investigated. Case 2 should remain this set-up. The complete CFD set-up is summarized in table 4.6.

Setting	Case 1			Case 2
	Coarse	Fine	Finest	
L_g	10m	10m	10m	1m
W_g	3.5m/s	3.5m/s	3.5m/s	3.5m/s
Grid resolution	60 x 45 cells $\Delta x = \Delta y = 0.33$	120 x 90 cells $\Delta x = \Delta y = 0.16$	300 x 225 cells $\Delta x = \Delta y = 0.066$	60 x 45 cells $\Delta x = \Delta y = 0.33$
Time-step Δt	0.015s	0.007s	0.003s	0.015s
Flow Model	Unsteady Euler with Low Speed Function (default values)			
Inlet	`projectname.windData`			
Initial Velocity	$V_x = 10$, $V_y = 0$			
Initial Pressure	101300Pa			
Initial Temperature	293K			
External Velocity	$V_x = 10$, $V_y = 0$			
External Pressure	101300Pa			
External Temperature	293K			
Multigrid	4, with coarse grid initialization			
Numerical scheme	Central 2^{nd} order with scalar Jameson dissipation			
# iterations per time-step	50			

Table 4.6: CFD set-up for test cases - VGM

4.4.4 Results and discussion

In figure 4.49 to 4.52 the V_x-component of the three simulations for case 1 are presented after different time steps. Figure 4.50 shows the V_y-component for the coarse mesh. In figure 4.53 the velocity profile at the center line y=0 is presented for all simulations at the corresponding time steps. From figure 4.54 to 4.55 the same plots can be observed for case 2.

From the V_x color plots the behavior of the gust can be observed very clearly. The coarse mesh shows best results. The gust seems to lose its straight vertical column when

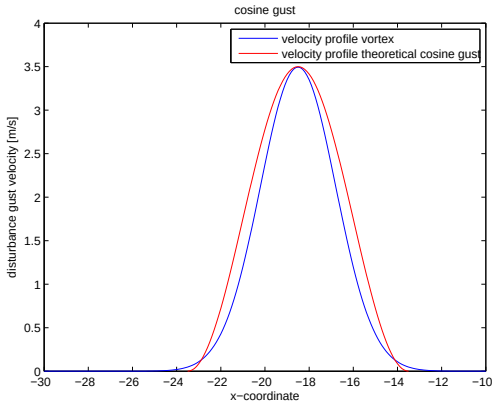


Figure 4.42: Case 1: Comparison vortex velocity vs. theoretical cosine gust velocity.

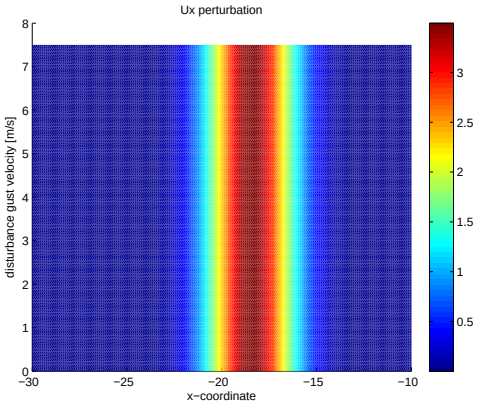


Figure 4.43: Case 1: Vortex velocity field - U_x.

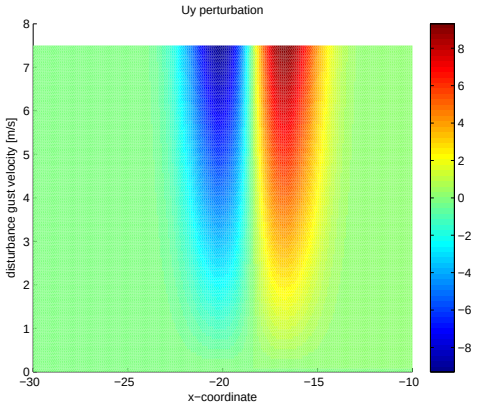


Figure 4.44: Case 1: Vortex velocity field - U_y.

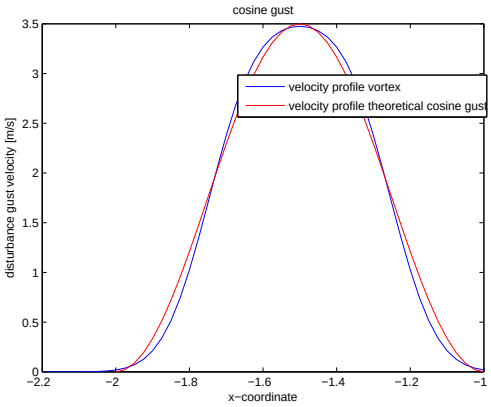


Figure 4.45: Case 2: Comparison vortex velocity vs. theoretical cosine gust velocity.

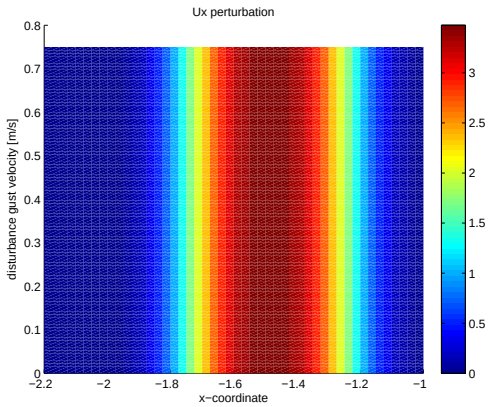


Figure 4.46: Case 2: Vortex velocity field - U_x.

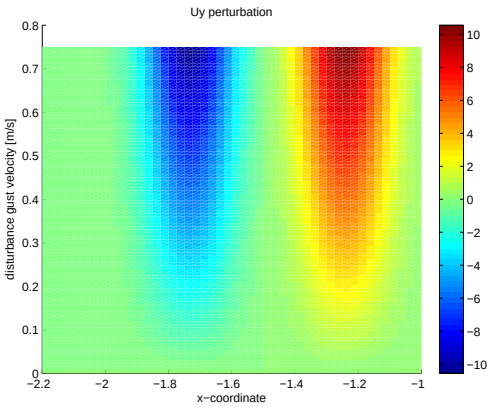


Figure 4.47: Case 2: Vortex velocity field - U_y.

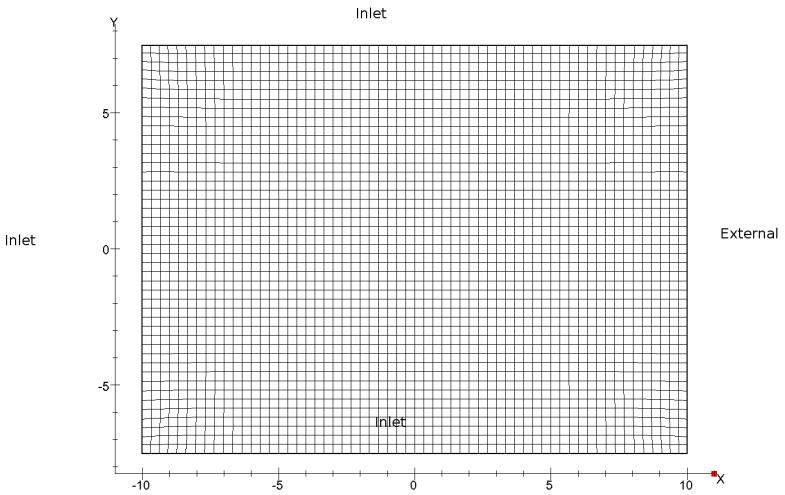


Figure 4.48: Coarse mesh for VGM computation: 60 x 45 cells, $\Delta x = \Delta y = 0.3m$

propagating through the domain. A small acceleration of the middle part w.r.t. the outer parts can be noticed. The white spots represent velocity out of the color range.
When a finer mesh is used results are less satisfactory. The fine mesh seems to amplify small disturbances instead of damping them out. The gust seems to get stretched out when traveling through the domain. The vertical homogeneity is destroyed which is harmful for a simulation of a horizontal gust.

The V_y-plot clearly shows the vertical components of the vortices of which the velocity-field is constructed. This is typical for a vortex and can not be avoided. The only measure to constrain this characteristic is stretching the vortices in the vertical direction such that these disturbances move to the exteriors. During the simulation these disturbances remain near the upper- and lower walls but are more likely to move inwards when a finer mesh is used. It would be harmful if they resolve to the interior of the domain since only a pure horizontal gust is desired.

The velocity-profiles show more accurate results. In the beginning of the simulation all curves match the best. The CFD velocity-profile of all cases seems to speed up slightly compared to the theoretical profile. Here a difference between the mesh resolutions can be observed. The amplitude of the coarse mesh increases slightly to 14.5m/s (cfr. 13.5m/s theoretically) while the amplitude of the finer meshes seem to decrease. The general cosine shape of the gust remains the best for the coarse grid. However, the gust is stretched out slightly to the right. The finer meshes show worse results on this.

The results of a 1m gust are presented in figure 4.54 to 4.55. The small acceleration of the gust which was observed in case 1 is also present in case 2. Moreover, the acceleration is even higher. The amplitude increases to almost 16.5m/s and the gust length is elongated. Upstream of the gust a dip in the flow velocity can be noticed which develops more during traveling.

The term quality is used to describe in general the characteristics of a simulated gust. The higher the quality, the more it matches the theoretical gust. So summarized:

- The cosine gust loses its quality when traveling through the domain.
- A too fine mesh can result in a lower quality since additional disturbances are amplified.
- A finer resolution destroys the vertical homogeneity of the gust column.
- A smaller gust is more likely to lose its quality.

Finally, the following conclusion can be drawn w.r.t investigating gust effects on an airfoil section.

- The airfoil should be positioned in the middle and close to the inlet in order to use the best part of the VGM.

- The upper- and lower wall should be far away from the center line to avoid vertical disturbances.

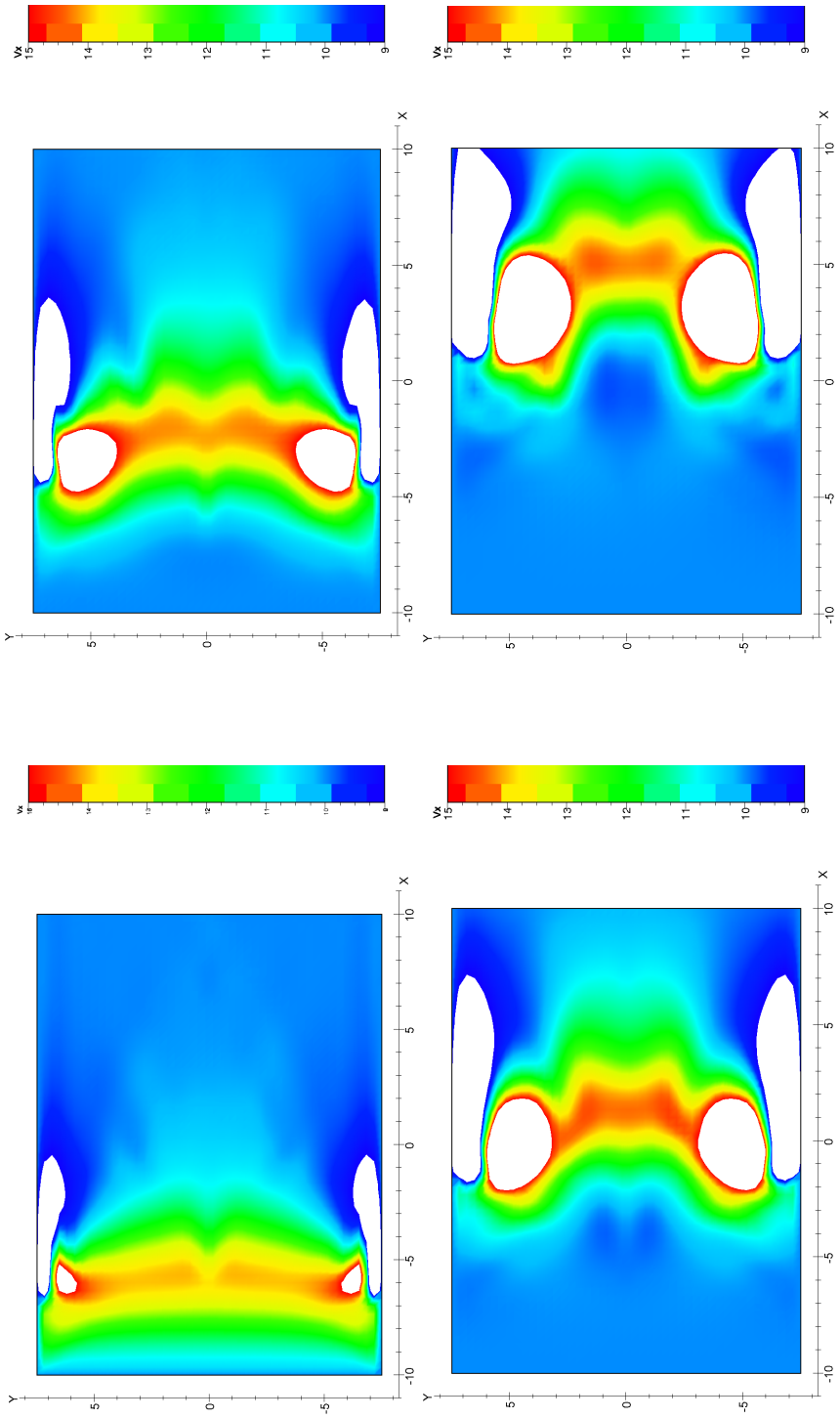


Figure 4.49: Case 1; V_x-component after respectively 1.3s-1.5s-1.8s-2.1s

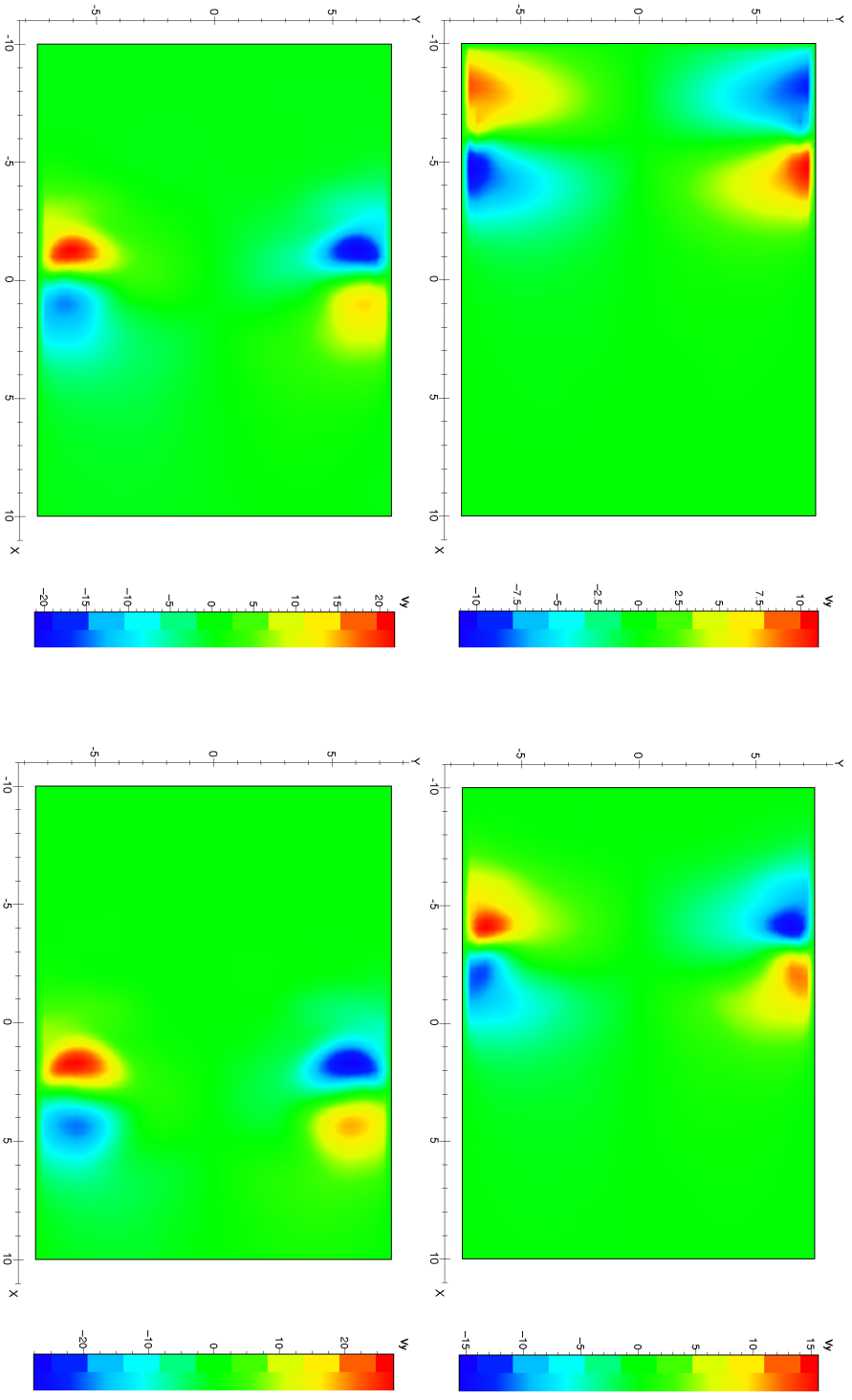


Figure 4.50: Case 1; V_y-component after respectively 1.3s-1.5s-1.8s-2.1s

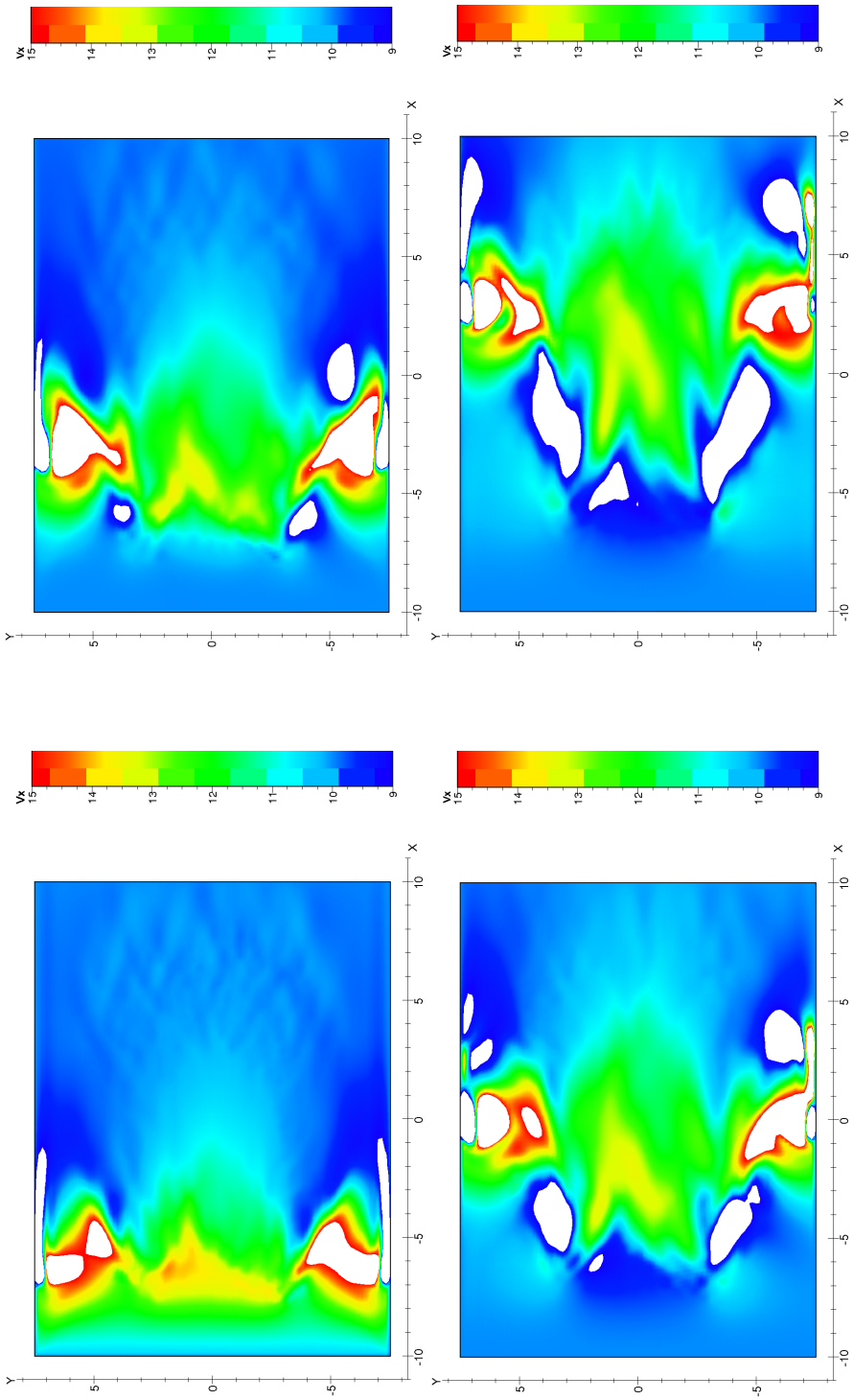


Figure 4.51: Case 2; V_x-component after respectively 1.3s-1.5s-1.8s-2.1s

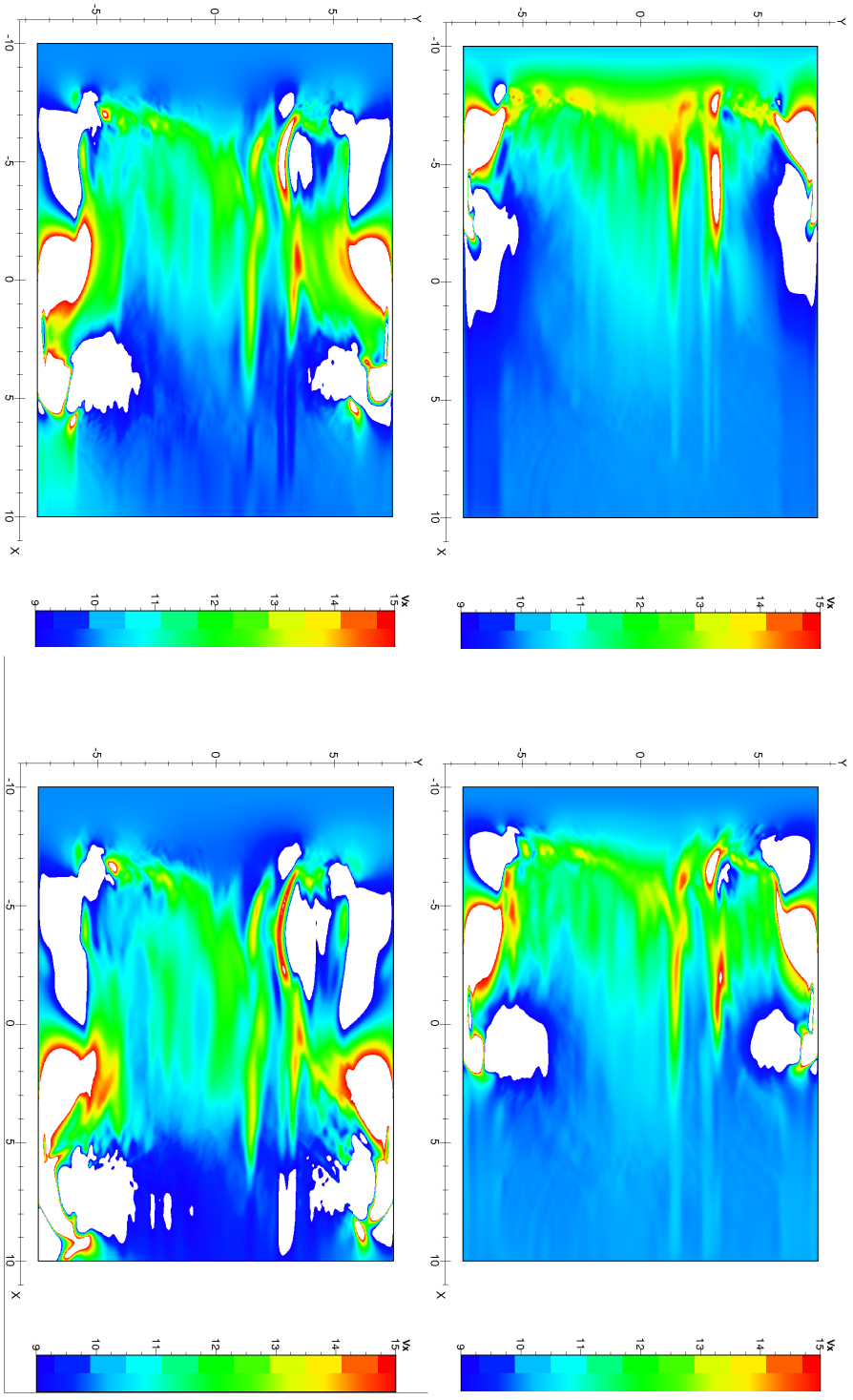


Figure 4.52: Case 3; V_x-component after respectively 1.3s-1.5s-1.8s-2.1s

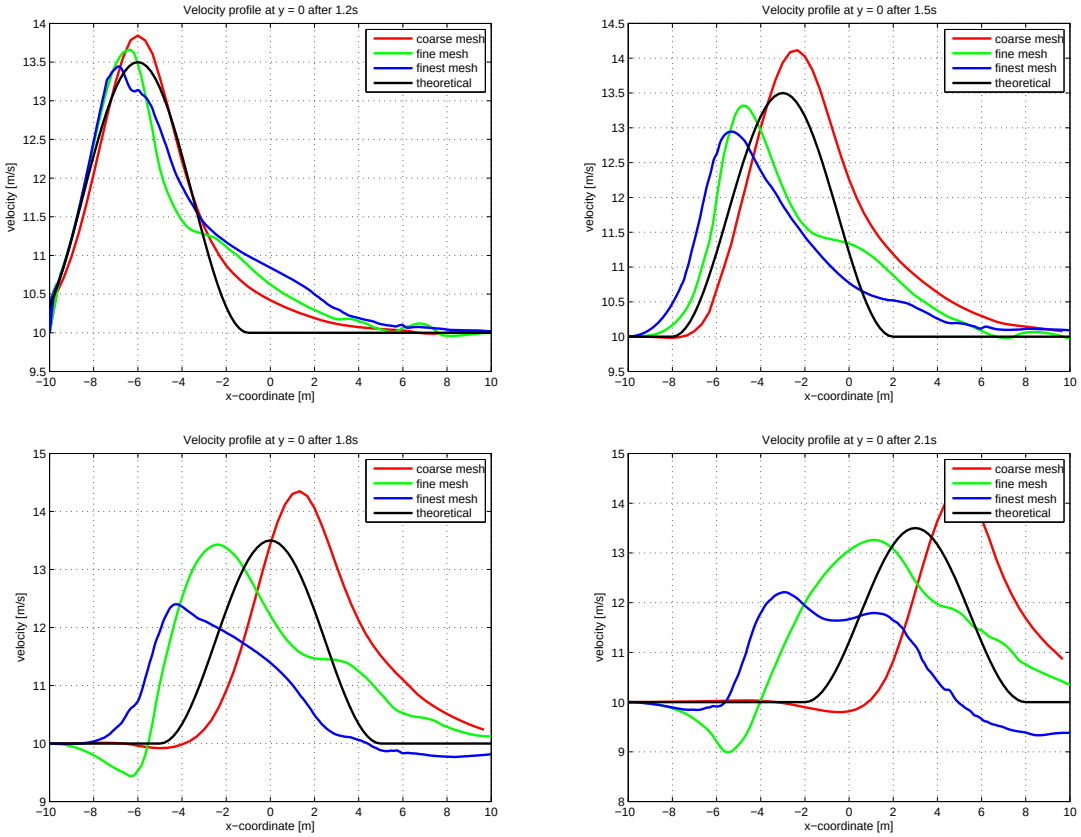


Figure 4.53: Case 1: Velocity profile after respectively 1.2s-1.5s-1.8s-2.1s

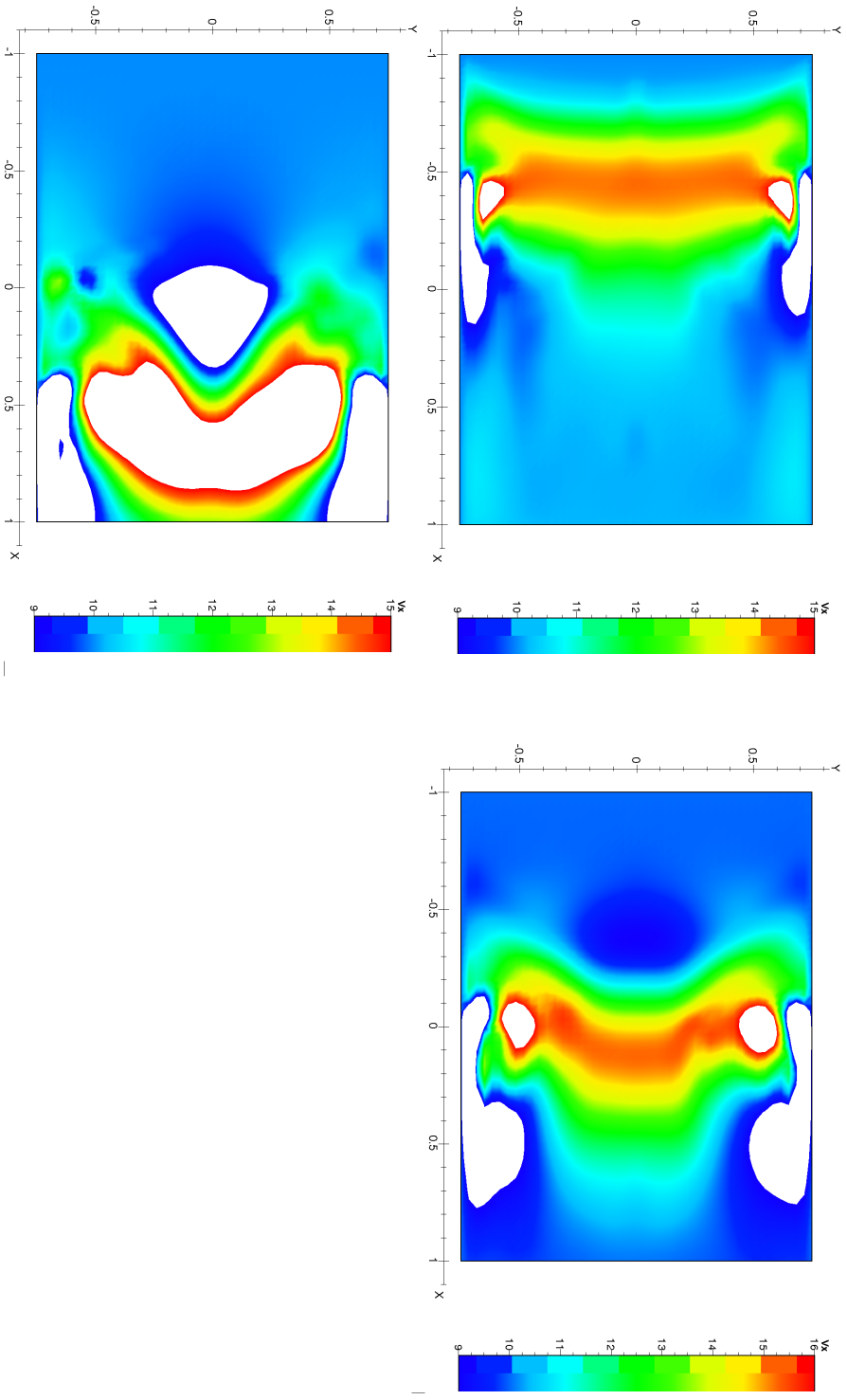


Figure 4.54: Case 4; V_x-component after respectively 0.1s-0.15s-0.2s

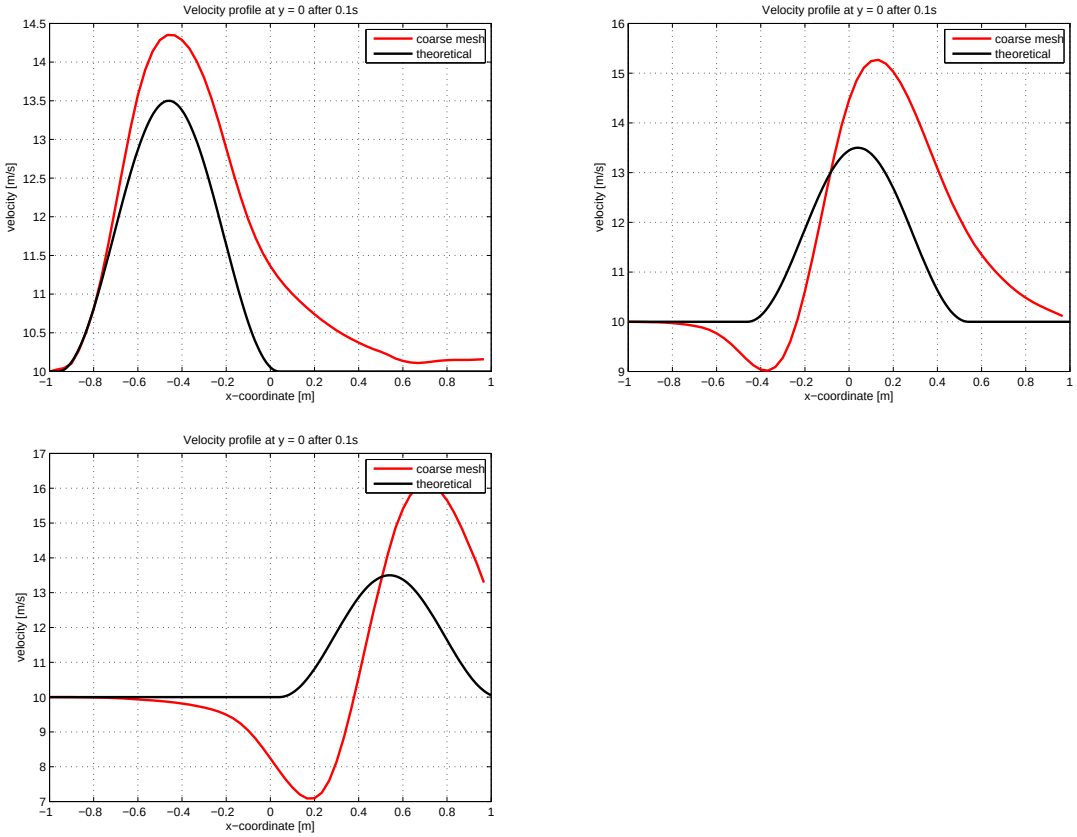


Figure 4.55: Case 2: Velocity profile after respectively 0.1s-0.15s-0.2s

Chapter 5

Effects of Vortex Gust Model on NACA 4415 airfoil

The previous chapter focused on the possibilities of using certain gust models and approaches to simulate an atmospheric gust. It can be concluded that the Vortex Gust Model is - up till now - the most promising approach to simulate a single cosine gust. In this chapter the first effects on a NACA 4415 airfoil are investigated. The NACA 44XX series are a commonly used airfoil section for horizontal axis wind turbines (HAWT) . The choice of the airfoil is not relevant for this investigation. However, the airfoil should not be symmetrical or - if so - should be positioned at a certain angel of attack. Otherwise no lift would be generated. For the ease of the investigation a NACA 4415 airfoil at zero angle of attack is chosen. Figure 5.1 shows the airfoil cross section.

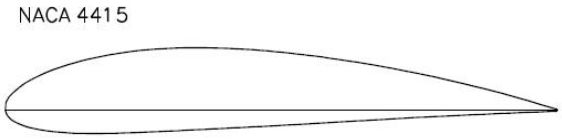


Figure 5.1: NACA 4415 airfoil section

5.1 Vortex Gust Model

In section 4.4 of the previous chapter the possibilities of the VGM were investigated. Since results were satisfying the effects of a gust on a NACA 4415 airfoil will now be investigated for a gust with a length of 10m and an amplitude of 3.5m/s. Only the general characteristics (lift, moment) are considered. In the future more complex characteristics can be investigated such as separation, transition and fluid structure interaction (FSI).
Two different CFD set-ups will be considered, referred to as case C1 and case C5. In case C5 the airfoil is positioned in the middle of the domain (5 chords downstream of the inlet) while in case C1 the airfoil is positioned only 1 chord away from the inlet. Comparing these two cases will show the VGM's quality-loss downstream of the flow which was also discovered in section 4.4.

5.1.1 CFD set-up

Figure 5.3 shows the meshes for both test cases. Both meshes show coarse parts near upper- and lower wall and near the exit. These parts are not relevant for this investigation; a coarse mesh will damp out their contributions to the rest of the domain. The fine mesh part consists of cells with a dimension of $\Delta x = \Delta y = 0.15m$. Near the airfoil boundary the mesh resolution is increased in order to capture the airfoil leading- and trailing edge curvatures. Assuming a Courant-number of 0.8 and a mean flow velocity of 15m/s induce a time-step (Δt) of 0.008s. All other parameters are summarized in table 5.1.

Setting	Case C1	Case C5
inlet-airfoil distance	1 chord	5 chords
L_g	10m	
W_g	3.5m/s	
Grid resolution	60 x 45 cells; $\Delta x = \Delta y = 0.3m$ (inner domain)	
Flow Model	Unsteady Euler with Low Speed Function (default values)	
Inlet	`projectname.windData`	
Initial Velocity	$V_x = 10$, $V_y = 0$	
Initial Pressure	101300Pa	
Initial Temperature	293K	
External Velocity	$V_x = 10$, $V_y = 0$	
External Pressure	101300Pa	
External Temperature	293K	
Multigrid	4 of 5, with coarse grid initialization	
Numerical scheme	Central 2^{nd} order with scalar Jameson dissipation	
Time-step Δt	0.008s	
# iterations per time-step	50	

Table 5.1: CFD set-up for the test cases - VGM NACA 4415 airfoil

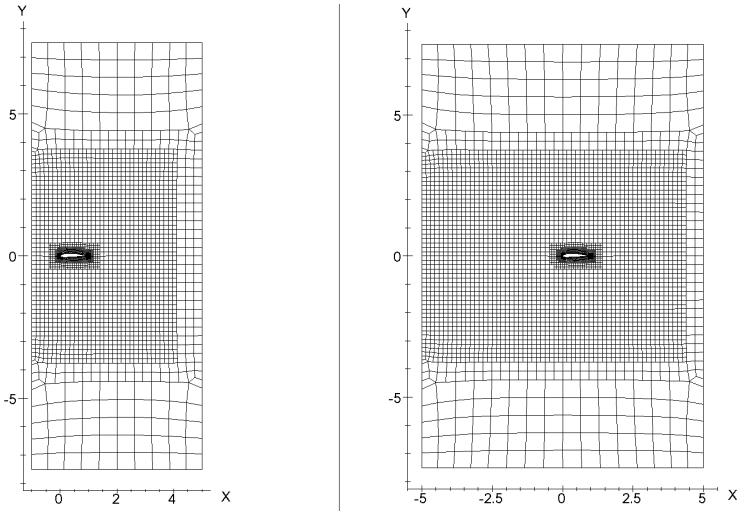


Figure 5.2: Mesh for VGM computation NACA 4415 airfoil; Left: 3356 cells, Right: 4656 cells 60 x 45 cells; $\Delta x = \Delta y = 0.3m$

5.1.2 Results and discussions

In figure 5.3 the behavior of the VGM can be observed for the mesh-without-airfoil. This verifies again the results obtained in section 4.4 which states that the VGM represents an atmospheric gust on a coarse mesh very well. Figure 5.5 and 5.4 are a snap shot of the velocity-fields at the time the airfoil enters the gust column. It also shows that the influence of the V_y-components to the airfoil is very small at that particular instant. The most interesting plot is shown in figure 5.6. Here lift and momentum are presented as a function of the simulation time. The full-line is the result of case C1, the dashed-line is the result of case C5. The cosine-gust results in a cosine-force for both cases. A difference in the force amplitude can be noticed between the two cases. At the beginning of the simulation there is no gust yet. Here, the lift is around 27N. Then, by scaling the lift by a factor 1.35^2 (V^2), a maximum lift force of 49N should be achieved theoretically. The CFD results match this number more or less. Remember that the velocity of the gust increases slightly when propagating through the domain. The curves for case C5 show that the right side of the force bubble is reached out to the right side and the amplitude is increased. The curves for case C1 show more symmetrical results. This confirms the previous conclusion from section 4.4; the quality of the VGM is the highest near the inlet.

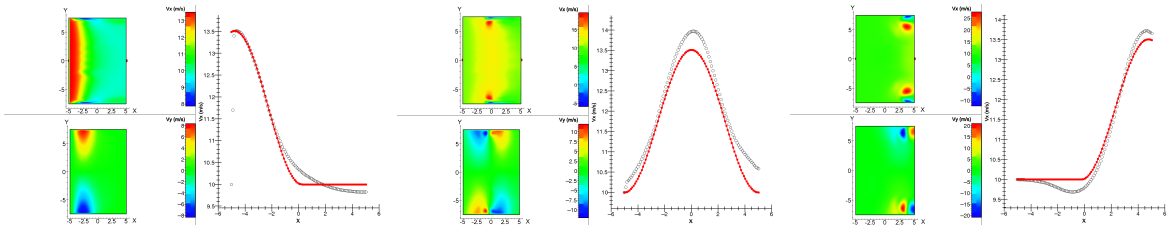


Figure 5.3: Vortex velocity field - U_x for NACA 4415 airfoil.

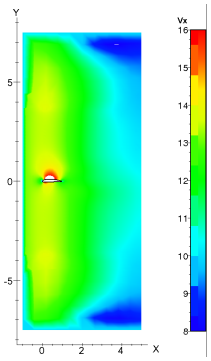


Figure 5.4: Case C1: Velocity field after 0.6s.

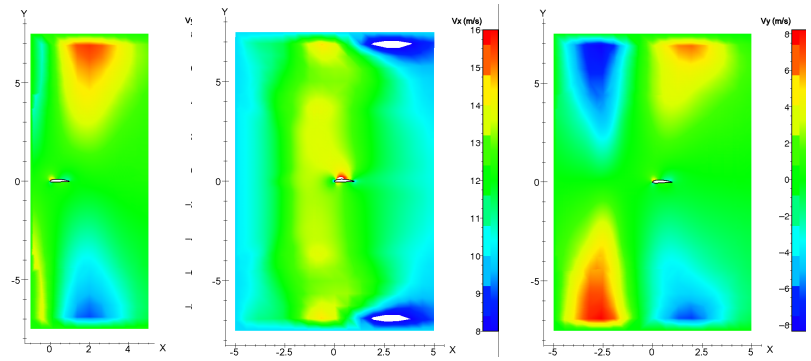


Figure 5.5: Case C5: Velocity field after 1s.

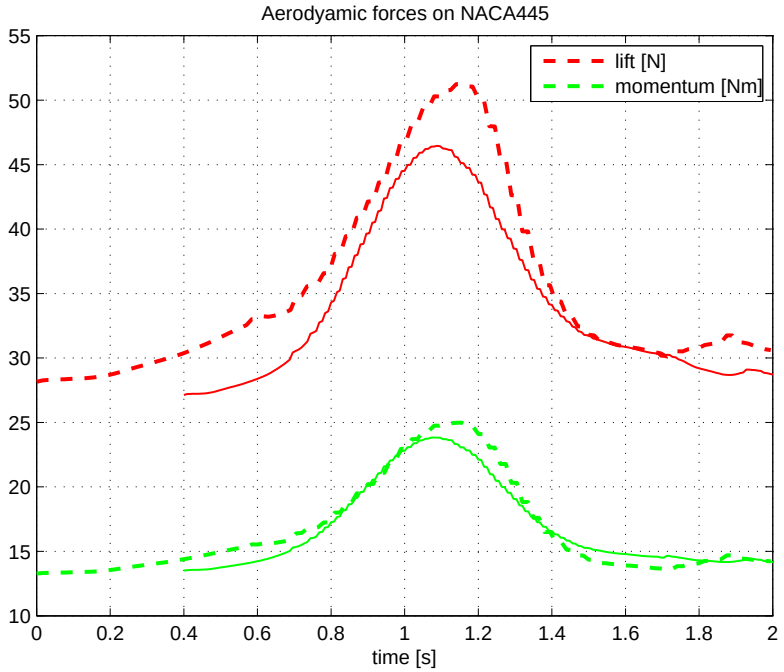


Figure 5.6: Aerodynamic forces on NACA 4415 airfoil during gust simulation: Case C1 (full), Case C5 (dashed)

Chapter 6

Conclusions and recommendations

In this report the possibilities of the simulation of two different gust models using CFD are investigated. The first model that was tested is the J.Mann model. This model consists of a 3D, divergence-free velocity field representing an atmospheric wind field. The use of this velocity for a CFD simulation is investigated. The second model which is used is a mathematical 1-cosine function that describes a single 1D cosine-shaped gust. Four different approaches were tested trying to model this cosine-gust using a CFD simulation. The two models were also mentioned in the International Electronical Commission ([1])

Conclusions and recommendations on J.Mann model The results of using the velocity field from the J.Mann model for a CFD simulation were not satisfactory. The velocity data of the model was used as an unsteady inlet condition for an Euler computation. Next the velocity field propagated through the CFD domain was compared with the original velocity field described by Mann. It turned out that the coherent velocity structures dissipate to fast. This dissipation is caused by different reasons. The main reason is the fact that the coherent structures of the original velocity field disappear. Structures with a high velocity will travel faster than the mean flow while slow structures will run behind the uniform flow. As a result the absent of a coherent structure will increase dissipation to the initial flow conditions. Secondly, also numerical dissipation will have its effects.
Therefore, two reasons can be found why the simulation is not suitable to investigate its effects on an airfoil or wind turbine. First of all the dissipation of the velocity field does not correspond to the physical dissipation of atmospheric structures. Secondly, the dissipation results in a smaller useful-domain part which is too short to investigate the aerodynamic effects on an airfoil or wind turbine.

The J.Mann model is a widely used model in wind engineering. Since the model is based on a spectral tensor for atmospheric surface layer turbulence it includes all atmospheric characteristics in one velocity field. Further investigations on using this model for CFD simulations

is highly advised. Especially to investigate 3D atmospheric effects on an entire wind turbine. However, therefore more complex CFD set-ups are required like is used in [5]. Here a precursor simulation is used in combination with Detached Eddy Simulation.

Conclusion and recommendations on cosine-shaped gust In the first approach to model a cosine-shaped gust an unsteady inlet condition was used. The inlet velocity was calculated by the cosine function. It was expected to find a velocity profile according to the cosine function traveling through the domain. Unfortunately this was not the case. The velocity profile traveled at acoustic convection speeds through the domain instead of the initial uniform flow velocity. After further reflecting it was found that the disturbance imposed at the inlet traveling at a convection speed of the mean flow violates the conservation of mass. To maintain the conservation law the disturbance travels at acoustic convection speeds.

It was expected to solve this issue by adding an extra source term to the Navier Stokes equations. The source term should act as a mass compensator to restore the mass conservation law. By using the cosine velocity function to determine the source terms, solving the Navier Stokes equations should result in the desired velocity profile of a cosine-shaped gust. The results were better but not perfect. The disturbance now traveled correctly along the mean velocity. However, the disturbance did not match the cosine-shaped gust described by the cosine function. This might be caused by the abrupt transition between two domain types. The domain consists of undisturbed parts which are solved without source terms and disturbed parts solved using the source terms. The latter results in a gust. Along the transition, the solver tries to find an equilibrium between the different domains which might explain the different gust shape. The different domain types are not beneficial to investigate the gust effects on an airfoil or wind turbine. To achieve physically correct effects no source terms should be included in the Navier Stokes equations. On the other hand, removing the source terms would result in an unphysical gust. This contradiction makes it difficult to simulate an atmospheric gust using this approach.

The third approach is entirely different. A vertical wall inflow-outflow is used to imitate a cosine-shaped duct. A convergent-divergent throat accelerates and decelerates the uniform horizontal inflow over the duct according to a cosine shape. By moving the wall-inflow downstream the gust will travel along the mean flow according to a desired velocity. The wall-inflow function is derived after combining the conservation law and the cosine-shaped velocity function. Symmetry will guarantee a zero vertical velocity component near the symmetry line y=0.
Results were partly satisfying. The gust showed a cosine shape having a gust length twice as long as the theoretical gust. The amplitude matched more or less for certain wall inflow configurations. From different test cases the following could be observed:

- Narrowing the wall inlet does not result in a shorter gust length but in an increased gust amplitude.
- An increased grid resolution amplifies additional disturbances which is harmful for a proper gust simulation.

- The flow needs a certain adaptation time to adjust to the wall inflow-outflow.

Besides the fact that results were not entirely satisfactory, the approach has some benefits.

- The use of a wall outflow to restore the violation of mass conservation seems to work better than the use of source terms.
- The gust can travel at every desired velocity.
- The inflow velocity function can be replaced by other functions. More complex gust shapes can be easily implemented.
- The CFD set-up is relatively simple.
- No fine mesh is needed to simulate a gust. Grid refinement can be limited to the area around the airfoil.

Therefore further investigations are worthy.
In the report a first improvement of the CFD set-up was suggested whereby the wall in- and outflow function is also used at the vertical inlet condition (instead of uniform flow) and whereby the wall flow only starts after the simulation has started. A second suggestion is the use of the z-direction to impose the wall in- and outflow. Side effects of the wall flows are then transfered to the third dimension which is not relevant for a 2D problem. Due to time limitations these improvements could not yet be made.

The fourth approach is again different. Now the analytical solutions for flow perturbations caused by vortices are used. Since a vortex is divergence-free and the solutions are known, it is excellently suitable for CFD simulations. First, superposition of multiple vortices is used to construct a vorticity field that corresponds to a single uniform cosine-shaped gust. This vortex configuration is then used for the CFD simulation.
This approach showed the best results. Gust length and amplitude correspond well to the theoretical velocity profile. During traveling, the gust amplitude and length slightly increased and also the vertical homogeneity of the gust slightly disappeared. For a gust length of 10m these deviations stayed limited over a distance of 20m.

The influence of a cosine-shaped gust - simulated using the fourth approach - on a NACA 4415 airfoil is investigated. The aerodynamic forces (lift and moment) are directly influenced by the velocity profile of the gust and show the same cosine behavior as the velocity profile. The position of the airfoil in the CFD domain is important. Near the inlet the gust quality is better than downstream of the flow.

General recommendation As was stated in the introduction this Msc. thesis is only the beginning of a wide series of investigations. The first stage is finding a suitable procedure to simulate the atmospheric effects of the atmospheric boundary layer. This thesis is only a beginning in which the possibilities of using the J.Mann model and the possibilities of

simulating a simple horizontal cosine-shaped gust using CFD is investigated. In the future simulations need to be extended by introducing viscosity and thermal effects. The use of LES or DES can extend the modeled flow scales.
Once the atmospheric effects are simulated well, more work can be done on looking to the aerodynamic effects on a 2D airfoil section together with its fluid structure interaction. After, investigations should be extended to the 3D effects of an entire blade and finally an entire wind turbine. Hereby, the relevant flow scales will change. A different gust simulation might be required. The J.Mann model might gain interests in this stage.

Bibliography

[1] *IEC 61400-1.*

[2] P. A. Anderson. *Turbulence*, volume 4. Oxford, 2004.

[3] L. J. Fingersh et al. Wind tunnel testing of nrel's unsteady aerodynamics experiment. In *Collection of the 2001 ASME Wind Energy Symposium Technical Papers Presented at the 39th AIAA Aerospace Science meeting and Exhibit.* January 2001.

[4] D. M. Gerritsma. *Computational Fluid Dynamics.* Delft University Press, December 2002.

[5] L. Gilling et al. Detached eddy simulation of an airfoil in turbulent inflow. 2009.

[6] R. Hadgi. Simulation of two dimensional wind turbine blade in turbulent wind to investigate the applicability of operational modal analysis. May 2010.

[7] N. International. *User Manual Fine/Hexa v2.5.* 5, Avenue Franklin Roosevelt 1050 Brussels Belgium, December 2007.

[8] P. Lucas, A. H. H. van Zuijlen, and H. Bijl. Fast unsteady flow computations with a jacobian-free newton-krylov algoritm. *Journal of Computational Physics*, 229, December 2010.

[9] J. Mann. The spatial structure of neutral atmospheric surface-layer turbulence. 1994.

[10] J. Mann. Wind field simulation. 1998.

[11] J. Manwell. *Wind Energy Explained.* Winley, 2002.

[12] M. Reggio, F. Villalpando, and A. Ilinca. Assessment of turbulence models for flow simulation around a wind turbine airfoil. *Hindawi Publishing Corporation*, 2011(714146):8, 2011.

[13] W. Rozenn et al. Simulation of shear and turbulence impact on wind turbine performance. 2010.

[14] L. N. Sankar et al. Evaluation of turbulence models for the prediction of wind turbine aerodynamics. *AIAA*, (2003-0517), 2003.

[15] Y. Zhou and G. W. W. and. High resolution conjugate filters for the simulation of flows. 2004.

Appendix A

Numeca source code adaptations

This appendix contains three Numeca source-code files which were modified in function of the gust modelation. In `CartesianSubsonicInlets.C` the different inlet conditions and inlet information are defined. In `TimeAccurateDataManager.C` the source-terms used for the cosine gust are programmed. In `VelocityWall.C` the wall-inflow for the duct-gust is defined.
Four different gust types can be distinguished: the vortex-gust, the cosine-gust, the Mann-model and the duct-gust. A specific gust type is selected by adding a correct data-file to the project directory. This data can be velocity data used as input at the inlet face or this can be a header containing parameters for a function used in the code. An overview of the different gust types and files is given in tabel A.1. The gust model `cosGustSource` is standard used with source terms. In the file `TimeAccurateDataManager.C` the source terms can be turned off. The appendix is only limited to the actual modified parts of the code.

Gust type	**Data-file**	**Data**
vortex gust model	`projectname.windData`	header [_Nv _u0 _v0 _T0 gamma] [X Y SIZE STRENGTH]
cosine gust model	`projectname.cosGustSource`	header [_Lg _perc]

J.Mann model	`projectname.MannU`
	`projectname.MannV`
	`projectname.MannW`
		velocity data; each line is a new time step

duct gust model	`projectname.ductGust`	header [Lg Wg Vg steady]

Table A.1: Overview different inlet conditions

The three gust types: vortex gust, cosine gust and J.Mann are programmed in the file `CartesianSubsonicInlets.C` while the duct-gust is programmed in `VelocityWall.C`.

A.1 CartesianSubsonicInlets.C

```
// ------------------------------------------------------------------------------------------
//
// NGHEXA - CartesianSubsonicInlets.C
// ------------------------------------------------------------------------------------------
//
// NUMECA International S.A.
// Av. F.D. Roosevelt, 5
// B- 1050 Brussels
// Belgium
// Tel : +32 2 647.83.11
// Fax : +32 2 647.93.98
// ------------------------------------------------------------------------------------------
//
// DESCRIPTION :
//
// -)   Implementation of the cartesian subsonic inlet boundary conditions
// ------------------------------------------------------------------------------------------
//
// IMPLEMENTATOR : Koen Hillewaert
// ------------------------------------------------------------------------------------------

#include <_Hexa/_hexaNS/_solver/_boundaryCondition/_bcs/CartesianSubsonicInlets.H>
#include <_Hexa/_hexaNS/_parallel/Communicator.h>

#include <iostream>
using std::cerr;
using std::endl;

#include <fstream>
using std::ofstream;
using namespace std;
#include <cstdlib> //for exit function
hReal _NN;
string s;
long length;
//---------------------------------------------------------------------------
ImposeVxVyVzT_ExtrapolateP::ImposeVxVyVzT_ExtrapolateP(hReal *u,hReal *v,hReal *w,hReal *T,
                            PhysicalBc *Patch,int level,DataStructureManager *dataStr)
  :BoundaryCondition(Patch,level,dataStr)
//---------------------------------------------------------------------------
{

  UMH_ECHO(3,"ImposeVxVyVzT_ExtrapolateP::constructor-->IN"<<endl);//VINCENT 5 to 3
  UMH_ECHO(3,"    bc is added to patch "<<_iGS<<" at mg level "<<_iLevel<<endl);//VINCENT 5 to 3

  _u = u;
  _v = v;
  _w = w;
  _T = T;

  _uN = 0;
  _vN = 0;
  _wN = 0;
  _TN = 0;

  oldT  = -1.e0;            // SvZ: Data has been initialized at t=0
  oldTn = -1.e0;            // SvZ: Data has been initialized at t=0

  _initUnsteadyBC();

   int iGSBNode,idNode;
   for (iGSBNode = 0; iGSBNode < *_gSNbBNode; iGSBNode++){
      idNode       = _gSBNode[iGSBNode];
      Vector3D x   = _nodeC[idNode];
   }

  UMH_ECHO(5,"ImposeVxVyVzT_ExtrapolateP::constructor-->OUT"<<endl)//VINCENT 5 to 3

}

//---------------------------------------------------------------------------
ImposeVxVyVzT_ExtrapolateP::ImposeVxVyVzT_ExtrapolateP(hReal *u,hReal *v,hReal *w,hReal *T,
                            hReal *uN,hReal *vN,hReal *wN,hReal *TN,
                            PhysicalBc *Patch,int level,DataStructureManager *dataStr)
  :BoundaryCondition(Patch,level,dataStr)
//---------------------------------------------------------------------------
{
  UMH_ECHO(5,"ImposeVxVyVzT_ExtrapolateP[2]::constructor-->IN"<<endl);
  UMH_ECHO(5,"    bc is added to patch "<<_iGS<<" at mg level "<<_iLevel<<endl);
  UMH_ECHO(5,"    nodal update activated "<<endl);

  _u = u;
  _v = v;
  _w = w;
  _T = T;

  _uN = uN;
  _vN = vN;
```

```
  _wN = wN;
  _TN = TN;

  _getNodeData();

  oldT = -1.e0;          // SvZ: Data has been initialized at t=0
  oldTn =-1.e0;          // SvZ: Data has been initialized at t=0

  _initUnsteadyBC();

//=========================================================VVDC: store cell nodes to file
   ofstream outdata2;
   outdata2.open(projectNameParallel_ + "_coordinates_nodes.dat"); // open and clean .dat file

   int iGSBNode,idNode;
   for (iGSBNode = 0; iGSBNode < *_gSNbBNode; iGSBNode++){
      idNode       = _gSBNode[iGSBNode];
      Vector3D x   = _nodeC[idNode];
      outdata2 << iGSBNode <<" "<< x.x <<" "<< x.y <<" "<< x.z  << endl;
   }
   outdata2.close();
   UMH_ECHO(5,"ImposeVxVyVzT_ExtrapolateP::constructor-->OUT"<<endl)
}

//--------------------------------------------------------------------------
ImposeVxVyVzT_ExtrapolateP::~ImposeVxVyVzT_ExtrapolateP() {
  if (isUnsteadyBC) {
    delete [] _X0;
    delete [] _Y0;
    delete [] _ls;
    delete [] _eta;
  }
}
//--------------------------------------------------------------------------

void ImposeVxVyVzT_ExtrapolateP::_initUnsteadyBC() {
  // SvZ: Check if data exists for unsteady BC
  _unsteadyBCType = 0;
  windDataFile = projectNameParallel_ + ".windData";
  ifstream fin;
  fin.open(windDataFile.data(),ios::in | ios::nocreate);
  if ( !fin ) {
      isUnsteadyBC = false;
  } else {
    _unsteadyBCType = 1;
    isUnsteadyBC = true;
    UMH_ECHO(3,"ImposeVxVyVzT_ExtrapolateP::constructor => READING wind field data from .windData
        file"<<endl);
    fin >> _Nv >> _u0 >> _v0 >> _T0 >> _gamma;
    _X0  = new hReal[_Nv];
    _Y0  = new hReal[_Nv];
    _ls  = new hReal[_Nv];
    _eta = new hReal[_Nv];
    UMH_ECHO(3,"ImposeVxVyVzT_ExtrapolateP::constructor => Initializing " << _Nv << " vortical
        structures."<<endl);
    for (int i=0; i<_Nv; i++)
      fin >> _X0[i] >> _Y0[i] >> _ls[i] >> _eta[i];
    fin.close();
    UMH_ECHO(3,"ImposeVxVyVzT_ExtrapolateP::constructor => Done" << endl);

  }
  // VvdC: for time dependent cos-gust
  if ( !isUnsteadyBC ) {
    windDataFile = projectNameParallel_ + ".cosGustSource";
    ifstream fin;
    fin.open(windDataFile.data(),ios::in | ios::nocreate);
    if ( fin ) {
      isUnsteadyBC = true;
     _unsteadyBCType = 2;
      UMH_ECHO(3,"ImposeVxVyVzT_ExtrapolateP::constructor => READING cosine gust data from .
          cosGustSource file " << endl);
      fin >> _Lg >> _perc;
      _V   = uFree_;
      _wg  = _perc * _V;
      _rho = _thermo->density(pFree_,TFree_);
      _Pt  = pFree_ + 0.5 * _rho * _V * _V;

      fin.close();
      cout << "VvdC: Cosine gust initialization:" << endl;
      cout << "_V    = " << _V << endl;
      cout << "_perc   = " << _perc << endl;
      cout << "_Lg   = " << _Lg << endl;
      cout << "_wg   = " << _wg << endl;
      cout << "_rho  = " << _rho << endl;
      cout << "_Pt   = " << _Pt << endl;

    }
  }
  if ( !isUnsteadyBC ) {
    windDataFile = projectNameParallel_ + ".MannU";
```

```
    ifstream fin;
    fin.open(windDataFile.data(),ios::in | ios::nocreate);
    if ( fin ) {
      _unsteadyBCType = 3;
      isUnsteadyBC = true;
      fin.close();
    }
  }

 if ( !isUnsteadyBC ) {
    windDataFile = projectNameParallel_ + ".MannNodesU";
    ifstream fin;
    fin.open(windDataFile.data(),ios::in | ios::nocreate);
    if ( fin ) {
      _unsteadyBCType = 4;
      isUnsteadyBC = true;
      fin.close();
    }
  }

}


hReal ImposeVxVyVzT_ExtrapolateP::myFuncU(Vector3D x, hReal t) {
  UMH_ECHO(5,"SvZ:myFuncU => x = (" << x.x << " , " << x.y << " , " << x.z << ") , t = " << t <<
       endl);
  hReal Vu;
  hReal DV;
  if ( _unsteadyBCType == 1 ) {
    hReal du    = 0.e0;
    hReal u0    = _u0;
    hReal v0    = _v0;
    hReal T0    = _T0;
    hReal gamma = _gamma;

    for (int i=0; i<_Nv; i++) {
      hReal X0  = _X0[i];
      hReal Y0  = _Y0[i];
      hReal eta = _eta[i];
      hReal ls  = _ls[i];

      hReal X   = X0 + u0 * t;
      hReal Y   = Y0;
      hReal dx  = x.x-X;
      hReal dy  = x.y-Y;
      hReal r2  = dx*dx + dy*dy;

      du += -0.5e0*ls/pi*dy*exp(eta*(1.e0-r2));
      Vu = u0+du;
    }
  // VvdC: calculate cos gust velocity for input
  } else {

    if (t < _Lg/_V) { //allow only one gust period
          DV = 0.5*_wg*(1-cos(2*PI*(t-x.x/_V)*_V/_Lg));
    }else {
          DV = 0.0;
          }
          Vu = _V + DV;

  }

  UMH_ECHO(5,"SvZ:myFuncU => u = " << (Vu) << endl);

  return (Vu);
}


hReal ImposeVxVyVzT_ExtrapolateP::myFuncV(Vector3D x, hReal t) {
  UMH_ECHO(5,"SvZ:myFuncV => x = (" << x.x << " , " << x.y << " , " << x.z << ") , t = " << t <<
       endl);
  hReal Vv;

  if ( _unsteadyBCType == 1 ) {
    hReal dv    = 0.e0;
    hReal u0    = _u0;
    hReal v0    = _v0;
    hReal T0    = _T0;
    hReal gamma = _gamma;

    for (int i=0; i<_Nv; i++) {
      hReal X0  = _X0[i];
      hReal Y0  = _Y0[i];
      hReal eta = _eta[i];
      hReal ls  = _ls[i];

      hReal X   = X0 + u0 * t;
      hReal Y   = Y0;
```

```
      hReal dx  = x.x-X;
      hReal dy  = x.y-Y;
      hReal r2  = dx*dx + dy*dy;

      dv += 0.5e0*ls/pi*dx*exp(eta*(1.e0-r2));
      Vv = v0 + dv;
    }
  } else {
    Vv = 0.0;
  }
  UMH_ECHO(5,"SvZ:myFuncV => v = " << (Vv) << endl);

  return (Vv);
}

hReal ImposeVxVyVzT_ExtrapolateP::myFuncT(Vector3D x, hReal t) {
  UMH_ECHO(5,"SvZ:myFuncT => x = (" << x.x << " , " << x.y << " , " << x.z << ") , t = " << t <<
       endl);
  hReal T;
  hReal DV;

  if ( _unsteadyBCType == 1 ) {
    hReal dT    = 0.e0;
    hReal u0    = _u0;
    hReal v0    = _v0;
    hReal T0    = _T0;
    hReal gamma = _gamma;

    for (int i=0; i<_Nv; i++) {
      hReal X0  = _X0[i];
      hReal Y0  = _Y0[i];
      hReal eta = _eta[i];
      hReal ls  = _ls[i];

      hReal X   = X0 + u0 * t;
      hReal Y   = Y0;
      hReal dx  = x.x-X;
      hReal dy  = x.y-Y;
      hReal r2  = dx*dx + dy*dy;

      dT += -0.0625e0*(gamma-1)*ls*ls/(pi*pi*gamma*eta)*exp(2.e0*eta*(1.e0-r2));
    }
    T = T0 + dT;
  } else {

    if (t<0.4) {
      DV = 0.5*_wg*(1-cos(2*PI*(t-x.x/_V)*_V/_Lg));
    }else {
      DV = 0.0;
        }
      hReal V = _V + DV;
      hReal P = _Pt - 0.5*_rho*V*V;
            T = _compressible->temperature(P,_rho);
  }
  UMH_ECHO(5,"SvZ:myFuncT => T = " << (T) << endl);

  return (T);
}


//---------------------------------------------------------------------------
void ImposeVxVyVzT_ExtrapolateP::numUpdate(hReal ****solInBCell,
                        hReal ****solInBFace)
//---------------------------------------------------------------------------
{
  UMH_ECHO(5,"ImposeVxVyVzT_ExtrapolateP::numUpdate-->IN"<<endl);

  int iGSBFace, jGSBFace;
  int idUpCell;
  // For PARALLEL correction
  int iVar,iCell;

  // PL: Depending on whether or not the _connectivity is used, different loops are used
  int   gSNbBFace;
  int*  GSBFace;
  int   stencil;
  bool  intfCoup = true;
  int unsteadyBCtype = 0;

  if (iJFNK_ == 0 || !_connectivity->useConnectivity()) {
    gSNbBFace = _gSNbBFace[_iLevel][_iGS];
    GSBFace   = _identityVector;
  } else {
    stencil   = _connectivity->getStencil();
    stencil   = MIN(stencil, 3)-1;
    gSNbBFace = _connectivity->nbGSBFace(stencil)[_iGS];
    GSBFace   = _connectivity->iGSBFace()[_iGS];
    if (_connectivity->interfaceCoupling() == 0)
      intfCoup = false;
  }
```

```
356
   if (isUnsteadyBC && (oldT !=  ta_physTime_) ) {
    // SvZ: update time dependent boundary data
    hReal    t   = ta_physTime_;

    //VVDC: open data file U.dat, V.dat, W.dat                                    361
    ifstream finU, finV, finW;
    RWCString MannUFileName = projectNameParallel_ + ".MannU";
    RWCString MannVFileName = projectNameParallel_ + ".MannV";
    RWCString MannWFileName = projectNameParallel_ + ".MannW";
    finU.open(MannUFileName.data(), ios::in | ios::nocreate);                    366
    finV.open(MannVFileName.data(), ios::in | ios::nocreate);
    finW.open(MannWFileName.data(), ios::in | ios::nocreate);

    // if Mann-data exists
    if (finU) {                                                                  371
        unsteadyBCtype = 3;
        finU.seekg(0, ios::beg);
    for (int i=0; i< ta_iter_; i++) {
        getline(finU, s);
    }                                                                            376
    length = finU.tellg();
    finU.seekg(length-1);
    }
    if (finV) {
    unsteadyBCtype = 3;                                                          381
    finV.seekg(0, ios::beg);
    for (int i=0; i< ta_iter_; i++){
        getline(finV, s);
    }
    length = finV.tellg();                                                       386
    finV.seekg(length-1);
    }

    if (finW) {
    unsteadyBCtype = 3;                                                          391
    finW.seekg(0, ios::beg);
    for (int i=0; i< ta_iter_; i++){
        getline(finW, s);
    }
    length = finW.tellg();                                                       396
    finW.seekg(length-1);
    }
//================================================ VVDC:make file to store cell centers
    ofstream outdata;
    if ( ta_iter_ == 0 && _iLevel == mg_nbLevel_-1 ) {                           401
    outdata.open(projectNameParallel_ + "_coordinates_centres.dat");
    cout << " open coordinates_centres.dat" << endl;

    }
//================================================                               406
    for (iGSBFace = 0; iGSBFace < _gSNbBFace[_iLevel][_iGS]; iGSBFace++){
      idUpCell     = _bFaceC[_iLevel][_iGS][iGSBFace];
      Vector3D x   = _coordInBFace[_iLevel][_iGS][iGSBFace];

      if ( unsteadyBCtype == 0 ) {                                               411
        _u[iGSBFace] = myFuncU(x,t);
        _v[iGSBFace] = myFuncV(x,t);
        _T[iGSBFace] = myFuncT(x,t);

      } else {                                                                   416
        // Mann model - only on fine grid
        if ( _iLevel == mg_nbLevel_-1 ) {

          finU >> _u[iGSBFace];
          finV >> _v[iGSBFace];                                                  421
          finW >> _w[iGSBFace];
        }
      }
//================================================   VVDC write coordinates to file - only for
    ta_iter_ == 0
 if ( ta_iter_ == 0 && _iLevel == mg_nbLevel_-1 ) {                             426

      _NN = _NN + 1.0;
      outdata << "  " << _NN << "  " << iGSBFace << "  " << t << "  " << x.x << "  " << x.y << "  "
          << x.z << endl;
   }
//================================================                               431

    }
    oldT = t;

    if (finU) finU.close();                                                      436
    if (finV) finV.close();
    if (finW) finW.close();

    if (outdata) outdata.close();
                                                                                 441
```

```
  }

  // PARALLEL CORRECTION
  // Copy the boundary values into the equivalent ghostcell (using idUpCell)                                    446
  // Communicate these values and copy back to _u etc
  // SHOULD NOT BE HERE, BUT WHERE!?
  if (iParallel_ == 1 && intfCoup)
   {
     // loop over  faces                                                                                        451
     for (iGSBFace = 0; iGSBFace < _gSNbBFace[_iLevel][_iGS]; iGSBFace++){
       idUpCell = _bFaceC[_iLevel][_iGS][iGSBFace];
       // copy any ghost values into tmp_comm
       _tmp_comm[0][idUpCell] = _u[iGSBFace];
       _tmp_comm[1][idUpCell] = _v[iGSBFace];                                                                  456
       _tmp_comm[2][idUpCell] = _w[iGSBFace];
       _tmp_comm[3][idUpCell] = _T[iGSBFace];
     }
     // Communicate - BLOCK
     Communicator::horzComm()->CommunicateBoundary(_iLevel,_tmp_comm,_nbCell,nbVar_,_iBlock,_iGS);           461
     // Copy back into dataCoarse array
     for (int iFace = 0; iFace < _nbFaceBPar[_iLevel][_iGS]; iFace++){
       iGSBFace = _bFaceBPar[_iLevel][_iGS][iFace];
       idUpCell = _bFaceCPar[_iLevel][_iGS][iFace];
       // copy any ghost values into tmp_comm                                                                  466
       _u[iGSBFace] = _tmp_comm[0][idUpCell];
       _v[iGSBFace] = _tmp_comm[1][idUpCell];
       _w[iGSBFace] = _tmp_comm[2][idUpCell];
       _T[iGSBFace] = _tmp_comm[3][idUpCell];
     }                                                                                                         471
   }

  // GLOBAL version
  if (iGravity_ == 0) {
     if (iPreconditioning_ == 0) {                                                                           476
//       for (iGSBFace = 0; iGSBFace < _gSNbBFace[_iLevel][_iGS]; iGSBFace++){
       for (jGSBFace = 0; jGSBFace < gSNbBFace; jGSBFace++){
         iGSBFace = GSBFace[jGSBFace];
     solInBFace[_iLevel][_iGS][0][iGSBFace] = _thermo->density(solInBFace[_iLevel][_iGS][4][iGSBFace
         ],_T[iGSBFace]);
     solInBFace[_iLevel][_iGS][1][iGSBFace] = _u[iGSBFace];                                                  481
     solInBFace[_iLevel][_iGS][2][iGSBFace] = _v[iGSBFace];
     solInBFace[_iLevel][_iGS][3][iGSBFace] = _w[iGSBFace];
     solInBFace[_iLevel][_iGS][4][iGSBFace] = solInBFace[_iLevel][_iGS][4][iGSBFace];
       }
     }                                                                                                       486
     if (iPreconditioning_ == 1) {
//       for (iGSBFace = 0; iGSBFace < _gSNbBFace[_iLevel][_iGS]; iGSBFace++) {
       for (jGSBFace = 0; jGSBFace < gSNbBFace; jGSBFace++){
         iGSBFace = GSBFace[jGSBFace];
     solInBFace[_iLevel][_iGS][0][iGSBFace] = solInBFace[_iLevel][_iGS][0][iGSBFace];                        491
     solInBFace[_iLevel][_iGS][1][iGSBFace] = _u[iGSBFace];
     solInBFace[_iLevel][_iGS][2][iGSBFace] = _v[iGSBFace];
     solInBFace[_iLevel][_iGS][3][iGSBFace] = _w[iGSBFace];
     solInBFace[_iLevel][_iGS][4][iGSBFace] = (_T[iGSBFace] - preco_TRef_);
       }                                                                                                     496
     }
  } else if (iGravity_ == 1) {
     if (iPreconditioning_ == 0){
       for (iGSBFace = 0; iGSBFace < _gSNbBFace[_iLevel][_iGS]; iGSBFace++){
     solInBFace[_iLevel][_iGS][1][iGSBFace] = _u[iGSBFace];                                                  501
     solInBFace[_iLevel][_iGS][2][iGSBFace] = _v[iGSBFace];
     solInBFace[_iLevel][_iGS][3][iGSBFace] = _w[iGSBFace];
     solInBFace[_iLevel][_iGS][4][iGSBFace] = solInBFace[_iLevel][_iGS][4][iGSBFace] + 0.5 *
         solInBFace[_iLevel][_iGS][0][iGSBFace] *
       ( gravity_vector_[0] * (_coordInBFace[_iLevel][_iGS][iGSBFace][0] - _coordInBCell[_iLevel][
           _iGS][iGSBFace][0]) +
         gravity_vector_[1] * (_coordInBFace[_iLevel][_iGS][iGSBFace][1] - _coordInBCell[_iLevel][      506
           _iGS][iGSBFace][1]) +
         gravity_vector_[2] * (_coordInBFace[_iLevel][_iGS][iGSBFace][2] - _coordInBCell[_iLevel][
           _iGS][iGSBFace][2]) );
     solInBFace[_iLevel][_iGS][0][iGSBFace] = _thermo->density(solInBFace[_iLevel][_iGS][4][iGSBFace
         ],_T[iGSBFace]);
       }
     }
     if (iPreconditioning_ == 1) {                                                                           511
       for (iGSBFace = 0; iGSBFace < _gSNbBFace[_iLevel][_iGS]; iGSBFace++) {
     idUpCell = _bFaceC[_iLevel][_iGS][iGSBFace];

     solInBFace[_iLevel][_iGS][0][iGSBFace] = solInBFace[_iLevel][_iGS][0][iGSBFace] + 0.5 *
         _densityInCell[idUpCell] *
       ( gravity_vector_[0] * (_coordInBFace[_iLevel][_iGS][iGSBFace][0] - _coordInBCell[_iLevel][      516
           _iGS][iGSBFace][0]) +
         gravity_vector_[1] * (_coordInBFace[_iLevel][_iGS][iGSBFace][1] - _coordInBCell[_iLevel][
           _iGS][iGSBFace][1]) +
         gravity_vector_[2] * (_coordInBFace[_iLevel][_iGS][iGSBFace][2] - _coordInBCell[_iLevel][
           _iGS][iGSBFace][2]) );
     solInBFace[_iLevel][_iGS][1][iGSBFace] = _u[iGSBFace];
     solInBFace[_iLevel][_iGS][2][iGSBFace] = _v[iGSBFace];
     solInBFace[_iLevel][_iGS][3][iGSBFace] = _w[iGSBFace];                                                  521
     solInBFace[_iLevel][_iGS][4][iGSBFace] = (_T[iGSBFace] - preco_TRef_);
```

```
      }
    }
  }

  UMH_ECHO(5,"ImposeVxVyVzT_ExtrapolateP::numUpdate-->OUT"<<endl);
}

//-----------------------------------------------------------------------
void ImposeVxVyVzT_ExtrapolateP::resUpdate(hReal **residual,
                        hReal ****resInBFace)
//-----------------------------------------------------------------------
{

  UMH_ECHO(5,"ImposeVxVyVzT_ExtrapolateP::resUpdate-->IN"<<endl);

  int iGSBFace;
  int idUpCell;

  if (iPreconditioning_ == 0) {
    for (iGSBFace = 0; iGSBFace < _gSNbBFace[_iLevel][_iGS]; iGSBFace++) {
      idUpCell = _bFaceC[_iLevel][_iGS][iGSBFace];

      resInBFace[_iLevel][_iGS][0][iGSBFace] = - residual[0][idUpCell];
      resInBFace[_iLevel][_iGS][1][iGSBFace] = - residual[1][idUpCell];
      resInBFace[_iLevel][_iGS][2][iGSBFace] = - residual[2][idUpCell];
      resInBFace[_iLevel][_iGS][3][iGSBFace] = - residual[3][idUpCell];
      resInBFace[_iLevel][_iGS][4][iGSBFace] =   residual[4][idUpCell];
    }
  }
  else {
    for (iGSBFace = 0; iGSBFace < _gSNbBFace[_iLevel][_iGS]; iGSBFace++) {
      idUpCell = _bFaceC[_iLevel][_iGS][iGSBFace];

      resInBFace[_iLevel][_iGS][0][iGSBFace] =   residual[0][idUpCell];
      resInBFace[_iLevel][_iGS][1][iGSBFace] = - residual[1][idUpCell];
      resInBFace[_iLevel][_iGS][2][iGSBFace] = - residual[2][idUpCell];
      resInBFace[_iLevel][_iGS][3][iGSBFace] = - residual[3][idUpCell];
      resInBFace[_iLevel][_iGS][4][iGSBFace] = - residual[4][idUpCell];
    }
  }

  UMH_ECHO(5,"ImposeVxVyVzT_ExtrapolateP::resUpdate-->OUT"<<endl);
}
//-----------------------------------------------------------------------
void ImposeVxVyVzT_ExtrapolateP::physUpdate(hReal ****solInBCell,
                        hReal **solInNode,
                        hReal **** solInBFace)
//-----------------------------------------------------------------------
{

  int iGSBNode,idNode;
  UMH_ECHO(3,"ImposeVxVyVzT_ExtrapolateP::physUpdate-->IN"<<endl);
  UMH_ECHO(3,"iPreconditioning_  =  " << iPreconditioning_ << endl);

  if ((_uN)&&(_vN)&&(_wN)&&(_TN)) { // nodal values have been created and passed
  UMH_ECHO(3,"Nodal Values have been created and passed"<<endl);
  //cout << " ta_iter_  | _iLevel | mg_nbLevel_ |  ======== " << ta_iter_ << " | " << _iLevel << " |
       " << mg_nbLevel_ << endl;

    if (iPreconditioning_== 0) {
    cout << "iPreconditioning_ == " << iPreconditioning_ <<endl;
    cout << " isUnsteadyBC === " << isUnsteadyBC <<     cout << " _unsteadyBCType  == " <<
         _unsteadyBCType << " oldTn === " << oldTn << "   ta_physTime_ == " << ta_physTime_ << endl;
      if ( isUnsteadyBC && (oldTn !=  ta_physTime_) ) {
      //if ( isUnsteadyBC ) {

    cout << "loop  isUnsteadyBC && (oldTn !=  ta_physTime_)  = " << isUnsteadyBC << " | " << oldTn
        << " | " <<  ta_physTime_ << endl;
    // SvZ: update time dependent boundary data
        hReal    t   = ta_physTime_;
    for (iGSBNode = 0; iGSBNode < *_gSNbBNode; iGSBNode++){

      idNode       = _gSBNode[iGSBNode];
          Vector3D x   = _nodeC[idNode];

    if ( _unsteadyBCType == 4 ){

        // open data file U.dat, V.dat, W.datthe windd
        ifstream finUnodes, finVnodes, finWnodes;
        RWCString MannNodesUFileName = projectName_ + ".MannNodesU";
        RWCString MannNodesVFileName = projectName_ + ".MannNodesV";
        RWCString MannNodesWFileName = projectName_ + ".MannNodesW";
        finUnodes.open(MannNodesUFileName.data(), ios::in | ios::nocreate);
        finVnodes.open(MannNodesVFileName.data(), ios::in | ios::nocreate);
        finWnodes.open(MannNodesWFileName.data(), ios::in | ios::nocreate);

        // if Mann data exists
        if (finUnodes) {
            finUnodes.seekg(0, ios::beg);
            for (int i=0; i< ta_iter_; i++) {
```

```
                getline(finUnodes, s);
            }
            length = finUnodes.tellg();
            finUnodes.seekg(length-1);
        }   else {
                        cout << MannNodesUFileName.data() << " not found!" << endl;
            }

        if (finVnodes) {
            finVnodes.seekg(0, ios::beg);
            for (int i=0; i< ta_iter_; i++){
                getline(finVnodes, s);
            }
            length = finVnodes.tellg();
            finVnodes.seekg(length-1);
            }   else {
            cout << MannNodesVFileName.data() << " not found!" << endl;
                }
//
        if (finWnodes) {
            finWnodes.seekg(0, ios::beg);
            for (int i=0; i< ta_iter_; i++){
                getline(finWnodes, s);
            }
            length = finWnodes.tellg();
            finWnodes.seekg(length-1);
        }   else {
            cout << MannNodesWFileName.data() << " not found!" << endl;
                } Four d

        finUnodes >> _uN[iGSBNode];
                finVnodes >> _vN[iGSBNode];
        finWnodes >> _wN[iGSBNode];
;
//================================================
    } else {

        _uN[iGSBNode] = myFuncU(x,t);
            _vN[iGSBNode] = myFuncV(x,t);
            _TN[iGSBNode] = myFuncT(x,t);

    }

        solInNode[0][idNode] = _thermo->density(solInNode[4][idNode],_TN[iGSBNode]);
        solInNode[1][idNode] = _uN[iGSBNode];
        solInNode[2][idNode] = _vN[iGSBNode];
        solInNode[3][idNode] = _wN[iGSBNode];

        cout << "solInNode[0] == " << solInNode[0][idNode] << endl;
        cout << "solInNode[1] == " << solInNode[1][idNode] << endl;
        cout << "solInNode[2] == " << solInNode[2][idNode] << endl;
        cout << "solInNode[3] == " << solInNode[3][idNode] << endl;


    }
        oldTn = t;

      } else {

      for (iGSBNode = 0; iGSBNode < *_gSNbBNode; iGSBNode++) {
      idNode    = _gSBNode[iGSBNode];

      // update the solution in corner nodes of boundary faces
      solInNode[0][idNode] = _thermo->density(solInNode[4][idNode],_TN[iGSBNode]);
      solInNode[1][idNode] = _uN[iGSBNode];
      solInNode[2][idNode] = _vN[iGSBNode];
      solInNode[3][idNode] = _wN[iGSBNode];

        }

        }
      }

    if (iPreconditioning_ == 1) {
      for (iGSBNode = 0; iGSBNode < *_gSNbBNode; iGSBNode++) {
    idNode    = _gSBNode[iGSBNode];
    // update the solution in corner nodes of boundary faces

    solInNode[4][idNode] = _TN[iGSBNode] - preco_TRef_;
    solInNode[1][idNode] = _uN[iGSBNode];
    solInNode[2][idNode] = _vN[iGSBNode];
    solInNode[3][idNode] = _wN[iGSBNode];

      }
    }
  }

  UMH_ECHO(3,"ImposeVxVyVzT_ExtrapolateP::physUpdate-->OUT"<<endl); //VINCENT changed from 5 to 3
```

```
}
```

A.2 VelocityWall.C

```
// ----------------------------------------------------------------------------
//
// NGHEXA - VelocityWall.C
// ----------------------------------------------------------------------------
//
// NUMECA International S.A.
// Av. F.D. Roosevelt, 5
// B- 1050 Brussels
// Belgium
// Tel : +32 2 647.83.11
// Fax : +32 2 647.93.98
// ----------------------------------------------------------------------------
//
// DESCRIPTION :
//
// Implementation of the solid wall velocity boundary conditions
// (slip and no-slip conditions)
// ----------------------------------------------------------------------------
//
// IMPLEMENTATOR : Koen Hillewaert
// ----------------------------------------------------------------------------

//-----------------------------------------------------------------
void SlipWall::numUpdate(hReal ****solInBCell,
             hReal ****solInBFace)
//-----------------------------------------------------------------
{

  UMH_ECHO(5,"SlipWall::numUpdate, patch "<<_iGS<<endl);

  // PL: Depending on whether or not the connectivity is used, different loops are used
  int    gSNbBFace;
  int*   GSBFace;
  int    stencil;
  int    iGSBFace;

  //VDC====
   hReal Wg, Lg, V, steady, A0, f, fN, InletNode;
   hReal t = ta_physTime_;
   hReal _wallVelocity[3];
   string ductGustDataFile;
   ductGustDataFile = projectNameParallel_ + ".ductGust";
   ifstream fin;
   fin.open(ductGustDataFile.data(),ios::in | ios::nocreate);
  //VDC====

  if (iJFNK_ == 0 || !_connectivity->useConnectivity()) {
    gSNbBFace = _gSNbBFace[_iLevel][_iGS];
    GSBFace   = _identityVector;
  } else {
    stencil   = _connectivity->getStencil();
    stencil   = MIN(stencil, 3)-1;
    gSNbBFace = _connectivity->nbGSBFace(stencil)[_iGS];
    GSBFace   = _connectivity->iGSBFace()[_iGS];
  }

  hReal dVelocity[3];

  // SvZ: use mov_mode_ = -1 to check the D-GCL: no influence of mesh-velocity on BC
  if ( (iMovingGrid_ == 0) or (mov_mode_ == -1) ) {

   // for (int iGSBFace = 0; iGSBFace < _gSNbBFace[_iLevel][_iGS]; iGSBFace++) {
    for (int jGSBFace = 0; jGSBFace < gSNbBFace; ++jGSBFace) {
      iGSBFace  = GSBFace[jGSBFace];
      // find velocity difference with the wall

    //VDC: Introduce IF-statement to introduce DuctGust
        ==================================================
     Vector3D x   = _BFacePos[_iLevel][_iGS][jGSBFace];
         Vector3D inletfacenode   = _nodeC[0];  // Store value of inlet coordinate of domain
     InletNode = inletfacenode.x;
      // Inletnode is used to allow negative position of domain inlet
        fin.open(ductGustDataFile.data(),ios::in | ios::nocreate);
    if (fin) {
       //incoming wall velocity will create a cosine duct using the parameters from the .ductGust -
           file
          fin >> Lg >> Wg >> V >> steady ;

        if (steady == 1) {  //steady
            if ((x.x-InletNode) < Lg){ // if statement to introduce only one gust period

                _wallVelocity[0] = 0.0;
                if (x.y <0){
                _wallVelocity[1] = -Wg*sin(2*pi*(t-(x.x-InletNode)/V)*V/Lg)*pi/Lg; // df/dx for
                     bottom wall
                       }
                else{
```

```
                _wallVelocity[1] = Wg*sin(2*pi*(t-(x.x-InletNode)/V)*V/Lg)*pi/Lg; // df/dx for top
                     wall
                    }
                    _wallVelocity[2] = 0.0;
                    }
            else{                                                                                    89

                _wallVelocity[0] = 0.0;
                _wallVelocity[1] = 0.0;
                _wallVelocity[2] = 0.0;
                }                                                                                    94
                          }

        else    {   //unsteady

            if (V*t < (x.x-InletNode) & (x.x-InletNode)< V*t + Lg ) { // if statement to introduce     99
                 only one gust period
                _wallVelocity[0] = 0.0;
                if (x.y <0){
                _wallVelocity[1] = -Wg*sin(2*pi*(t-(x.x-InletNode)/V)*V/Lg)*pi/Lg; // df/dx for
                     bottom wall

                       }                                                                             104
                else{
                _wallVelocity[1] = Wg*sin(2*pi*(t-(x.x-InletNode)/V)*V/Lg)*pi/Lg;  // df/dx for top
                     wall
                    }
                    _wallVelocity[2] = 0.0;
                                                                                                     109

                                        }
            else {

                _wallVelocity[0] = 0.0;                                                              114
                _wallVelocity[1] = 0.0;
                _wallVelocity[2] = 0.0;
                 }

            }                                                                                        119

        dVelocity[0] =  solInBFace[_iLevel][_iGS][1][iGSBFace];
        dVelocity[1] =  solInBFace[_iLevel][_iGS][2][iGSBFace] - _wallVelocity[1];
        dVelocity[2] =  solInBFace[_iLevel][_iGS][3][iGSBFace];
                                                                                                     124
        }
    else {

// VDC================================================
                                                                                                     129
       dVelocity[0] =  solInBFace[_iLevel][_iGS][1][iGSBFace];
       dVelocity[1] =  solInBFace[_iLevel][_iGS][2][iGSBFace];
       dVelocity[2] =  solInBFace[_iLevel][_iGS][3][iGSBFace];

          }                                                                                          134
        fin.close();
// SvZ: Bugfix Numeca rotating frame
/*
       if (rot_iVelSy_ == 0) {
     dVelocity[0] = dVelocity[0] + rot_Omega_* _faceY[iGSBFace];                                     139
     dVelocity[1] = dVelocity[1] - rot_Omega_* _faceX[iGSBFace];
       }
*/

       // find normal component of this difference, and subtract from velocity                      144

       hReal dVn = (dVelocity[0] * _normal[iGSBFace][0] +
            dVelocity[1] * _normal[iGSBFace][1] +
            dVelocity[2] * _normal[iGSBFace][2]);
                                                                                                     149
       dVn *= _underRelaxation;

       solInBFace[_iLevel][_iGS][1][iGSBFace] -= dVn * _normal[iGSBFace][0];
       solInBFace[_iLevel][_iGS][2][iGSBFace] -= dVn * _normal[iGSBFace][1];
       solInBFace[_iLevel][_iGS][3][iGSBFace] -= dVn * _normal[iGSBFace][2];                         154


    }

}                                                                                                    159
  else if (iMovingGrid_ == 1) {
//     cout << "SvZ: WARNING: SlipWall::numUpdate Moving mesh DEACTIVATED
     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n";

/*     for (int iGSBFace = 0; iGSBFace < _gSNbBFace[_iLevel][_iGS]; iGSBFace++) {*/
    for (int jGSBFace = 0; jGSBFace < gSNbBFace; ++jGSBFace) {                                       164
      iGSBFace  = GSBFace[jGSBFace];

       // find velocity difference with the wall

       dVelocity[0] =  solInBFace[_iLevel][_iGS][1][iGSBFace] - _BFmoveFaceVelocity[_iLevel][_iGS][  169
```

```
          iGSBFace][0];
      dVelocity[1] =  solInBFace[_iLevel][_iGS][2][iGSBFace] - _BFmoveFaceVelocity[_iLevel][_iGS][
          iGSBFace][1];
      dVelocity[2] =  solInBFace[_iLevel][_iGS][3][iGSBFace] - _BFmoveFaceVelocity[_iLevel][_iGS][
          iGSBFace][2];

// SvZ: Bugfix Numeca rotating frame
/*                                                                                                  174
      if (rot_iVelSy_ == 0) {
    dVelocity[0] = dVelocity[0] + rot_Omega_* _faceY[iGSBFace];
    dVelocity[1] = dVelocity[1] - rot_Omega_* _faceX[iGSBFace];
      }
*/                                                                                                  179

      // find normal component of this difference, and subtract from velocity

      hReal dVn = (dVelocity[0] * _normal[iGSBFace][0] +
           dVelocity[1] * _normal[iGSBFace][1] +                                                    184
           dVelocity[2] * _normal[iGSBFace][2]);

      dVn *= _underRelaxation;

      solInBFace[_iLevel][_iGS][1][iGSBFace] -= dVn * _normal[iGSBFace][0];                        189
      solInBFace[_iLevel][_iGS][2][iGSBFace] -= dVn * _normal[iGSBFace][1];
      solInBFace[_iLevel][_iGS][3][iGSBFace] -= dVn * _normal[iGSBFace][2];

    }
  }                                                                                                 194
  UMH_ECHO(5,"SlipWall::numUpdate-->OUT"<<endl);
}
```

A.3 TimeAccurateDataManager.C

```
// ----------------------------------------------------------------------------------------
//
// NGHEXA - TimeAccurateDataManager.C
// ----------------------------------------------------------------------------------------                4
//
// NUMECA International S.A.
// Av. F.D. Roosevelt, 5
// B- 1050 Brussels
// Belgium                                                                                                 9
// Tel : +32 2 647.83.11
// Fax : +32 2 647.93.98
// ----------------------------------------------------------------------------------------
//
// DESCRIPTION :                                                                                          14
//
// -)   Construction of the time-accurate solver complementary datastructure
// ----------------------------------------------------------------------------------------
//
// IMPLEMENTATOR : Alpesh PATEL                                                                           19
// ----------------------------------------------------------------------------------------


 {
   UMH_ECHO(5,"DataStructureManager::_computeTASourceTerm-->IN [TimeAcurateDataManager.C] " << endl)      24
       ;
   int i,j;
   hReal rhoV,u,v,w,energyV,k,eps,zNuTV,rhoV_old,u_old,v_old,w_old,energyV_old,k_old,eps_old,
        zNuTV_old,L,perc,V,Wg,h1,h2,dfdt,dfdx,dpdx,Vol,rho, Vxt;
   hReal ta_beta0T, ta_betaM1T;

   ...                                                                                                   29

     if (iMovingGrid_ == 0) {
     for (i = 0; i < _nbCell; i++) { // Basic 5 variables common for all methods {
       // initialisation - local copy of the new and old solutions
       // note that solutionXXX[0] and solutionXXX[4] contain the volumes at t[n] and t[n-1]              34

       rhoV      = _solutionTN[0][i];
       rho       = 1.225;
       u         = _solutionTN[1][i];
       v         = _solutionTN[2][i];                                                                    39
       w         = _solutionTN[3][i];
       energyV       = _solutionTN[4][i];
       rhoV_old      = _solutionTNM1[0][i];
       u_old         = _solutionTNM1[1][i];
       v_old         = _solutionTNM1[2][i];
       w_old         = _solutionTNM1[3][i];                                                              44
       energyV_old   = _solutionTNM1[4][i];

//VVDC:add extra cource term START
       string windDataFile;                                                                              49
       windDataFile = projectNameParallel_ + ".cosGustSource";
       ifstream fin;
       fin.open(windDataFile.data(),ios::in | ios::nocreate);


                                                                                                         54
    if (fin) {

    //cout << " cosGustSource - file aanwezig ! "<<endl;

    fin >> L >> perc;                                                                                    59

    V = 10;
        Wg = V + perc*V;
    hReal    t   = ta_physTime_;
        Vector3D x   = _cellC[i];                                                                        64

    //cout << " V == " << V << " t == " << t << " x.x == " << x.x << endl;
        if ( ( x.x > V*t ) | (x.x < V*t-L) ){
        //if ( x.x > V*t )  {
                                                                                                         69
            //cout << " no change in source term " << endl;

            u = 10;
            v = 0;
            w = 0;                                                                                       74

            _taSourceTerm[0][i] = (ta_beta0_*rhoV + ta_betaM1_*rhoV_old)/ta_physTimeStep_;
                _taSourceTerm[1][i] = (ta_beta0_*rhoV*u + ta_betaM1_*rhoV_old*u_old)/
                    ta_physTimeStep_;
                _taSourceTerm[2][i] = (ta_beta0_*rhoV*v + ta_betaM1_*rhoV_old*v_old)/
                    ta_physTimeStep_;
                _taSourceTerm[3][i] = (ta_beta0_*rhoV*w + ta_betaM1_*rhoV_old*w_old)/                    79
                    ta_physTimeStep_;
                _taSourceTerm[4][i] = (ta_beta0_*energyV + ta_betaM1_*energyV_old)/ta_physTimeStep_;

        } else {
```

```
            //cout << "== COSINE source term ==" << endl;

            Vxt = V + Wg/2*(1-cos(2*pi*(t-x.x/V)*(V/L)));
            h1 = -Wg*rhoV*sin(2*pi*(t-x.x/V)*V/L*pi)/L;

            dfdt  = rhoV*Wg*sin(2*pi*(t-x.x/V)*V/L)*V*pi/L;
            dffdx = -rhoV*Wg*Wg*(1-cos(2*pi*(t-x.x/V)*V/L)*sin(2*pi*(t-x.x/V)*V/L)*pi)/L;
            dpdx  = -rhoV*Vxt*-Wg/L*sin(2*pi*(t-x.x/V)*V/L)*pi;

            h2 = dfdt + dffdx + dpdx/1.2;

                _taSourceTerm[0][i] = (ta_beta0_*rhoV + ta_betaM1_*rhoV_old)/ta_physTimeStep_ - h1
                    ;
                _taSourceTerm[1][i] = (ta_beta0_*rhoV*u + ta_betaM1_*rhoV_old*u_old)/
                    ta_physTimeStep_ - h2 ;
                _taSourceTerm[2][i] = (ta_beta0_*rhoV*v + ta_betaM1_*rhoV_old*v_old)/
                    ta_physTimeStep_;
                _taSourceTerm[3][i] = (ta_beta0_*rhoV*w + ta_betaM1_*rhoV_old*w_old)/
                    ta_physTimeStep_;
                _taSourceTerm[4][i] = (ta_beta0_*energyV + ta_betaM1_*energyV_old)/ta_physTimeStep_;
        }

    } else {

    //cout << "no cosinegust file found --> no change in source term " << endl;
        _taSourceTerm[0][i] = (ta_beta0_*rhoV + ta_betaM1_*rhoV_old)/ta_physTimeStep_;
        _taSourceTerm[1][i] = (ta_beta0_*rhoV*u + ta_betaM1_*rhoV_old*u_old)/ta_physTimeStep_;
        _taSourceTerm[2][i] = (ta_beta0_*rhoV*v + ta_betaM1_*rhoV_old*v_old)/ta_physTimeStep_;
        _taSourceTerm[3][i] = (ta_beta0_*rhoV*w + ta_betaM1_*rhoV_old*w_old)/ta_physTimeStep_;
        _taSourceTerm[4][i] = (ta_beta0_*energyV + ta_betaM1_*energyV_old)/ta_physTimeStep_;
    }

     }

     if (model_ == iNSTurb1) // Spalart-Allmaras

       for (i = 0; i < _nbCell; i++) {
     // initialisation - local copy of the new and old _solutions
     // note that _solutionXXX[5] also contains the volumes at t[n] and t[n-1]

     zNuTV      = solTurbTN[0][i];
     zNuTV_old  = solTurbTNM1[0][i];
     _taSourceTerm[5][i] = (ta_beta0T*zNuTV + ta_betaM1T*zNuTV_old)/ta_physTimeStep_;
       }

     if (model_ == iLESSMA || model_ == iLESWAL)
       for (i = 0; i < _nbCell; i++) _taSourceTerm[5][i] = 0.0;

     if (model_ == iNSTurb2) // K-Epsilon

       for (i = 0; i < _nbCell; i++) {
     // initialisation - local copy of the new and old _solutions
     // note that _solutionXXX[0] and _solutionXXX[4] contain the volumes at t[n] and t[n-1]

     rhoV       = solTurbTN[0][i];
     k      = solTurbTN[1][i];
     eps        = solTurbTN[2][i];
     rhoV_old   = solTurbTNM1[0][i];
     k_old      = solTurbTNM1[1][i];
     eps_old        = solTurbTNM1[2][i];

     _taSourceTerm[5][i] = (ta_beta0T*rhoV*k + ta_betaM1T*rhoV_old*k_old)/ta_physTimeStep_;
     _taSourceTerm[6][i] = (ta_beta0T*rhoV*eps + ta_betaM1T*rhoV_old*eps_old)/ta_physTimeStep_;
       }
  }
  else if (iMovingGrid_ == 1) {
    for (i = 0; i < _nbCell; i++) { // Basic 5 variables common for all methods
      // initialisation - local copy of the new and old solutions
      // note that solutionXXX[0] and solutionXXX[4] contain the volumes at t[n] and t[n-1]

      rhoV      = _solutionTN[0][i] * _volTN[i];
      u         = _solutionTN[1][i];
      v         = _solutionTN[2][i];
      w         = _solutionTN[3][i];
      energyV       = _solutionTN[4][i] * _volTN[i];
      rhoV_old      = _solutionTNM1[0][i] * _volTNM1[i];
      u_old         = _solutionTNM1[1][i];
      v_old         = _solutionTNM1[2][i];
      w_old         = _solutionTNM1[3][i];
      energyV_old   = _solutionTNM1[4][i] * _volTNM1[i];

      _taSourceTerm[0][i] = (ta_beta0_*rhoV + ta_betaM1_*rhoV_old)/ta_physTimeStep_;
      _taSourceTerm[1][i] = (ta_beta0_*rhoV*u + ta_betaM1_*rhoV_old*u_old)/ta_physTimeStep_;
      _taSourceTerm[2][i] = (ta_beta0_*rhoV*v + ta_betaM1_*rhoV_old*v_old)/ta_physTimeStep_;
      _taSourceTerm[3][i] = (ta_beta0_*rhoV*w + ta_betaM1_*rhoV_old*w_old)/ta_physTimeStep_;
      _taSourceTerm[4][i] = (ta_beta0_*energyV + ta_betaM1_*energyV_old)/ta_physTimeStep_;
```

```
                                                                                                    169
    }

    if (model_ == iNSTurb1) // Spalart-Allmaras
      for (i = 0; i < _nbCell; i++) {
    // initialisation - local copy of the new and old _solutions                                    174
    // note that _solutionXXX[5] also contains the volumes at t[n] and t[n-1]

    zNuTV      = _solutionTN[5][i] * _volTN[i];
    zNuTV_old  = _solutionTNM1[5][i] * _volTNM1[i];
    _taSourceTerm[5][i] = (ta_beta0T*zNuTV + ta_betaM1T*zNuTV_old)/ta_physTimeStep_;              179
      }

    if (model_ == iLESSMA || model_ == iLESWAL)
      for (i = 0; i < _nbCell; i++) _taSourceTerm[5][i] = 0.0;
                                                                                                    184
    if (model_ == iNSTurb2) // K-Epsilon

      for (i = 0; i < _nbCell; i++) {
    // initialisation - local copy of the new and old _solutions
    // note that _solutionXXX[0] and _solutionXXX[4] contain the volumes at t[n] and t[n-1]        189

    rhoV       = _solutionTN[0][i] * _volTN[i];
    k      = _solutionTN[5][i];
    eps        = _solutionTN[6][i];
    rhoV_old   = _solutionTNM1[0][i] * _volTNM1[i];                                              194
    k_old      = _solutionTNM1[5][i];
    eps_old        = _solutionTNM1[6][i];

    _taSourceTerm[5][i] = (ta_beta0T*rhoV*k + ta_betaM1T*rhoV_old*k_old)/ta_physTimeStep_;
    _taSourceTerm[6][i] = (ta_beta0T*rhoV*eps + ta_betaM1T*rhoV_old*eps_old)/ta_physTimeStep_;   199
      }
  }

...
```

Appendix B

windFieldGenerator.m

This matlab tool makes it possible to construct a desired velocity field by combining vortices. The tool stores the vortice information [X Y SIZE STRENGTH] in a data file (`projectname.windData`). In the first part of the code some general parameters should be defined. They are used to construct a field which corresponds to the desired size. In the second part the vortex configuration is defined. The third part calculates the velocity-field. The fourth part generates the plots. The final part writtes the data to `projectname.windData`.

```
%%
close all
clear all
clc;

Xbc    = -10;       % location of inlet BC
Ybc    = [0 20];    % height of inlet BC
pcty   = 1.0;       % percentage of height to be used by vortices
T      = 2;         % total simulation time
Tstart = 0.1;       % start-up time - time before first vortex core could enter through BC

u0    =  10;
v0    =   0;
T0    = 293;
gamma = 1.4;

L = 3
W = 5;

%% Start of computation of wind field

Xmin = Xbc-u0*T;                            % The domain range is defined
Xmax = Xbc-u0*Tstart;

Ymin = pcty*Ybc(1);
Ymax = pcty*Ybc(2);

xvals = [-20];                              % Locations for vortice(s)
yvals = [4 5 6 7 8 9 10 11 12 13 14 15 16]; % Locations for vortice(s)

Nv = length(xvals)*length(yvals)*2;
Vparam = zeros(Nv,4);
idx = 1;
for X = xvals
    for Y = yvals
        Vparam(idx,:) = [X Y 100 0.05];          % [ X Y SIZE STRENGTH]
        idx = idx + 1;                           % By combining a certain vortex size and strength
        Vparam(idx,:) = [X -Y 100 0.05];         % different velocity fields can be generated
        idx = idx + 1;                           % SIZE and STRENGTH will influence each other!
    end
end
Nv = length(Vparam(:,1));

if (1)
```

```
    Nx = 200;
    Ny = 100;
    X  = (ones(Ny+1,1) * [Xmin + (0:Nx)*(Xbc -Xmin)/Nx])';
    Y  =  ones(Nx+1,1) * [Ybc(1) + (0:Ny)*(Ybc(2)-Ybc(1))/Ny];

    u_ = zeros(Nx+1,Ny+1);
    v_ = zeros(Nx+1,Ny+1);
    T_ = zeros(Nx+1,Ny+1);

    for i=1:Nv
        dx = X - Vparam(i,1);
        dy = Y - Vparam(i,2);
        r2 = dx.*dx + dy.*dy;
        u_ = u_ - Vparam(i,3) / 2 / pi * dy .* exp(Vparam(i,4)*(1-r2));
        v_ = v_ + Vparam(i,3) / 2 / pi * dx .* exp(Vparam(i,4)*(1-r2));
        T_ = T_ - (gamma - 1) * Vparam(i,3)^2 / ...
                  (16 * Vparam(i,4) * gamma * pi * pi) * ...
                  exp(2*Vparam(i,4)*(1-r2));
    end

    vX = X(:,1);
    vY = Y(1,:)';

    figure(1);
    title('Ux perturbation');
    surface(vX,vY,u_','LineStyle','None');

    figure(2);
    title('Uy perturbation');
    surface(vX,vY,v_','LineStyle','None');

    figure(3);
    title('T perturbation');
    surface(vX,vY,T_','LineStyle','None');

    figure(4);
    title('Ux perturbation along y=0');
    plot(vX,u_(:,1),'b-');
    hold on

    figure(4);
    title('cosine gust');
    cos = W/2*(1-cos(2*pi*(m-X(:,1))/L));

end

if (0)
    ID = 1;
    fname = ['windfield_' int2str(ID) '.dat'];
    while ( exist(fname,'file') )
        ID = ID + 1;
        fname = ['windfield_' int2str(ID) '.dat'];
    end
    yn = ' ';
    while ( (yn ~= 'y') && (yn ~= 'n') )
        yn = input(['Do you wish to write field data to ' fname '? (y/n) : '],'s');
        if ( isempty(yn) )
            yn = ' ';
        end
    end
    if ( yn == 'y' )
        FID = fopen(fname,'w+');
        fprintf(FID,'%6.0f %12e %12e %12e %12e\n',[Nv u0 v0 T0 gamma]);
        fprintf(FID,'%12e %12e %12e %12e\n', Vparam');
        fclose(FID);
    end
end
```

Appendix C

Derivation of the source terms

In this appendix the derivation of the source terms used for the simulation of the cosine gust model: `CGSource`.
Starting with the Euler equations:

mass:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} = 0 \tag{C.1}$$

momentum in x-direction:

$$\frac{\partial \rho u}{\partial t} + \frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} + \frac{\partial p}{\partial x} = 0 \tag{C.2}$$

momentum in y-direction:

$$\frac{\partial \rho v}{\partial t} + \frac{\partial \rho uv}{\partial x} + \frac{\partial \rho v^2}{\partial y} + \frac{\partial p}{\partial y} = 0 \tag{C.3}$$

energy:

$$\frac{\partial \rho E}{\partial t} + \frac{\partial \rho Eu}{\partial x} + \frac{\partial \rho Ev}{\partial y} + \frac{\partial Ep}{\partial x} = 0 \tag{C.4}$$

where $E = e + \frac{V^2}{2}$.
Assuming an incompressible flow ($\frac{\partial \rho}{\partial t} = 0$) and the following solutions for the velocity:

$$u = u_0 + f(x,t) = u_0 + \frac{W_g}{2}\left[1 - \cos\left(2\pi(t - \frac{x}{V_g})(\frac{V_g}{L_g})\right)\right] \tag{C.5}$$

$$v = 0 \tag{C.6}$$

equation C.1 to C.4 can be written as:

$$\rho\frac{\partial}{\partial x}f(x,t) = h_1(x,t) \tag{C.7}$$

$$\rho\frac{\partial}{\partial t}f(x,t) + \rho\frac{\partial}{\partial x}f^2(x,t) + \frac{\partial P}{\partial x} = h_2(x,t) \tag{C.8}$$

$$\frac{\rho}{2}\frac{\partial}{\partial t}f^2(x,t) + \rho e\frac{\partial}{\partial x}f(x,t) + \frac{\rho}{2}\frac{\partial}{\partial x}f^2(x,t) + \frac{\rho}{2}\frac{\partial}{\partial x}f^3(x,t) = h_3(x,t) \tag{C.9}$$

where $h_1(x,t)$, $h_2(x,t)$ and $h_3(x,t)$ are the source terms for equation C.1, C.2 and C.4. The source term for the momentum in y-direction (C.3) is zero.

The relation for the pressure gradient ($\frac{\partial P}{\partial x}$) can be found by combining the Bernoulli equation

$$P = -\frac{1}{2}\rho V^2 \tag{C.10}$$

and equation C.6. Taking the derivative will result in

$$\begin{aligned}\frac{\partial P}{\partial x} &= -\rho f(x,t)\frac{\partial f(x,t)}{\partial x} \\ &= -\rho\left[u_0 + \frac{W_g}{2}\left[1-\cos\left(\frac{2\pi V_g}{L_g}(t-\frac{x}{V})\right)\right]\right]\frac{W_g\pi}{L_g}\sin\left(2\pi(t-\frac{x}{V_g})\frac{V}{L_g}\right)\end{aligned} \tag{C.11}$$

Now equation C.6 and C.11 are substituted in equations C.7 to C.9 to find the expressions for the source terms:

$$h_1(x,t) = -\rho\frac{W_g\pi}{L_g}\sin\left(2\pi(t-\frac{x}{V_g})\frac{V_g}{L_g}\right) \tag{C.12}$$

$$\begin{aligned}h_2(x,t) = \rho\frac{W_gV_g\pi}{L_g}\sin\left(2\pi(t-\frac{x}{V_g})\frac{V_g}{L_g}\right) - \\ \rho\frac{W_g^2\pi}{2L_g}\left[1-\cos\left(2\pi(t-\frac{x}{V_g})\right)\frac{V_g}{L_g}\right]\sin\left(2\pi(t-\frac{x}{V_g})\frac{V_g}{L_g}\right)\end{aligned} \tag{C.13}$$

$$\begin{aligned}h_3(x,t) = \rho\frac{W_g^2V_g}{2L_g}\left[1-\cos\left(\frac{2\pi V_g}{L_g}(t-\frac{x}{V_g})\right)\right]\sin\left(2\pi(t-\frac{x}{V_g})\frac{V_g}{L_g}\right) - \\ \rho e\frac{W_g\pi}{L_g}\sin\left(2\pi(t-\frac{x}{V_g})\frac{V_g}{L_g}\right) - \\ \rho\frac{W_g^2\pi}{2L_g}\left[1-\cos\left(2\pi(t-\frac{x}{V_g})\right)\frac{V_g}{L_g}\right]\sin\left(2\pi(t-\frac{x}{V_g})\frac{V_g}{L_g}\right) - \\ \rho\frac{3W_g^3\pi}{8L_g}\left[1-\cos\left(2\pi(t-\frac{x}{V_g})(\frac{V_g}{L_g})\right)\right]^2\sin\left(2\pi(t-\frac{x}{V_g})\frac{V_g}{L_g}\right)\end{aligned} \tag{C.14}$$