CODEN:LUTEDX/(TEIE-5290)/1-194(2011)

Industrial Electrical Engineering and Automation

Modeling and Simulation of a Vertical Wind Power Plant in Dymola/Simulink

Joel Petersson
Pär Isaksson

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Modeling and Simulation of a Vertical Wind Power Plant in Dymola/Modelica

J. Petersson & P. Isaksson

December 22, 2011

Abstract

A small wind power plant connected to the grid has been modeled in Modelica/Dymola and controlled using external controllers written in C++. The small wind power plant consists of three wind power units with a nominal power of 3kW and one grid connection interconnected with an internal DC-grid. All the controls needed for control and optimization for the operation of the individual parts in the plant was developed and implemented. Apart from this a managing control for the entire plant was developed and implemented.

The control developed in this project was implemented using an external static library interconnected with Dymola, the External Object approach of implementing objects in Modelica was also tested. The optimization algorithms developed for the wind turbine was done in a way so that no measurements of the wind speed is needed. The controls was developed so that they can achieve a number of different tasks such as Reactive Power Compensation and Island Control.

The modeling was done in Modelica using Dymola. In order to model the power electronics involved in the system the Spot library has been utilized. Models for the wind turbine was developed and tested.

The models and control algorithms were tested by running different test cases. The test cases involves both normal operation and island operation. The results was compared with grid codes from Denmark and Sweden.

Acknowledgements

We would like to thank all the nice people working at Modelon AB and all the other students who made their master thesis at Modelon AB for their encouragement, good input and nice company. Without all of you our master thesis would been much harder to do.

We would especially like to thank our mentors at Modelon AB Jens Pålsson and Hubertus Tummescheit, and our supervisor at Lund University Faculty of Engineering Jörgen Svensson for their continuous input and comments. Without them this project would not been possible to finalize.

Contents

Nomenclature

β	Blade pitch angle.
λ	Tip speed ratio.
λ_i	Used to calculate C_p.
$\mu(\omega)$	Velocity dependent friction coefficient.
ω	Rotational speed.
ω_e	Electrical rotational speed.
ω_m	Mechanical rotational speed.
ω_r	Rotational speed of the rotor.
ω_T	Rotational speed of the turbine.
Ψ	Turbine angle.
ψ_d	Magnetic flux in d-axes.
ψ_q	Magnetic flux in q-axes.
Ψ_T	Half the angle for when rotor blade is considered to be behind the tower.
ρ	Air density.
τ	Torque.
τ^{ref}	Torque reference.
$\tau_{MaxStat}$	Maximum static friction torque before flange becomes “unstuck”.
$\tau_{Sliding}$	Sliding friction.
θ	Electrical angle.
A	Area swept by the rotor.
a	Acceleration.
c	Gain used in the optimizing algorithms.
C_p	Efficiency of the wind turbine.
c_{1-6}	Used to calculate C_p.
c_{geo}	Geometrical constant.
D_S	Duty cycle.
e	Error, used in control algorithms.
E_S	Total energy loss.
$E_{S,cond}$	Conduction loss.
$E_{S,off}$	Turn off loss.
$E_{S,on}$	Turn on loss.
f	Frequency.
f_n	Normal force.
f_{sw}	Switching frequency.
h_s	Height of wind obstacle.
i	Current.
I_0	Conduction current.

I_D Current through diode.
i_d Direct current.
i_q Quadrature current.
i_{di} Iron loss current d-axes.
i_{dm} Magnetizing direct current.
i_{qi} Iron loss current q-axes.
i_{qm} Magnetizing quadrature current.
J Inertia.
k Gain, used in control.
K_e Permanent magnetic flux linkage.
L Inductance.
N_i Anti windup time constant, used by PI- and PID-controllers.
P Power, if electrical active power
P Proportional gains used by PI- and PID-controllers.
P_c Copper losses in the generator.
P_i Iron losses in the generator.
P_m Mechanical losses in the generator.
P_s Stray losses in the generator.
$P_{D,cond}$ Conduction loss in diode.
$P_{D,rr}$ Power loss in diode due to reverse recovery.
P_{in} Input power.
P_{out} Output power.
$P_{S,cond}$ Power loss due to conduction.
$P_{S,off}$ Power loss due to turn off.
$P_{S,on}$ Power loss due to turn on.
$P_{S,sw}$ Switching losses in transistor.
pp Number of pole pairs in the generator.
Q_{rr} Reverse recovery charge.
R Resistance.
r Radius of the turbine.
R_D Diode resistance.
R_i Iron loss resistance.
R_s Resistance.
S Apparent power.
T_i Integration time constant, used by PI- and PID-controllers
T_ω Mechanical torque.
t_{cond} Conduction time.
t_{off} Turn off time.
t_{on} Turn on time.
T_{sw} Time for one switching period.
T_{ts} Torque component caused by tower shadowing.
t_{ts} Design parameter for tower shadow effect.
v Voltage.
v_d Direct voltage.
v_q Quadrature voltage.
V_{AC} AC-Voltage level.

$V_{D(on)}$ Voltage drop over the diode.
V_{D0} Voltage drop over diode at zero current.
V_{DC} DC-Voltage level.
V_{S0} Forward voltage at zero current.
w Wind speed.
w_{base} Base wind component.
w_{gust} Wind gust component.
w_{noise} Wind noise component.

Chapter 1

Introduction

1.1 Background

Wind power is at the moment in a globally expansive phase were many different kinds of technical wind power solutions from different suppliers exists. In most solutions power electronics is incorporated in different degrees. Most wind power plants currently operates with a horizontal axes turbine, however vertical axes turbines is an interesting future alternative. Some Swedish companies are interested in testing vertical plants, and some have already been built. One of these companies in Skåne has the ambition of supplying its own commercial vertical wind power plant. They have already tested a first prototype using a five-bladed turbine at 2 meters diameter with a nominal power of 2kW. The company is mainly interested in the small scale market consisting of plants connected to, for example, houses and boats, with a nominal power of 0.2-30kW. The pros of using a vertical axes turbine is a simple and robust construction with a minimal amount of moving parts, which allows for a cost efficient wind power plant with an aspect both to investment, operation and maintenance. Other pros are independence of wind direction, less sensitivity to turbulence, simple blade profiles and lower noise levels.

There are also some rules and guidelines related to wind power where the ability to contribute with active/reactive power-, voltage- and frequency- control is becoming increasingly important.

1.2 Motivation

The Purpose of this project is to design a model of a wind power plant using the Modelica based tool Dymola which contains many different model libraries, including Spot which is well suited for modeling of both the power-electronics and the generator. The task consists of modeling the wind turbine, generator, power electronics and grid. It also consists of constructing all the different controllers needed to run the power plant at both maximum efficiency and at a specific power level. The control algorithms should preferably be written in C/C++ and interconnected with Dymola. The system should be modeled as one or several wind power units connected to the grid through back to back full inverters as shown in figure 1.1. The models should, if possible, also be compared and validated against real measurement-data obtained from a wind power plant that is planned to be constructed on the roof of LTH during the spring.

Apart from modeling a single wind power unit a small wind power plant consisting of three separate units should also be modeled and a simple control algorithm for managing the plant should be developed and tested. The impact from the power plant on the grid should also be studied and compared to wiring and grid codes from both Sweden and Denmark. This since Denmark has higher demands on the controllability of their wind power plants. The plant is supposed to do general decisions of how both the grid side inverter and

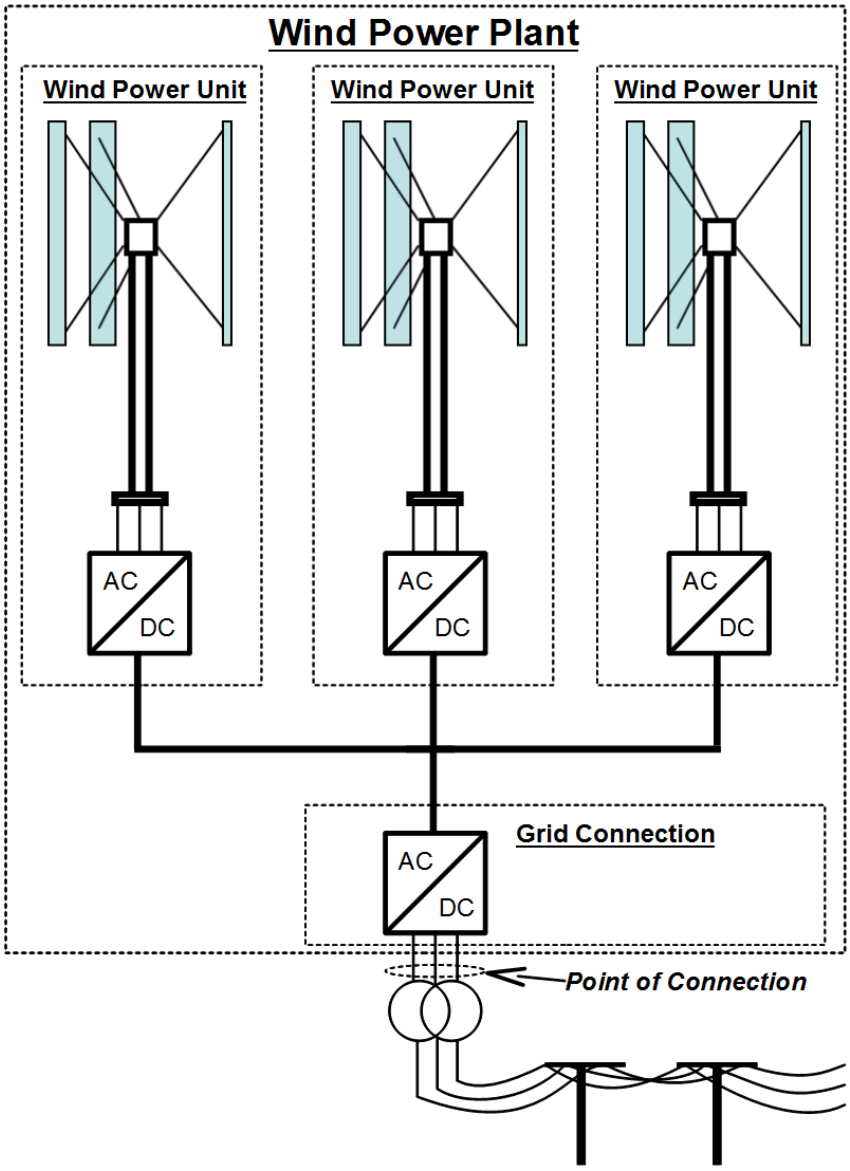


Figure 1.1: Schematic picture of how the wind power system is set up.

the different wind power units should operate, depending on the customers[1] demands.

1.2.1 Project Limitations

The project tasks are limited to the tasks listed below.

- Develop a simple model for a wind turbine unit.
- Develop and test models for simulation of a small wind power plant connected to the grid.

[1]The customer in our case is the grid owner to which we sell the generated power.

- Develop control strategies for optimization and control of the plant in C/C++.
- Interconnect the control strategies developed in C/C++ with the power plant model in Dymola.
- Evaluate different methods for control of power, frequency and voltage level.
- Develop some simple control strategies for running a wind power plant.
- If possible verifies the models against real measurement-data.

Optional tasks that should be done, if time permits, is listed below.

- Evaluate the wind power plant's impact on the grid in comparison to the Swedish and Danish grid's wiring requirements.
- Consider the possibility of running the models in real-time.

Chapter 2

Wind Power

Wind is an abundant source of renewable free energy and has been used for thousands of years by mankind. In the beginning wind power was used to propel small sailing boats over short distances [1]. Ashore wind power has been used as windmills to run machines such as flour mills. Over the years wind power has developed, and in 1887-1888 the first wind power plant was built by Charles F. Brusch [2]. Wind power today is considered to be one of the world's renewable energy sources and the wind power industry has been growing rapidly the past years. One reason to this is that once a wind power plant is installed the cost of running the plant is small in comparison to other energy sources, only some regular maintenance is needed. The wind power in Sweden increased with 78 percent between 2008 and 2010, and in 2010 there were 1'665 wind power plants in operation in Sweden with a total of 2'019 MW installed power [3].

(a) Giromill design of a VAWT placed in Falkenberg, Sweden [5]

(b) Darrieus design of a VAWT [6]

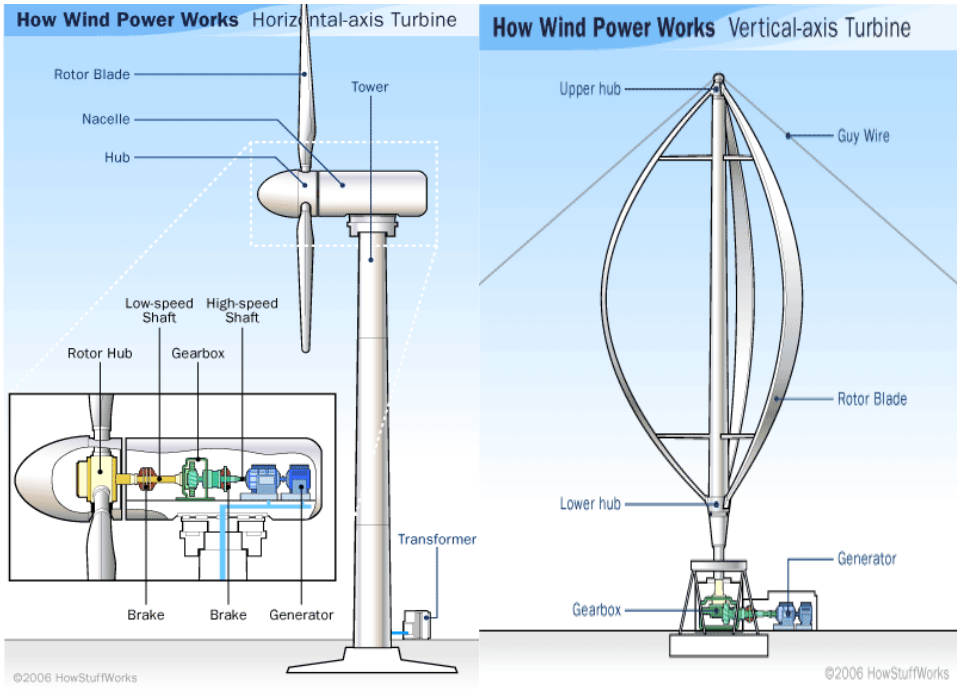
Figure 2.1

2.1 Technology

In general there are two different types of wind power turbines, vertical axis wind turbine, VAWT, and horizontal axis wind turbine, HAWT. The end result from the two types is the same, they both produce electrical energy from kinetic energy captured from the wind. The typical differences are the power efficiency, the production cost, aesthetic form and the noise factor.

2.1.1 Vertical Axis Wind Turbine

The first type of the wind turbine is the VAWT. Two models, the giromill design and the Darrieus design, of a VAWT can be seen in figure 2.1. The only difference between the two models is the design of the blade shape. A VAWT has the main rotor arranged vertically. The key advantages are that a VAWT can operate and start to generate power at a lower wind speed which means that the wind turbine can be mounted at a lower place, this feature makes it possible to mount a VAWT on top of buildings [4]. The VAWT is independent of the wind direction which means that it does not have to turn into the wind as the HAWT needs to. All the heavy parts of a VAWT, such as generator, gearbox etc., can be placed at the base of the plant connected by a shaft, which provides easy access for maintenance work. The VAWT has a smaller amount of movable parts which can break, and the noise factor are also lower compared to a HAWT due to that a VAWT is spinning more quietly.



(a) Components in a HAWT [8] **(b)** Components in a VAWT [8]

Figure 2.2

2.1.2 Horizontal Axes Wind Power

The second type of wind turbine is the HAWT. A HAWT has the generator and the main rotor shaft placed in the top of the tower. Most of the HAWT has gearboxes to gear up the slow rotation of the blades to quicker rotation for the generator. The key advantages are the higher efficiency for a HAWT compared to the VAWT. Further on large HAWT has less material expenditure per square meter of surface covered compared to large VAWT, which means less material will be used to construct larger HAWT and the weight will be less.

The dominating type in commercial use is by far the HAWT due to its higher efficiency and lighter weight relative the area swept by the rotor [7].

2.1.3 Components in a Wind Turbine

The basic parts of a HAWT and a VAWT can be seen in 2.2. The both wind turbine types uses essentially the same components.

The basic parts of a wind turbine are:

- Rotor blades - The Rotor blades are used to capture the energy of the wind. The wind makes the blades rotate which creates rotational energy. The larger area swept by the rotor blades the more kinetic energy in the wind has a potential of being converted into rotational energy. Larger blades means a larger tower which also means access to faster winds but

it will in turn increase the difficulties in maintenance work for plants with the generator mounted at top, typically HAWT.

- Shaft - The Shaft is connected to the rotor in one end and a generator in the other. The shaft's purpose is to transfer the rotational energy from the rotor to the generator.
- Generator - The generator is the component that will transform the mechanical energy into electrical energy. A generator uses the property of electromagnetic induction to produce electrical voltage. There exist a lot of different types of generator, all of which performs the transformation in a slightly different way. One example of a simple generator consists of a permanently magnetized rotor and a stator with electrical windings. The shaft is connected to the magnetized rotor in the generator, when the shaft rotates the rotor in the generator also rotates which will induce a voltage in the windings.
- Gearbox - A gearbox may be used to gear up the speed of the shaft connected to the generator. However a gearbox means higher losses, more components with a possibility to break and more costs, which has lead to that more and more manufactures has started to construct wind turbines without it.
- Anemometer - The anemometer is used to measure the current wind speed. The wind speed is crucial to know in order to decide whether the wind turbine should be active or not.

2.2 Theory

2.2.1 Wind Characteristic

The wind speed and turbulence is directly tied to the amount and the quality of the produced power in a wind power plant. The wind varies over time and place, the variations are dependent of the temperature, local topography and ground cover variations. The wind fluctuates over both the long and short time scope stochastically, which make it hard to predict the wind speed. The mean wind speed over a year is called the annual mean wind speed, which is a crucial parameter when planning and optimizing a wind power plant.

The wind speed is lower near ground since it is heavily affected by buildings, trees, hills and other obstacles, this property is called wind gradient. The wind gradient is a quantity which indicates the change of the wind speed relative to the height over the ground.

2.2.2 Wind Variations in Time

The wind varies over both the short and long time scope, both variations affects the power production of a wind power plant. The wind speed variations can according to [9] be divided into the following categories:

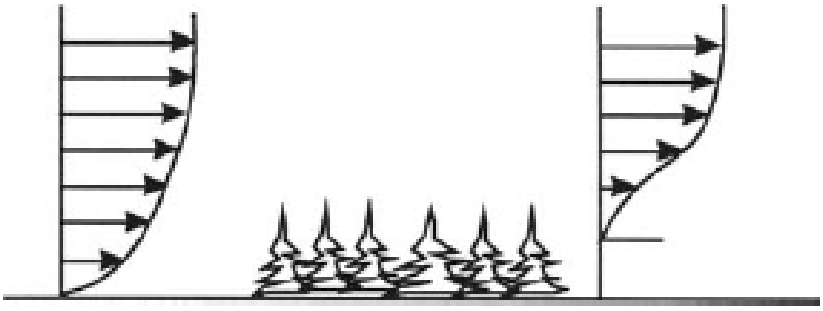


Figure 2.3: How the wind gradient changes when wind passes an area with obstacles [9]

- Inter-annual - the variations of the wind speed which affect the power production in wind power plant on a time scale greater than a year.
- Annual - the variations of the wind speed are season based. For example in Sweden the mean wind speed is higher during the winter months than during the summer months [10]. This is an advantage since the energy consumption is higher during the winter.
- Diurnal - this type of wind speed variations is most due to the different temperatures during the day. A typical diurnal variation is a low wind speed at sunrise, an increase during the morning with a peak during the day and a decrease during the evening, with a low wind speed at midnight. During the summer diurnal variations are greater due to higher temperature differences.
- Short-term - the short term wind speed variations includes gusts and turbulence. A gust is a discrete event and can be described by the rise time, amplitude and fall time. Turbulence on the other hand can be seen as noise of the wind, it occurs when the wind passes an obstacle. The short-term variation affects the power quality produced by a wind power plant. This makes it important to consider the short-term variations in order to design a proper control system of a wind power plant. However a larger turbine is less affected by gusts than smaller turbines, due to its size.

2.2.3 Wind Gradient

The wind speed is affected of local topographical, in most natural terrain the surface changes significantly from place to place. This affects the local wind profile. Figure 2.3 illustrates this effect, the wind gradient varies with height when the wind is passing a forest. The wind gradient is smoother higher above ground. This means that a tall wind power plant will have a smoother and higher wind speed than shorter one.

The turbulence will be higher close to the ground level or right after an obstacle, see figure 2.4. When the wind is passing a building with the height

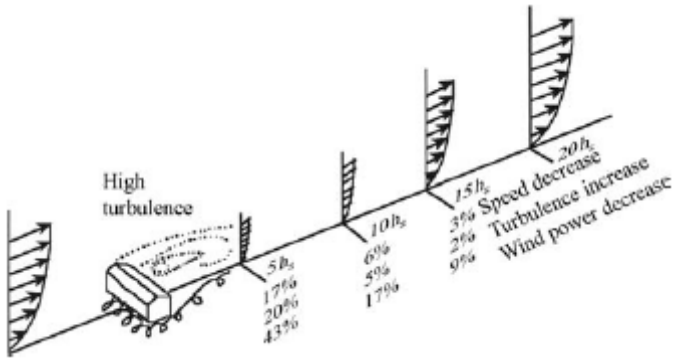


Figure 2.4: Illustration of turbulence in the wind when passing a building [9]

h_s. The wind speed, turbulence level and the wind power are heavily affected by the building and their values are not back to normal until after >15 h_s.

2.2.4 Wind Power Production

A wind turbine's power production depends on the interactions between the wind and the rotor. When the wind blows against the wind turbine it is resulting in a pressure difference between the rotors' leaf front and back. This difference in pressure results in a force which drives the rotor blade. The power extracted from the wind by the rotor can be described as the kinetic energy of the wind times an efficiency coefficient. The efficiency coefficient is varying with the pitch angle of the blades and tip speed ratio. The tip speed ratio is the wind speed relative the speed of the tip of the turbine's blades.

The mechanical power P in a vertical wind power unit can be described by equation 2.1. Where ρ is the density of the air, A is the area swept by the rotor blades, w the wind speed and C_p the efficiency coefficient which can be described according to equation 2.2, according to [11] and [12], where c_i is system dependent constants, β is the pitch angle of the blades and λ the tip speed ratio which can be calculated according to equation 2.4 where w_T is the turbine rotational speed and w is the wind speed.

$$P = \frac{1}{2} \cdot \rho \cdot A \cdot \mathrm{w}^3 \cdot C_p \tag{2.1}$$

$$C_p(\lambda, \beta) = c_1 \cdot (\frac{c_2}{\lambda_i} - c_3 \cdot \beta - c_4) \cdot e^{\frac{-c_5}{\lambda_i}} + c_6 \cdot \lambda \tag{2.2}$$

$$\frac{1}{\lambda_i} = \frac{1}{\lambda + 0.08 \cdot \beta} - \frac{0.035}{\beta^3 + 1} \tag{2.3}$$

$$\lambda = \frac{\omega_T \cdot r}{\mathrm{w}} \tag{2.4}$$

2.3 Losses

As with all systems a wind power plant is subject to losses. Some losses occurs in the mechanical system and is mainly caused by friction and some occurs in the electrical system and is mainly caused by heat generation in resistive components.[1] The goal of our models is to represent the real world as good as possible which means that these losses need to be incorporated by our models. The losses have been divided into two main types, mechanical and electrical.

2.3.1 Mechanical Losses

The mechanical losses in the power plant are mainly due to friction. Usually the largest contribution to the mechanical losses is from the gearbox, since the wind power unit modeled in this project is not using a gearbox this will not be considered. The main mechanical losses of the power plant instead occur in the shaft or drive train. A common, and acceptable, way to model a drive train is as a number of discrete masses connected with springs that is defined by damping and stiffness coefficients [13]. The complexity of the model can, and should, be varied depending on the simulation purposes, when studying effects such as torsional fatigue, a more advanced drive train model needs to be taken into consideration. In our case the goal of the simulation is to study the electrical and overall general behavior of the system, the drive train is therefore modeled as a single inertia connected to the generator. The losses in the shaft were modeled as friction between the shaft and its housing using a Coulomb friction model.

Bearing Friction

The model chosen to model the losses in the shaft was the bearingFriction[2] model in Modelica's standard library. When the rotational speed not is zero the model uses a table of rotational speeds and breaking torques in order to do linear interpolations of the current breaking torque. When the rotational speed is zero the shaft gets "stuck" and will not move until enough torque is applied in order to make it rotate.

2.3.2 Electrical Losses

The electrical losses in this system are present in all the electrical components. The two major loss components that need to be taken into consideration are the generator and inverters.

[1] Note that the efficiency with which we absorb the wind energy is not considered as a loss in such, see section 2.2.4 for more information about the efficiency coefficient, C_p.

[2] See bearingFriction model in *Modelica Standard Library (Version 3.2)*

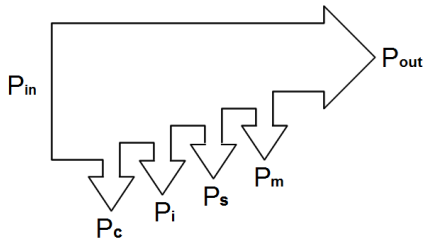
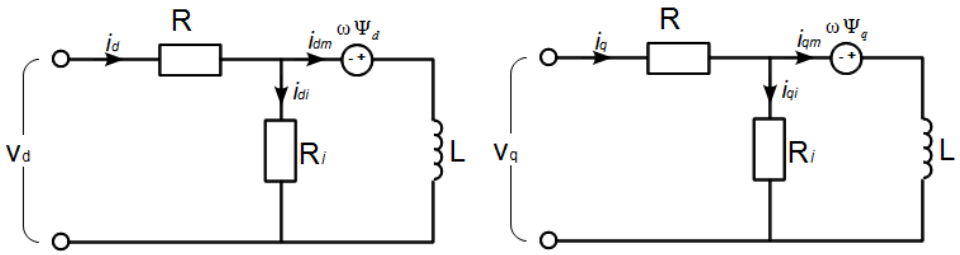


Figure 2.5: Power flow in the generator, [14]



(a) Equivalent circuit in d-axes **(b)** Equivalent circuit in q-axes

Figure 2.6: Equivalent circuits for the generator in d-q axes, [14]

Generator Losses

The losses in the generator can be considered to consist of four different parts [14], copper losses, P_c, iron losses, P_i, stray losses, P_s, and mechanical losses, P_m, as depicted in figure 2.5. Equivalent circuits for the generator in the dq-reference frame is shown in figure 2.6, the currents, i_d and i_q, in the figure is divided into a magnetizing part, i_{dm} and i_{qm}, and an iron loss part, i_{di} and i_{qi}. The total magnetic flux can then be calculated according to equation 2.5 where K_e denotes the emf constant, or permanent magnet flux linkage.

$$\begin{cases} \psi_d = Li_{dm} + K_e \\ \psi_q = Li_{qm} \end{cases} \tag{2.5}$$

By using the equivalent circuits, see figure 2.6 expressions for the magnetizing currents according to equation 2.6 is obtained [14]. Also the equation for the steady state voltages can be obtained according to equation 2.7, assuming steady state conditions [14].

$$\begin{cases} i_{dm} = i_d + \dfrac{\omega_e L}{R_i}\left(i_q - \dfrac{\omega_e K_e}{R_i}\right) \\ i_{qm} = i_q - \dfrac{\omega_e L}{R_i}\left(i_d + \dfrac{K_e}{L}\right) \end{cases} \tag{2.6}$$

$$\begin{cases} v_d = Ri_d - \omega_e \psi_q \\ v_q = Ri_q + \omega_e \psi_d \end{cases} \tag{2.7}$$

By using equation 2.5 to 2.7 an expression for the electrical input power can be calculated according to equation 2.8, where the first term can be identified as the copper losses, P_c, the second term as the iron losses, P_i, and the third term consists of the stray losses, P_s, and mechanical losses, P_m, and the output power, P_{out}. [14]

$$P_{in} = \overbrace{R\left(i_d^2 + i_q^2\right)}^{P_c} + \frac{\overbrace{\omega_e^2\left(\psi_d^2 + \psi_q^2\right)}^{P_i}}{R_i} + \overbrace{\omega_e K_e i_{qm}}^{P_s+P_m+P_{out}} \tag{2.8}$$

The output torque can then be described as the output power divided by the mechanical rotational speed according to equation 2.9.

$$\tau = \frac{P_{out}}{\omega_m} = \frac{\omega_e K_e i_{qm} - P_m - P_s}{\omega_m} = \left[\begin{array}{l} \dfrac{\omega_e}{\omega_m} = pp \\ \\ \dfrac{P_m + P_s}{\omega_m} = \tau_{ms} \end{array}\right] = pp \cdot K_e i_{qm} - \tau_{ms} \tag{2.9}$$

Inverter Losses

The most important loss to include in the models is most likely the loss from the power electronics, such as the inverters, this since the cooling system of the inverter needs to be designed[15]. In order to do a good thermal design a good estimation of the losses in the different parts of the inverter is needed. A three-phase inverter generally consists of six transistors, two per phase, each with one anti parallel diode, or freewheeling diode, see figure 2.7.

The power loss in the transistor can be described as the instantaneous voltage times the instantaneous current through the transistor, $S(t) = v(t) \cdot i(t)$. However ideally the transistor is in one of two modes, either it is not conducting any current, resulting in a large voltage drop over the transistor, or it is conducting a large current, without any voltage drop over the transistor. Both these modes produce zero losses since either the voltage drop or the current is zero. In reality however there are two separate main types of losses, the switching losses and the conductance losses[15]. The switching losses occurs every time a switch is made, the conductance loss occurs when the transistor is conducting, see figure 2.8. The total loss for one switching period can be divided into three parts, one conduction loss and two switching losses. To get the total energy loss over one switching period the instantaneous power loss is integrated over one switching period, T_{sw}, see equation 2.10.

$$E_S(T_{sw}) = \int_{T_{sw}} P_S(\tau)d\tau = E_{S,on}(T_{sw}) + E_{S,cond}(T_{sw}) + E_{S,off}(T_{sw}) \tag{2.10}$$

As can be seen in figure 2.8 the highest instantaneous loss occurs at the switching instants, the reason for this, as can be seen in the top of the two

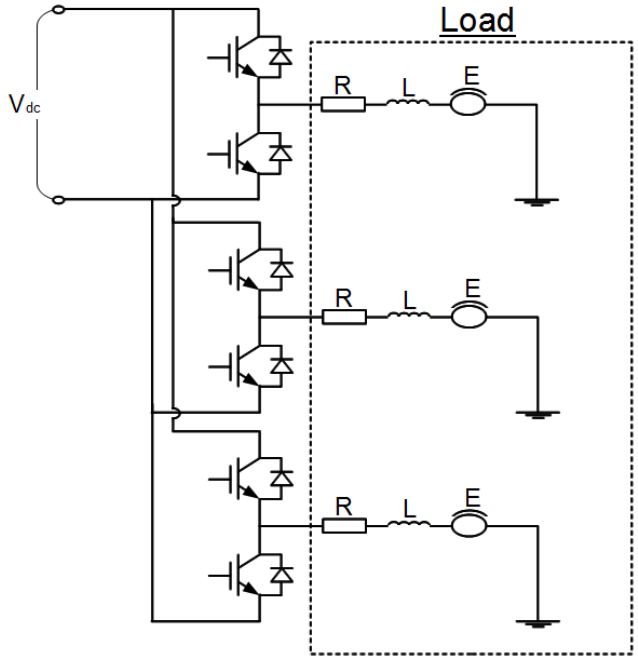


Figure 2.7: Schematic picture of a three phase Inverter.

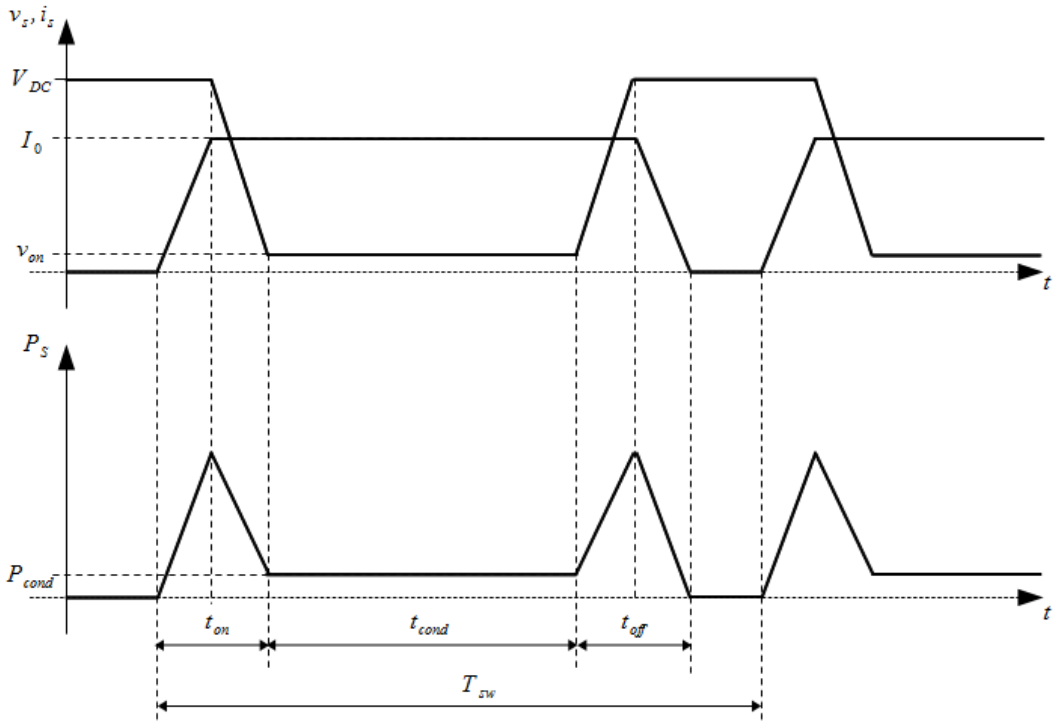


Figure 2.8: **Top diagram:** Voltage drop over the transistor and current flowing through the transistor.
Bottom Diagram: Instantaneous power loss. [15]

diagrams, is that for a short time instant there exists both a large voltage drop and a large current flowing through the transistor. By integrating over the on/off switching time the total energy loss during that time is obtained, see equation 2.11 and 2.12.

$$E_{S,on}(T_{sw}) = \int_{t_{on}} P_S(\tau)d\tau = V_{DC} \cdot I_0 \cdot \frac{t_{on}}{2} \tag{2.11}$$

$$E_{S,off}(T_{sw}) = \int_{t_{off}} P_S(\tau)d\tau = V_{DC} \cdot I_0 \cdot \frac{t_{off}}{2} \tag{2.12}$$

The conduction losses on the other hand are due to the fact that the transistor is resistive when conducting. This results in a small voltage drop over the transistor at the same time as there is a large current flowing through the device. By integrating over the conduction time the total conduction energy loss is obtained, see equation 2.13, where $V_{S(on)}$ can be calculated as a forward voltage at zero current, V_{S0}, times the resistance R_s, see equation 2.14.

$$E_{S,cond}(T_{sw}) = \int_{t_{cond}} P_S(\tau)d\tau = V_{S(on)} \cdot I_o \cdot t_{cond} \tag{2.13}$$

$$V_{S(on)} = V_{S0} + R_s \cdot I_0 \tag{2.14}$$

When the total energy loss during one switching period is known the average power loss can be calculated by dividing the total energy loss during one switching period with the time for one switching period, according to equation 2.15. The average power loss caused by the switching instant is calculated as the total energy loss when switching on/off times the switching frequency, f_{sw}, see equation 2.16 and 2.17. The average power loss due to conduction of current is calculated according to equation 2.18. The voltage drop over the transistor, $V_{s(on)}$, times the current flowing through the transistor, I_0, times the duty cycle, D_S, of the transistor, note that the duty cycle of a single transistor not is the same as the duty cycle for the whole inverter [15].

$$P_S(T_{sw}) = \frac{E_S(T_{sw})}{T_{sw}} = P_{S,on}(T_{sw}) + P_{S,off}(T_{sw}) + P_{S,cond}(T_{sw}) \tag{2.15}$$

$$P_{S,on}(T_{sw}) = \frac{E_{S,on}(T_{sw})}{T_{sw}} = E_{S,on}(T_{sw}) \cdot f_{sw} = \frac{V_{DC} \cdot I_0 \cdot t_o n}{2} \cdot f_{sw} \tag{2.16}$$

$$P_{S,off}(T_{sw}) = \frac{E_{S,on}(T_{sw})}{T_{sw}} = E_{S,off}(T_{sw}) \cdot f_{sw} = \frac{V_{dc} \cdot I_0 \cdot t_{off}}{2} \dot{f}_{sw} \tag{2.17}$$

$$P_{s,cond}(T_{sw}) = \frac{E_{S,cond}(T_{sw})}{T_{sw}} = V_{S(on)} \cdot I_0 \cdot \frac{t_{cond}}{T_{sw}} = V_{S(on)} \cdot I_0 \cdot D_S \tag{2.18}$$

By combining the turn on- and off- losses an expression for the total switching losses can be obtained according to equation 2.19.

$$P_{S,sw}(T_{sw}) = P_{S,on}(T_{sw}) + P_{S,off}(T_{sw}) \tag{2.19}$$

However in order to perform this calculation the turn-on, t_{on}, and turn-off, t_{off}, times are needed. The turn-on/off times usually varies with the DC-link voltage and are therefore in most cases not directly given in the data sheets, instead the turn-on and turn-off energies, $E_{on,n}$ and $E_{off,n}$, are specified at a specific DC-link voltage, $V_{DC,n}$, and load condition, $I_{0,n}$. Therefore in order to get a good estimation of the switching losses the turn-on/off energies should be used and scaled according to equation 2.20 and 2.21.[15]

$$E_{S,on}(T_{sw}) = \frac{E_{on,n}}{V_{DC,n} \cdot I_{0,n}} \cdot V_{DC} \cdot I_0 \tag{2.20}$$

$$E_{S,off}(T_{sw}) = \frac{E_{off,n}}{V_{DC,n} \cdot I_{0,n}} \cdot V_{DC} \cdot I_0 \tag{2.21}$$

Apart from the transistor losses the inverter losses also include the losses from the freewheeling diodes. The losses in the diode can be divided into two separate parts, the conduction losses, $P_{D,cond}$, and the reverse recovery loss, $P_{D,rr}$ [15]. The conduction losses of the diode is calculated in the same way as the conduction losses of the transistor, see equation 2.22, where the voltage drop over the diode, $V_{D(on)}$, can be calculated according to equation 2.23, and the duty-cycle of the diode can be estimated according to equation 2.24.

$$P_{D,cond}(T_{sw}) = V_{D(on)} \cdot I_0 \cdot D_D \tag{2.22}$$

$$V_{D(on)} = V_{D0} + R_D \cdot I_D \tag{2.23}$$

$$D_D \approx 1 - D_S \tag{2.24}$$

The reverse recovery loss, however, is a bit more complex to calculate. The source of the reverse recovery loss can be seen in figure 2.9. According to the figure when the polarity changes there will be a high current spike flowing into the diode at the same time as there also exists a large voltage drop. This results in very high losses during a short time period. The cause for this reverse recovery is that there are a large amount of excess carriers stored inside the diode, mainly in the drift region of the diode, which needs to be swept away before the diode starts blocking current [15]. To calculate the losses the charge Q_f needs to be estimated, the manufacturer does not usually provide this, instead they provide the total reverse recovery charge, Q_{rr}. The charge Q_f can then be estimated according to equation 2.25 [15]. The reverse recovery takes place once every switching period and can be calculated according to equation 2.26.

$$Q_f \approx \frac{1}{S+1} \cdot Q_{rr} \quad \texttt{where} \quad S = \frac{T_{rr1}}{T_{rr2}} \tag{2.25}$$

$$P_{D,rr} = V_{DC} \cdot Q_f \cdot f_{sw} \tag{2.26}$$

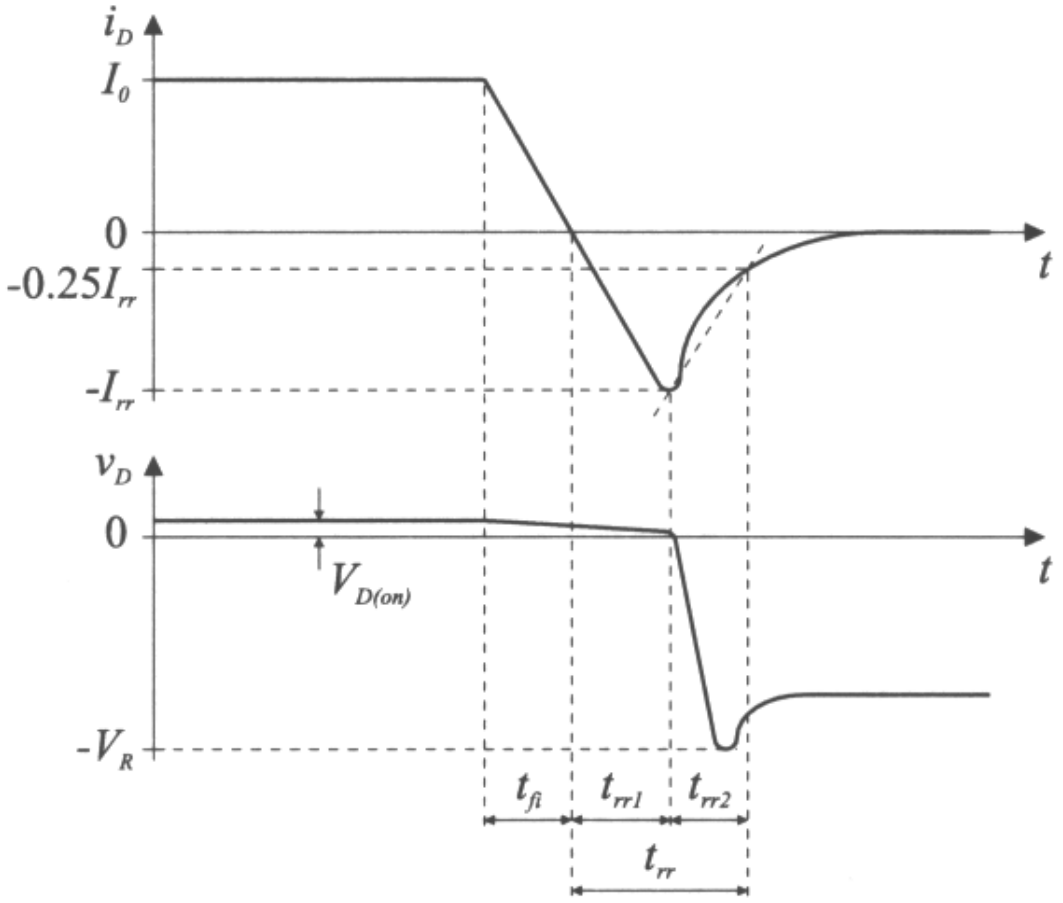


Figure 2.9: Diagram displaying voltage drop over and current through a diode at turn off. [15]

However sometimes the manufacturer does provide the turn-off energy not only for the transistor but for the diode as well, in this case the turn-off losses, which includes the reverse recovery losses, can be calculated in the same way as for the transistor, see equation 2.27 [15].

$$P_{D,off} = E_{D,off}(T_{sw}) \cdot f_{sw} \quad , \quad E_{D,off}(T_{sw}) = \frac{E_{off,n}}{V_{DC,n} \cdot I_{0,n}} \cdot V_{DC} \cdot I_0 \qquad (2.27)$$

Chapter 3

Wind Power Unit

The following chapter will describe how the model of the wind turbine was developed and modeled, it will also briefly describe how the generator and inverter model from the Spot library is modeled. It will also describe both how the wind power theory, see section 2.2, can be used in order to obtain maximum efficiency from the wind power units and how the overall wind turbine control is designed and implemented. Also the control of the generator and inverter is described. An overview of the system can be seen in figure 3.1. Finally some simulation cases are run in order to test how the system performs in different control tasks and situations.

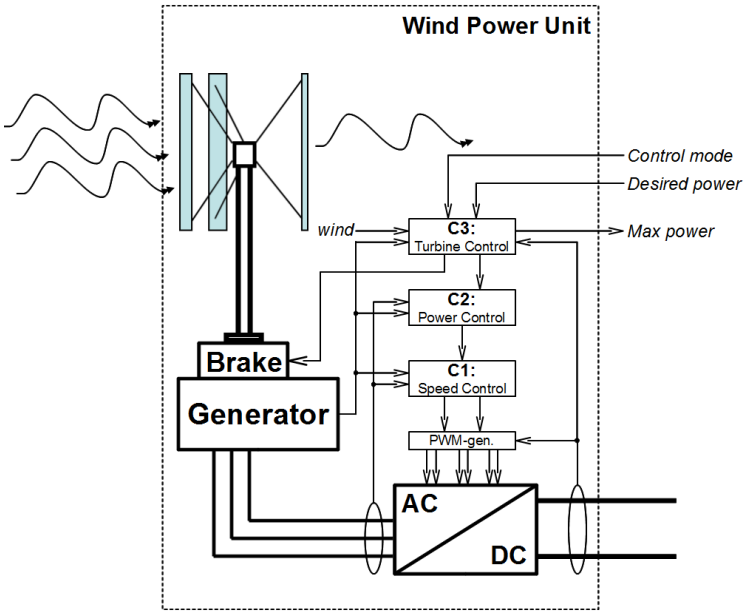


Figure 3.1: Schematic overview of a wind power unit.

3.1 Objectives

One of the objectives in this project is to model and control one wind power unit, a wind power unit is here defined according to figure 1.1, as one wind power turbine with an internal or external inverter connected to a DC-grid. A more detailed figure with a proposed control structure is presented in figure 3.1. The DC-grid can either be a shared DC-grid for several units or directly connecting the wind power unit with the grid connection, see chapter 4. The objective in this chapter is to model and control the wind power unit. The control system which is to be designed should be able to achieve a number of different control tasks listed below.

- Maximize the output power.
- Output a desired power, or if the wind is too low for achieving the desired power the control should maximize the power output.
- Performing a control where a specific percent of the maximum power is actuated, so called delta control.
- Control the DC-level.
- Shut down when ordered to.
- Automatically shut down when the wind speed is too high or low.

The wind power unit which is to be modeled and controlled is configured as a turbine connected via a shaft to a brake and generator. The generator is in turn connected to an inverter which connects the wind power unit to the DC-grid, as proposed in figure 3.1. In the figure a potential control structure is also proposed.

3.1.1 Test Cases

In order to test the models and control algorithms which will be developed in this chapter a number of simulation cases should be evaluated. The test cases are constructed in such a way so that they test different realistic control scenarios and/or a specific control algorithm. Since the DC-level control requires the whole system to be in operation the tests of the DC-control will be performed in chapter 5. The different test cases are listed below.

1. Test operating at max power from the wind power unit using the sensorless MPPT, see section 3.3.2.
2. Test operating at max power from the wind power unit using the MPPT, see section 3.3.3.
3. Try to operate at nominal power from the wind power unit using the power controller, see section 3.4.2, with too low wind speed for nominal power.
4. Test operating at nominal power from the wind power unit using the power controller with sufficient wind speed.
5. Test tracking a power output reference with a varying wind speed, in the end order shut down of the unit.
6. Test operating in delta control.
7. Test a "total case" where the unit should operate at nominal power while the wind speed varies between too low to allow operation, to high to allow operation, to low for reaching nominal power and high enough for reaching nominal power.

3.2 Modeling

The main tool used for the modeling in this project was the Modelica tool Dymola.

3.2.1 Dymola

Dymola is developed by Dassault Systèmes and is a powerful Modelica tool which supports visual animation and a drag and drop graphical modeling interface. With Dymola it is possible to build and simulate advanced multi domain models using different model libraries and/or to create your own equation based models.

3.2.1.1 Modelica

Modelica is a multi domain modeling language that is developed by the non-profit organization Modelica Association since 1996 which consists of members from Europe, U.S.A and Canada [16]. Modelica is an object-oriented equation-based modeling language developed specifically for modeling of physical systems containing mechanical, electrical, hydraulic, thermal, control or process-oriented subparts. The models in Modelica can be described by differential, algebraic and discrete equations. Modelica is a free language for which there exist many different simulation tools, both free and commercial, such as Dymola.

3.2.2 Wind Turbine

The model of the wind turbine was chosen to be done according to the equations presented in section 2.2. The idea is that the power extracted from the wind can be described according to equation 3.1 where ρ is the air's density, A the area swept by the rotor, v the wind speed and C_p an efficiency coefficient.

$$P = \frac{1}{2} \cdot \rho \cdot A \cdot \mathrm{w}^3 \cdot C_p \tag{3.1}$$

The area swept by a vertical wind power turbine is expressed as the rotor diameter times the rotor length. The efficiency coefficient, C_p, can be calculated according to equation 2.2 to 2.4. The mechanical torque T_ω can be obtained by dividing the power absorbed, P, with the rotational speed of the turbine, ω_T. However, phenomena such as tower shadow may interfere and may cause oscillation to the produced torque. Every time a rotor blade passes behind the tower a shadowing effect occurs, briefly reducing the produced torque, see figure 3.2. This since the blade shadowed not will be able to catch the same amount of wind. The tower shadow effect can be included in the wind turbine model by adding a negative torque component every time a blade passes behind the tower, as described in [7]. The torque component T_{ts} is described by equation 3.3. Where Ψ is the angle of the blades relative to the tower, Ψ_T is half of the circle sector when the blade is considered to be behind the tower and t_{ts} is considered to be a design parameter.

$$T_\omega = \frac{P}{\omega_T} \tag{3.2}$$

$$T_{ts} = -t_{ts} \cdot \cos(\Psi), -\Psi_T \leq \Psi \leq \Psi_T \tag{3.3}$$

Apart from modeling the wind turbine as an energy producing unit, catching wind and producing torque, modeling of the actual mechanical properties also needs to be done in order to get a good overall model. The wind turbine was modeled to be directly coupled to a shaft and via a brake to the permanent magnet generator. The shaft was modeled as an inertia, containing both the rotor's inertia and the actual shaft's inertia, coupled via a model of coulomb friction in bearings, in order to simulate losses in the shaft, to the brake. For

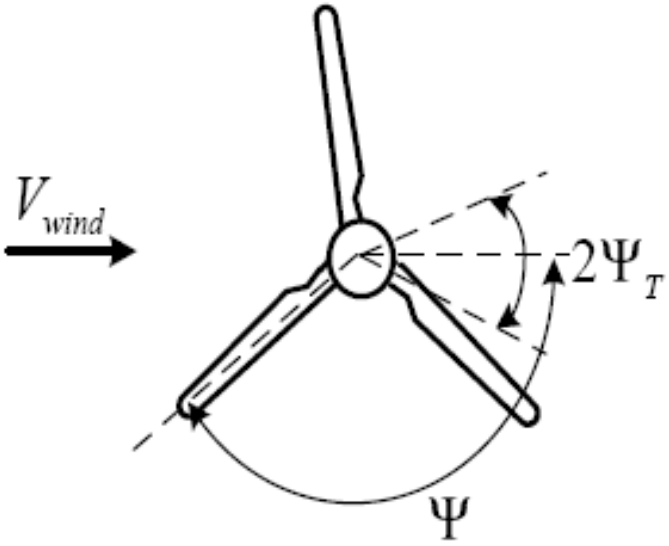


Figure 3.2: The phenomena of tower shadow.[7]

modeling of the inertia, bearing friction and brake components from Modelica's standard library was used.

Inertia

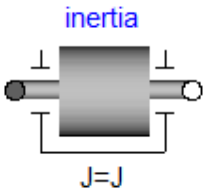


Figure 3.3: Icon for Modelica's standard library inertia.

The model for an inertia is very simple, the force-equation of an inertia is described by equation 3.4. Where J is the inertia, a the inertia's acceleration and τ the total rotational torque acting on the inertia.

$$\tau = J \cdot a \tag{3.4}$$

ω	τ
0	0
1	2
2	5
3	8

Table 3.1: Example of input table to Modelica's bearing friction model.

Bearing Friction

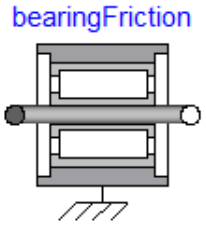


Figure 3.4: Icon for Modelica's standard library bearing friction.

The Modelica standard library model for bearing friction model the Coulomb friction in bearings. When the rotational speed is not zero a frictional torque is acting on the flange based upon a table provided by the user, see example table 3.1. The magnitude of the frictional torque is linearly extrapolated from the table provided. If the rotational speed is higher than the highest one provided by the table, the frictional torque is calculated by a linear interpolation of the last two table values. The frictional torque is assumed to be the same in both rotational directions. If the rotational speed is zero, the flange become stuck, the frictional torque is then calculated from a torque balance in order to keep the absolute acceleration to zero. The flange does not begin to rotate again until the driving torque is higher than a certain threshold value, the maximum static friction torque, calculated by equation 3.5.

$$\tau_{MaxStat} = peak \cdot \tau_{Sliding}(\omega = 0), (peak \geq 1) \tag{3.5}$$

Brake

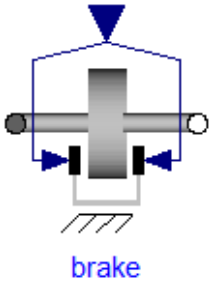


Figure 3.5: Icon for Modelica's standard library brake.

The Modelica standard library model of a brake, model a component where a frictional torque is acting between the housing and flange and a controlled normal force is pressing the flange to the housing in order to increase frictional torque. When the absolute rotational speed is not zero the frictional torque is calculated according to equation 3.6, depending on the normal force, $f_n, (0 \leq f_n \leq 1)$, a velocity dependent friction coefficient, $\mu(\omega)$, and a geometry constant, c_{geo}, which takes into account the geometry of the device.

$$\tau_{frictional} = c_{geo} \cdot \mu(\omega) \cdot f_n \tag{3.6}$$

When the absolute velocity of the flange is zero the component behaves in the same way as the bearing friction model, and gets stuck. The frictional torque is then calculated according to a torque balance in order to achieve an absolute acceleration of zero. The flange is stuck until the driving torque exceeds a threshold value calculated as the frictional torque at zero velocity times a constant, *peak*, see equation 3.7.

$$\tau_{FrictionalMax} = peak \cdot c_{geo} \cdot \mu(\omega = 0) \cdot f_n, (peak \geq 1) \tag{3.7}$$

3.2.3 Wind Model

Wind is moving air and the wind turbine uses the kinetic energy of the wind to produce electrical power. The produced power is tightly linked to the current wind speed. The wind changes both during the day and the seasons. In order capture these changes and to simulate the real wind conditions, a wind model consisting of three components is used in this project. The three components are:

- A base component, w_{base}
- A gust component, w_{gust}
- A noise component, w_{noise}

The three components are summarized to $\mathrm{w} = \mathrm{w}_{base} + \mathrm{w}_{gust} + \mathrm{w}_{noise}$. The base component is always present, it may be constant, a ramp signal or have any other form. The gust component appears randomly during time and the noise component is modeled as white noise.

3.2.4 Power Electronics

Since the system is designed using back to back full inverters the modeling of the power electronics becomes essential. Spot was chosen to be the main tool used for modeling the power electronics since it was well suited for the task.

3.2.4.1 Spot

The SPOT package is a Modelica library used for modeling of power electronics and can be used in both steady state and transient mode for the simulations

and initializations. SPOT provides components for modeling in AC three phase systems, AC one phase systems and DC systems. The AC three phase systems can be represented in the abc-, dqo- and rst-frame. Especially modeling in the dqo-/rst- reference frame provides relatively quick simulations, compared to simulations in the abc-reference frame. SPOT was originally written by H.J. Wiesmann and is currently under development by Modelon AB.

Reference Frames

One of Spot's key strength comes from the fact that it can utilize different reference frames when modeling power electronics. When dealing with a symmetrical three phase system there will always be a redundancy which is not necessary to simulate. By instead transferring the system to another reference plane the redundancy can be removed and the simulation time drastically decreased. The reference frame mainly used during this project was the dq0-reference frame which can be obtained either directly through a transformation from the abc[1]-reference frame to the dq0-reference frame, or via the $\alpha\beta\gamma$-reference frame, which is more pedagogical.

When transferring the system from the abc-frame to the $\alpha\beta\gamma$-frame phase b and c are phase shifted by 120 and 240 degrees and then summarized in a complex reference frame where the complex axis represents the β-component and the real axes represents the α-component, see figure 3.6, and appendix A.1 for mathematical calculations. As can be seen in figure 3.6 the symmetrical three phase system in the abc-reference frame is transferred into a rotating vector with a constant amplitude and rotational speed in the $\alpha\beta\gamma$-reference frame. Only by doing this transformation analysis, as well as simulation, of the system is greatly simplified. However the system can be simplified even further. Since the system always rotates with a constant rotational speed in the $\alpha\beta\gamma$-frame proportional to the system frequency, $\omega = 2\pi \cdot f$, the system can be further simplified by multiplying the $\alpha\beta\gamma$-frame with a vector rotating with the same speed. This results in a rotating reference frame in which the three phase symmetrical system is described as a point, the new reference frame is called the dq0-reference frame. A comparison between the simulated voltage levels in the two different reference frames is shown in figure 3.7. Both analysis and simulation in the dq0-reference frame is much simpler than in the physical abc-reference frame. Instead of describing three sinusoidal signals with a rotational speed of $\omega = 2\pi \cdot f$ the three signals which makes a relatively small change is generated.

The change in reference frame from abc- to dq0- in Spot is done using the Park transform which transfers the system directly from the abc- to the dq0-reference frame. When transferring from the abc- to the dq0-reference frame the voltage or current vector in the abc-reference frame is multiplied with the transformation matrix P, see equation 3.8, and when going from the dq0-frame back to the abc-frame the voltage or current vector is multiplied

[1]The abc-reference frame is the actual physical system.

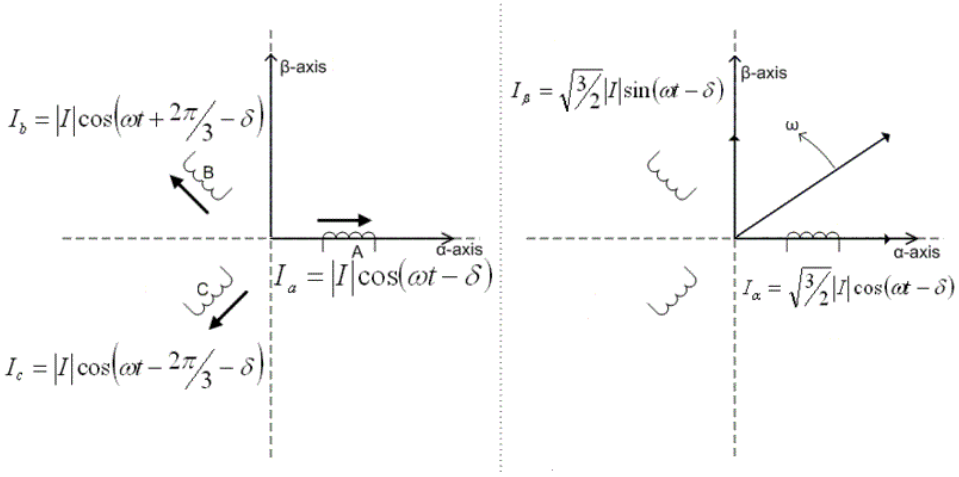
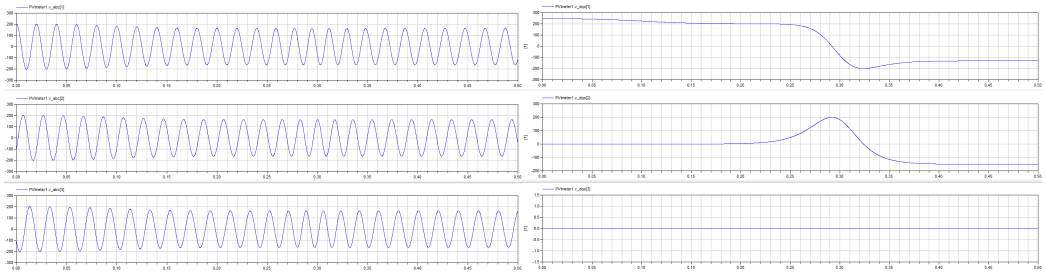


Figure 3.6: Geometrical representation for transformation from the abc-reference frame to the $\alpha\beta\gamma$-reference frame.



(a) Voltage levels in the abc-frame **(b)** The same voltage levels in the dq0-frame

Figure 3.7: Comparison of voltage levels in the abc- and dq0-reference frames.

with the inverse transformation matrix $P^{-1} = P^T$, see equation 3.9.

$$P = \sqrt{\frac{2}{3}} \begin{bmatrix} \cos(\theta) & \cos(\theta - \frac{2\pi}{3}) & \cos(\theta + \frac{2\pi}{3}) \\ \sin(\theta) & \sin(\theta - \frac{2\pi}{3}) & \sin(\theta + \frac{2\pi}{3}) \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \tag{3.8}$$

$$P^{-1} = \sqrt{\frac{2}{3}} \begin{bmatrix} \cos(\theta) & \sin(\theta) & \frac{\sqrt{2}}{2} \\ \cos(\theta - \frac{2\pi}{3}) & \sin(\theta - \frac{2\pi}{3}) & \frac{\sqrt{2}}{2} \\ \cos(\theta + \frac{2\pi}{3}) & \sin(\theta + \frac{2\pi}{3}) & \frac{\sqrt{2}}{2} \end{bmatrix} \tag{3.9}$$

3.2.4.2 Components

In order to model the power electronics used in the wind power unit two components from the Spot library was used, the Inverter and the Generator.

Inverter

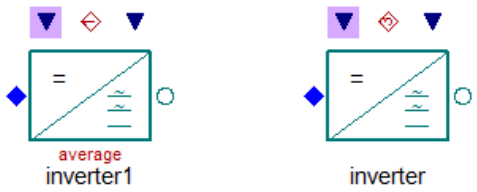


Figure 3.8: Icon for Spot's inverter model.

The Inverter model in Spot exists in two different versions. One version which incorporates switching and one which neglects the effects of switching. The model which incorporates switching is much closer to the reality, but is slowed down a great deal by the switching. The average inverter on the other hand neglects the effects of switching which makes it relatively fast. The main difference between the outputs from the two inverter models is that the output from the non switching model is a continuous signal, while the output from the switched model is a PWM signal, if not filtered. However the average output from the two models is the same. When incorporating switching in the models ripple effects in currents is introduced, which could affect the system, however these effects are generally quite small and can thus also often be neglected. They could however be interesting in some cases. The two models are easily exchangeable in Dymola so that long simulations can be run using the average model while short simulations, for example in order to study a special case, can be run using the switched model.

The average model of the inverter is in reality based upon a voltage source model with some key differences. The first, and most important, difference is that the u_phasor-input signal defines the AC-voltage output level in terms of the V_DC voltage. The actual mapping from AC- to DC-voltage is slightly different depending on which type of modulation is used. When sine modulation is chosen the relation between the AC- and DC-side is calculated according to

3.10, and when space-vector modulation is used the relationship is calculated according to equation 3.11.

$$|V\_AC| = u\_phasor\sqrt{\frac{3}{2}} \cdot \frac{V\_DC}{2} \tag{3.10}$$

$$|V\_AC| = u\_phasor\sqrt{\frac{2}{3}} \cdot V\_DC \tag{3.11}$$

The second difference is that the average inverter implements a loss model using data which can either be found directly in or derived from the data sheets of the inverter/transistors, see section 2.3.2.

The switched model, while it takes the exact same input signals as the average model, the relationship between the average AC- and DC- voltage is also essentially the same. However the actual contents of the model are completely exchanged. The switched inverter model consists of two separate parts, the modulation control and the actual inverter. The modulator model is the part of the model which controls the inverter transistors, just as done in reality. The output from the modulator block is a boolean vector of length six, one signal for each transistor. The modulator block is implemented in a couple of different modes, both synchronous and asynchronous, or variants, both sine and space-vector modulation. The second part, the inverter block, is implemented in three different ways, the first variant is a small simplification neglecting the anti-parallel diodes, which have the effect that the transistors cannot work in passive mode. The inverter is here modeled by a couple of short equations. Also the second implementation is an equation based implementation, however it does not neglect the anti-parallel diodes. The third and last implementation is a modular version, which utilizes three transistor models with anti-parallel diodes in order to model the inverter. All three of the models have in common that they operate in the abc-reference frame which is one of the reasons to why the switched models are slower than the average model.

In this project the average inverter was used as the main model, this since it provides much faster simulations. However the system was also tested using the switched model. Both of the Spot's models also provide modeling of the losses. This is done using information usually found in data sheets of the transistors, such as resistance when 'on', conductance when 'off', forward threshold voltage, switching loss at nominal voltage/current and temperature coefficients for thermal losses.

Generator

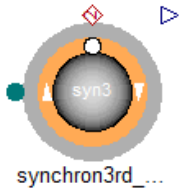


Figure 3.9: Icon for Spot's Permanent Magnet Synchronous Generator model.

In order to model the generator Spot's model of a permanent magnet synchronous motor, PMSM, was used. The Spot model uses an equation based approach which is quite advantageous since the interaction between electrical and magnetic energy as well as how voltage, current and electrical fields behave is well documented and relatively exact equations for this exists. The equations describing PMSM's is usually described in the xy-reference frame, which is closely related to the dq0- reference frame with the difference that the rotational speed of the reference frame is not necessarily constant. Instead of using a reference frame which rotates at a constant rotational speed relative to 50 or 60Hz, a reference frame which rotates with the same speed as the electrical speed of the machine is used. This is done in order to achieve a reference frame which is stationary from the rotor's perspective. The model in Spot uses equation 3.12 to describe the voltages and currents in the machine. In equation 3.12 u and i is the voltages and currents in the dq0/xy-frame, R_s is the rotor resistance, ψ_m the permanent magnetization of the rotor, L is the inductance and ω_r is the rotor's rotational speed.

$$\begin{aligned} u_{sd} &= -R_s i_{sd} + \frac{d}{dt}(\psi_m + L_{sd} i_{sd}) - \omega_r L_{sq} i_{sq} \\ u_{sq} &= -R_s i_{sq} + L_{sq}\frac{di_{sq}}{dt} + \omega_r(\psi_m + L_{sq} i_{sq}) \end{aligned} \tag{3.12}$$

The torque output from the PMSM can then be expressed according to equation 3.13 where τ is the output torque. As can be seen in the expression for the torque, as well as in equation 3.12, there are two different inductances involved, L_{sd} and L_{sq}, which can be used in order to obtain torque. These two inductances reflect the design of the rotor. A completely circular symmetrical rotor will, magnetically, have $L_{sd} = L_{sq}$ which results in that only i_{sq} will have an effect on the output torque. On the other hand an elongated rotor will have two completely different inductances which will enable control using both i_{sd} and i_{sq}. This kind of torque is called reluctance torque and originates from the fact that a magnet will try to rotate in order to line up with the surrounding magnetic field, compare with a compass. As described by equation 3.14 an unbalance in the inductances will invoke a change in the magnetization of the rotor proportional to the currents which can be used in order to produce torque.

$$\tau = \psi_m i_{sq} + (L_{sd} - L_{sq}) i_{sd} i_{sq} \tag{3.13}$$

$$\begin{aligned} \psi_{sd} &= \psi_m + L_{sd} i_{sd} \\ \psi_{sq} &= L_{sq} i_{sq} \end{aligned} \tag{3.14}$$

The model in Spot uses the same equations in both the abc- and dq0-reference frame model. The only difference being that the abc-model transfers the electrical system to the dq0-reference frame before the calculations are performed.

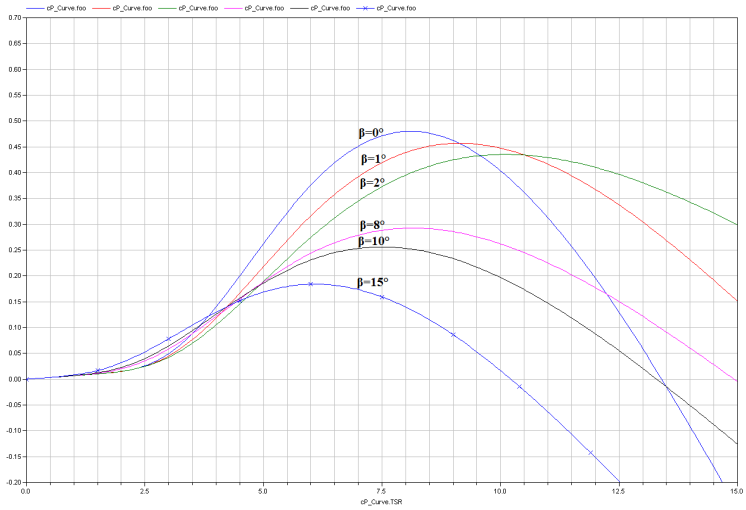


Figure 3.10: A diagram showing the general shape of a C_p-curve.

3.3 Optimizing Algorithms

The power extracted from a wind power unit can be described as the total kinetic effect of the air that passes through the turbine's wingspan multiplied with the efficiency coefficient C_p, see section 2.1. An example of the typical shape of the efficiency coefficient, C_p, described by equation 2.2 is shown in figure 3.10. In order to run the unit efficiently it is important to always be operating as close to the peak of the C_p-curve as possible. As can be seen in figure 3.10 the peek's position can be moved by changing the pitch angle. The control strategies proposed below is, however, designed for a system with a constant pitch angle. When assuming a constant pitch angle three strategies to achieve the optimal rotational speed of the rotor are presented below.

3.3.1 Known Wind Speed and C_p-Curve

If assuming that the C_p-curve is known it is possible to numerically calculate the optimal tip speed ratio, λ. When knowing the optimal tip speed ratio the optimal rotor speed can be calculated as a function depending on the wind speed using equation 2.4. In this way the optimal rotor speed reference can be algebraically calculated by measuring the current wind speed accurately.

3.3.2 Sensorless

When assuming that very little is known about the wind power unit and the power extracted from the unit is the only measured variable the algebraic strategy is no longer a viable option. Instead of searching for an algebraic solution the method proposed in [17] uses information about the direction and magnitude of the change in power extracted is affected by the change of the demanded rotational speed. The flow chart in figure 3.11 describes the algorithm, but the general idea is to figure out on which section of the C_p-curve the system currently is operating and take appropriate action. If the

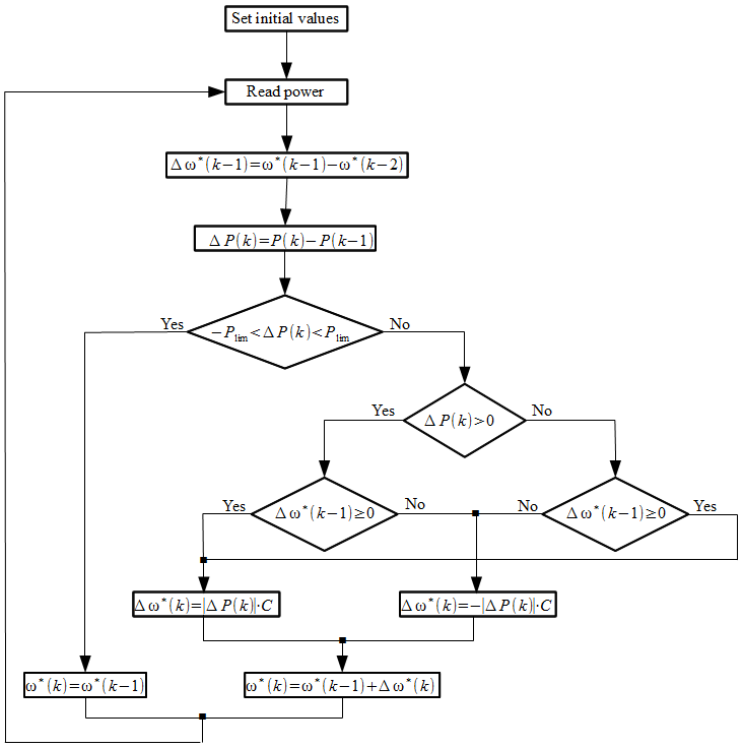


Figure 3.11: Flow chart showing the operation of the maximum power point tracking algorithm [17]

current operating point is to the left of the maximum point the command rotational speed should increase, if the current operating point is to the right of the maximum point the command rotational speed should decrease and if the current operating point is at the maximum point the speed reference should be kept constant. The operation is briefly summarized below.

- If the change in power is smaller than a predefined limit no action is taken.
- If the change in power is positive, $\Delta P(k) > 0$, and the change in rotational speed is positive or zero, $\Delta\omega(k-1) \geq 0$, then the speed reference is increased.
- If the change in power is positive, $\Delta P(k) > 0$, and the change in rotational speed is negative, $\Delta\omega(k-1) < 0$, then the speed reference is decreased.
- If the change in power is negative or zero, $\Delta P(k) \leq 0$, and the change in rotational speed is positive or zero, $\Delta\omega(k-1) \geq 0$, then the speed reference is increased.
- If the change in power is negative or zero, $\Delta P(k) \geq 0$, and the change in rotational speed is negative, $\Delta\omega(k-1) < 0$, then the speed reference is decreased.

The change in the rotational speed reference, $\Delta\omega(k)$, is always calculated as the change in power, $\Delta P(k)$, times a constant c. When tuned correctly the algorithm should increase the rotational speed reference with increasingly smaller steps until the optimal operational point is achieved.

3.3.3 Rotational Speed Known

The algorithm described in 3.3.2 can be improved by also measuring the rotational speed of the rotor. By doing this the risk that is imposed by assuming that the rotor speed is the same as the demanded speed is removed. What differs this algorithm from 3.3.2 is that the change in rotational speed is calculated using the measurements of the actual rotational speed of the rotor. This is done in order to achieve a more robust optimization of the rotational speed.

3.4 Control Design

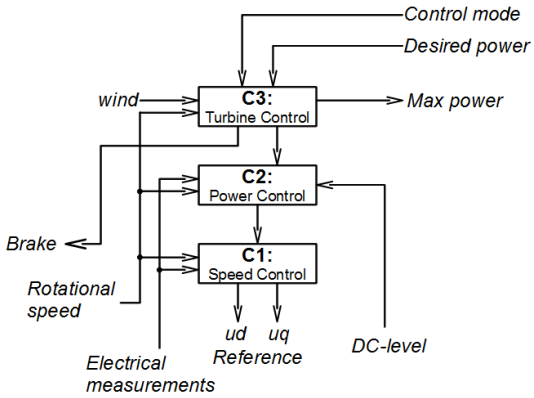


Figure 3.12: Overview of control structure for the wind power unit.

To effectively run a wind power unit there is a lot of different controls needed, both for controlling the rotor speed and for controlling the power electronics. The power unit is apart from this also required to be able to operate in a number of different control modes with different control goals. In order to achieve these goals a control structure for the wind power unit was designed according to figure 3.12. Some of the possible tasks that should be handled by the controller are listed below.

- DC-voltage level controller, needed for operation in island mode where the DC-voltage level is controlled by the wind power units.
- Power optimization/power tracking of the output from the wind power unit, this is needed in order to optimize the wind power unit's operation point and in order to control the unit's power output to a specific level in order to operate at nominal power, output a specific power or control the DC-voltage level.

- Speed control of the generator, needed in order to make the rotor follow the reference speed given from the power control.
- Current control for the generator, needed in order to control the generator.
- Automatically make decision of when it is desirable or safe to operate.

3.4.1 Turbine Control

The Turbine Control's task is to manage the wind power unit, which is to decide when the unit should start and stop as well as giving instruction as to in which mode the unit currently should be working in. The Turbine Control should communicate with both the Plant Control and the Power Controller, see section 3.4.2. The communication should be kept to a minimum and no actual control should be done by the Turbine Control and Plant Control. The controller's main task is to make decisions about when the unit could be in operation and provide information about the unit's current capacity to the Plant Control. In order to do this the Turbine Control needs information about the current wind speed as well as orders from the Plant Control. Apart from this the control also needs information about how fast the rotor is spinning in order to avoid using the brake at high speeds and instead do a soft deceleration using the generator.

In order for the Turbine Control to decide when the wind speed conditions are good enough for operation limits for high and low wind speeds are needed. A usual problem dealing with wind measurement data is the existence of turbulence and gusts. In order for an effective control the Turbine Control cannot, and should not, halt operation due to an erroneous measurement value or small gust. To avoid this the wind speed measurements needs to be filtered before evaluated. Care also needs to be taken when resuming operation after bad wind condition so that the operation is not halted immediately after resuming operation.

By using the current wind speed the wind power unit's maximum potential power output can be calculated, according to equation 3.15, where $C_{p,max}$ is an estimation of the unit's maximum efficiency, ρ is the air's density, A the area swept by the rotor and v the current wind speed. The information about the unit's current maximum potential power output should then be communicated to the Plant Control. The maximum potential power output is also used when operating in delta-control. The delta-control mode is a mode which is commonly used when the wind power unit should be operating at as high power output as possible and at the same time be ready to quickly change mode, see section 4.2.1. When in delta-control the power output should be a constant percentage below its maximum potential power output.

$$P_{max} = \frac{1}{2}\rho A v^3 \cdot C_{p,max} \tag{3.15}$$

3.4.2 Power Controller

In order to control the power output from the wind power unit the Power Controller was developed. By controlling the power output from the wind power unit a number of different control modes can be achieved. The Power Controller is designed in two different parts. The first part, here named Power Reference, is to set an appropriate power reference to achieve the requested control goal and the second part, here named Power Control, is to control the power output by controlling the rotational speed of the rotor.

3.4.2.1 Power Reference

The power reference for the Power Controller is set in order to achieve one of the different control modes supported by the system. The different control modes are listed below.

- SHTDWN - Shut Down Mode, the controller shall bring the rotor to a stop.
- NOMEFF - Nominal Power Mode, the controller shall operate the wind power unit at nominal power.
- DCCONTR - DC-Level Control Mode, the controller shall control the DC-Voltage level to its nominal value.
- POWTRAC - Power Tracking Mode, the controller shall operate the wind power unit at a specific power, this control mode can also be used for delta-control.

Shut Down

When the shut down order is received the controller should bring the rotor to a stop in a safe and controlled way. The controller could potentially do this by simply applying the break, however in order to avoid unnecessary use of the break. The main deceleration should be performed by the generator. In order to achieve this the power reference is set to zero which will force the Power Control to decelerate the rotor to a minimum speed. When the rotor speed is low, i.e. below a predefined threshold, the brake should be applied stopping the rotor in a controlled fashion.

Nominal Power

When the wind power unit is ordered to operate at nominal power the power reference is simply set to the nominal power of the wind power unit. Nominal power is in practice equal to maximum power since operating at a power level over the rated power of the equipment will cause the generator and power electronics to operate at a higher power level than it was designed for, which will expose the system to a high risk of failure.

DC-Level Control

When the DC-level control mode is ordered the wind power unit's control task is to control the DC-voltage level to its nominal value. In order to achieve this a desired power is passed down from the Plant Control through the Turbine Control. This desired power is related to the power currently consumed by the grid, see section 5.3.2. The desired power received from the Plant Control is an approximation of the power output the wind power unit shall output in order for the DC-level to be inert, i.e. for the power generation to be equal to the power consumption. The desired power is used as an offset in the DC-control, and depending on the error in the DC-voltage level the power reference is set to be higher or lower than the desired power. When the error in the DC-level is low, the power reference is controlled by a simple P-controller, see equation 3.16. When the DC-level error is a bit larger the power reference is set to be significantly higher or lower than the desired power and when the DC-level error is big the power reference is set to nominal power or zero.

$$P_{ref} = P_{desired} + k \cdot V_{DC,error} \tag{3.16}$$

Power Tracking

The power tracking mode is used to achieve two different control goals, both the power tracking control and the delta-control. In both cases the Power Controller receives a desired power reference from the Turbine Control, and in both cases the power reference should be set to the desired power reference.

3.4.2.2 Power Control

The Power Control part of the Power Controller's task is to set a rotational speed that outputs the power reference set by the power reference part of the Power Controller. As long as the wind speed is high enough for achieving the desired power output the task is quite trivial and easily achieved by a simple PI-controller. However when the wind speed is to low and the desired power output cannot be achieved the controller should do as good as possible. In this case the controller should maximize the power output from the unit. In order to achieve this a control algorithm was developed based on the optimizing algorithms described in section 3.3. The algorithm was developed in two main stages the first using mechanical power measurements and the second using electrical power measurements.

Using Mechanical Power Measurements

When using mechanical measurements the designed algorithm is very similar to the maximum power point algorithm described in section 3.3.3. The algorithm is described by the flow chart in figure 3.13.

The control algorithm starts by setting initial conditions and reads new measurements. It then tries to figure out whether the current operating point is on the left or right side of the optimal operating point. If the current

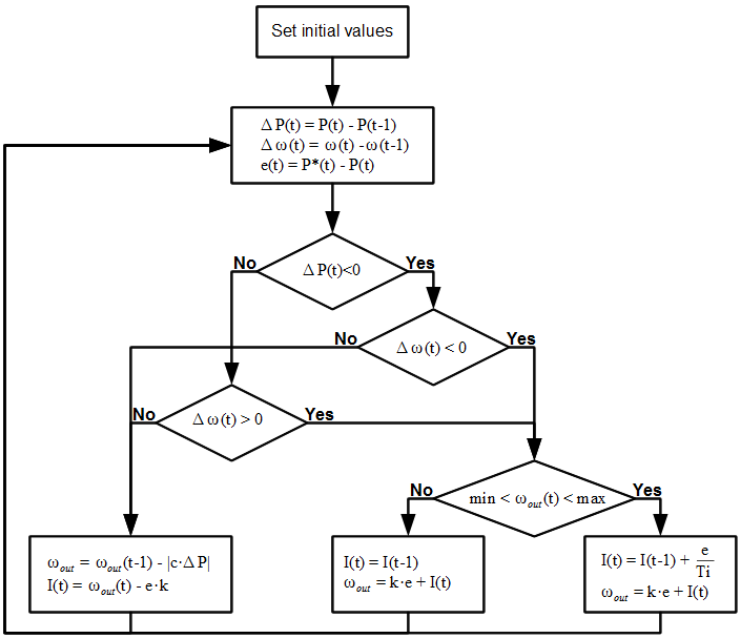


Figure 3.13: Flow chart over the power controller using mechanical power measurements.

operating point is considered to be on the left side of the optimal point the control output is calculated by a PI-controller with a simple anti-windup. If it is considered to be on the right side of the optimal point the control output is calculated as $\omega_{out}(t) = \omega_{out}(t-1) - |c \cdot \Delta P|$ where c is a control constant and ΔP is the change in power. Also the integral for the PI-controller is updated in order to achieve a bumpless control switch.

Using Electrical Power Measurements

If the power measurements is not taken from the rotating shaft but instead is calculated from the electrical voltages and currents as $P = u_d i_d + u_q i_q$ the algorithm developed is more advanced. The electrical power output is zero unless the generator is generating any current, i.e. unless the generator is applying a negative torque on the rotor. This makes the previous control structure ineffective, unless modifications are done. This since the power readings when accelerating and decelerating the rotor will be misleading due to the fact that the power output is greater during deceleration, and lower during acceleration. To cope with this the control algorithm was modified so that after a control output is set the control waits for the inner control algorithms to achieve the desired rotational speed. When the desired rotational speed has been achieved the algorithm performs a new power reading and, if necessary, calculates a new output reference.

A flow chart over the modified control algorithm, along with a couple of other improvements, is shown in figures 3.14 and 3.15. Apart from this the modified control algorithm works roughly the same as the one using mechanical power measurements. It starts by making some initializations and reads

the new measurements, it then checks whether the previous control output has been achieved, if so it updates the reference, if necessary. Below follows a detailed description of the modified control algorithm. The list numbers references the circled numbers in figure 3.14.

1. The algorithm starts by setting some initial values.
2. The algorithm reads new values for the electrical power, p_e, rotational speed, ω, and the reference power level p_ref.
3. The algorithm checks whether the reference power has changed or not, if so a flag signaling that the next update should be done by the PI-controller is set to **True**.
4. Since it is the electrical power that is measured the generator has to hold the rotor at a specific speed in order to make a power measurement. If the rotor speed has not yet reached its last set point the controller continues to the middle.
5. If the flag indicating that the reference power has changed is set to **True** the algorithm will set it to **False** and continue to the PI-controller, left.
6. If the flag is not set a check is made to see if the power output is larger than the reference value, if so the algorithm continues to the PI-controller, left.
7. If the current power output is smaller than the reference value a check is made to see whether the change in power is smaller or bigger than a predefined limit, if not the algorithm continues to the middle.
8. If the power change is larger than the predefined limit a check whether the change in rotational speed is larger than a predefined limit is done, if not the controller continues to the middle.
9. If both the change in power and change in rotational speed was big enough the algorithm continues to either left or right.

Left[2]

By continuing to "the left" the controller is signaling that it either considers the current operating point to be on the left side of the optimal point on the C_p-curve or that the current state has changed in a way that makes it favorable to use a PI-controller to set the new reference. The PI-controller first checks whether the control-signal is saturated or not, if the control signal is saturated the integrator is not updated. The controller then continues by calculating the new output and stores it in a temporary variable which is used in order to calculate how large the control step is. The step is then limited

[2]See figure 3.15

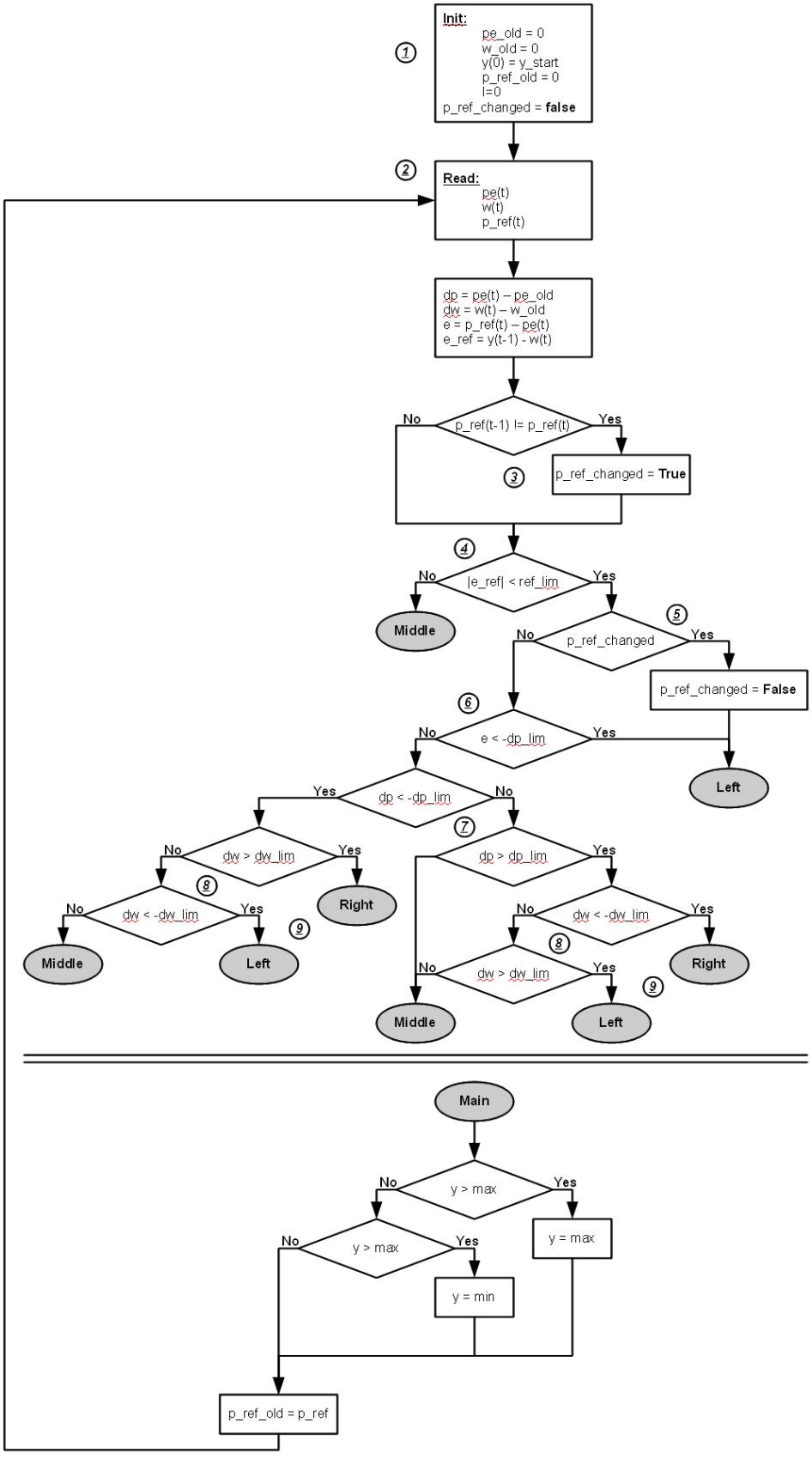


Figure 3.14: Flow chart over the power controller.

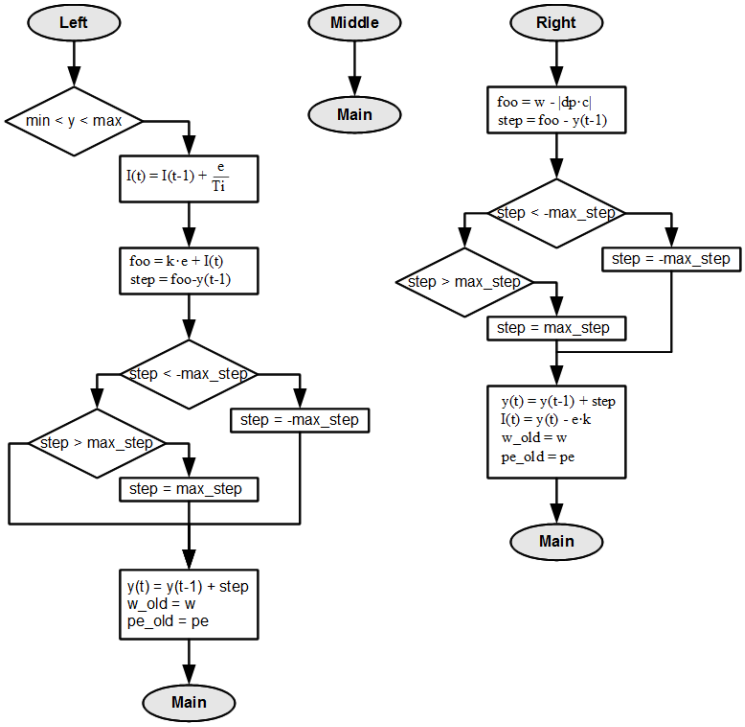


Figure 3.15: Flow chart over the power controller, continued.

by a predefined limit and added to the previous control signal. This is done so that a sudden spike in the measurement signal not should drive the rotor speed to standstill or maximum speed. When the step has been added w_old and p_old is updated.

Middle[3]

By continuing to "the middle" the controller is signaling that it either considers the current operating point to be at its optimal value or that the current state has not changed enough to make a new control output valid. In short the controller does nothing except making sure that ω_old and p_old not is updated.

Right[4]

By continuing to "the right" the controller is signaling that it considers the current operating point to be on the right side of the optimal operating point. When the controller continues to the right the new output value is calculated. This is done by first calculating the new output as $foo = \omega - |dp \cdot c|$ where ω is the current rotational speed of the rotor, dp is the change in power, c a control constant and foo a temporary variable. foo is used in order to calculate the change, step, from the last output that should be done, the step is then limited to the maximum allowed step size of the controller and then

[3]See figure 3.15
[4]See figure 3.15

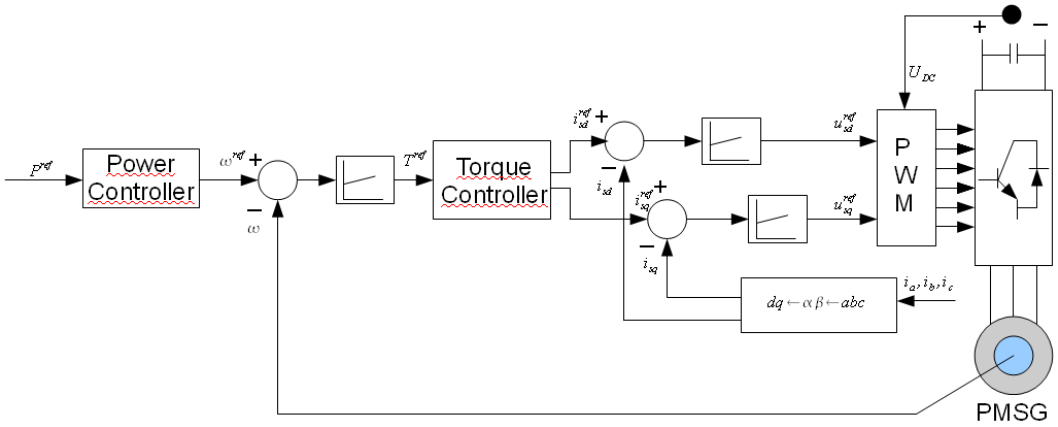


Figure 3.16: Block diagram over the control of the generator

added to the previous control signal to generate the new reference. After this is done the integral of the PI-controller needs to be updated in order to achieve a bumpless control switch. The new integral value is calculated so that the output of the PI-controller would be the same as the current output. After this ω_old and p_old is also updated.

3.4.3 Speed Control

The overall control structure for controlling the generator, as can be seen in figure 3.16, consist of the Power Controller followed by a cluster of controllers called the Speed Controller. The Speed Controller consists of in total two PI-controllers and one PID-controller. The PID-controller is controlling the rotational speed of the rotor and the two PI-controllers are controlling the currents into the generator.

The Speed Controller gets its reference value from the Power Controller and uses a PID-controller in order to control the rotational speed. The PID-controller outputs a torque reference which is passed to the Torque Control which calculates the appropriate current references in order to achieve the desired torque.

3.4.3.1 Torque Control

The task of the torque control is to calculate the currents needed for achieving the desired torque. There exist a couple of different torque control strategies, for example *Constant Torque Angle Control* and *Maximum Torque per Ampere Control*, in the later of the two the i_{sd} and i_{sq} currents are optimized in order to get as high torque as possible while keeping $|I|$ as low as possible, see equation 3.17. In *Constant Torque Angle Control* the torque angle is kept constant, usually at 90 °. This allows for a relatively simple control where i_{sd} is kept at zero at all times which reduces the torque equation to a linear dependency between the torque and i_{sq} current, see equation 3.18. Which control strategy to choose partly depends on the design of the rotor, whether if there is a big difference in L_{sd} and L_{sq} or if they are approximately the same, and partly the

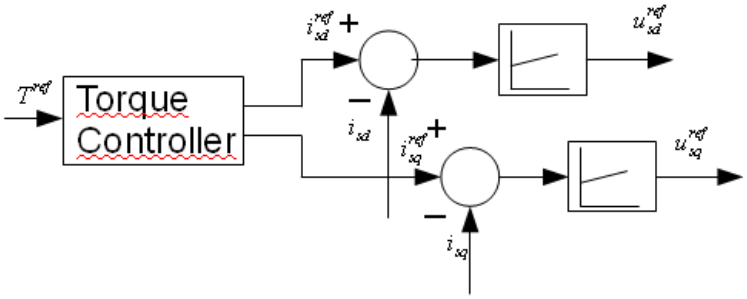


Figure 3.17: Structure of the control loops for the d- and q-currents

control goal. For this project the *Constant Torque Angle Control* was chosen, mainly due to its simplicity.

$$\begin{aligned} \tfrac{\tau}{pp} &= \psi_m i_{sq} + (L_{sd} - L_{sq}) i_{sd} i_{sq} \\ |I| &= \tfrac{1}{2}\sqrt{i_{sd}^2 + i_{sq}^2} \end{aligned} \tag{3.17}$$

$$\frac{\tau}{pp} = \psi_m i_{sq} \tag{3.18}$$

Since the *Constant Torque Angle Control* algorithm was chosen the current-calculations is done according to equation 3.19. The i_{sd} current is kept at a constant zero and i_{sq} is linearly proportional to the demanded torque.

$$\begin{aligned} i_{sd}^{ref} &= 0 \\ i_{sq}^{ref} &= \frac{\tau^{ref}}{\psi_m \cdot pp} \end{aligned} \tag{3.19}$$

3.4.3.2 Current Control

The current control loop is the inner most control of the Speed Control, and its overall structure can be seen in figure 3.17. The current controller get its reference values from the torque → current conversion and outputs voltage references for the inverter. The current controllers are essentially PI-controllers with compensation for the back emf, or so called PIE-controllers. The parameters used for the control is derived in [15], the parameters are derived in a way so that the controller will achieve its control goal in one sampling instant. However caution must be taken when using the controller, general practice is that half the proportional gain and double the integral time constant is a good starting point for tuning the controllers. The reason to this is that the actual machine parameters deviates a bit from what they should be, and in order to not make the control unstable it is suitable to tune down the controller, and, if necessary, increase them later. The control expression for the current control can be seen in equation 3.20.

$$u_{sd}^{ref}(k) = \left(\frac{L_{sd}}{T_s} + \frac{R_s}{2}\right)\left((i_d^{ref}(k) - i_d(k)) + \frac{T_s}{(\frac{L_{sd}}{R_s} + \frac{T_s}{2})}\sum_{n=0}^{n=k-1}((i_d^{ref}(n) - i_d(n)\right) - \omega_r L_{sq} i_{sq}(k)$$

$$u_{sq}^{ref}(k) = \left(\frac{L_{sq}}{T_s} + \frac{R_s}{2}\right)\left((i_q^{ref}(k) - i_q(k)) + \frac{T_s}{(\frac{L_{sq}}{R_s} + \frac{T_s}{2})}\sum_{n=0}^{n=k-1}((i_q^{ref}(n) - i_q(n)\right) + \omega_r(\psi_m + L_{sq} i_{sd}(k))$$

(3.20)

3.5 Implementation

The implementation was decided to be done using two different programming languages, Modelica and C/C++, this since the Modelica language is very well suited for modeling of the wind power unit and power electronics and C/C++ is widely used when programming control units. The controllers could have been written in Modelica, however the aim of this project was partly to write the control structure in a way so that the same code could be used both in simulation as well as on the real plant.

3.5.1 Modelica

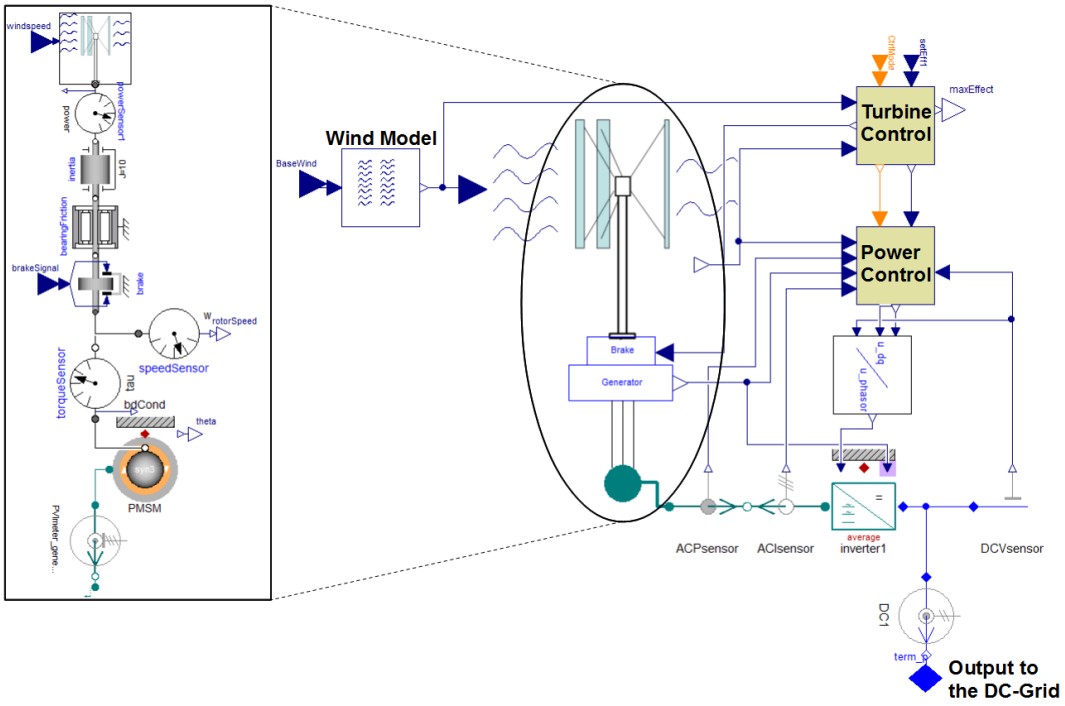


Figure 3.18: Overview of the Modelica model of a wind power unit with controllers.

The Modelica implementation was done using the Modelica tool Dymola, see section 3.2.1. Most of the implementation was done using Modelica's standard

library and Modelon's Spot library, however some models for the wind turbine was developed as well as a wind model.

An overview of the Modelica model over a wind power unit can be seen in figure 3.18, to the right is the top view, with inverter and controllers, and to the left is the contents of the actual unit, with generator, shaft and turbine model. The unit model consists of a model of the turbine connected to a shaft, modeled as inertia and a bearing friction. The shaft is connected via a brake to the permanent magnet synchronous generator, PMSG, the electrical output from the PMSG is connected to an inverter which performs an AC to DC conversion. Additionally a wind model was developed in order to model the wind in a realistic way. The inverter is controlled by the control-blocks on top of it, and by performing the AC/DC conversion the inverter controls the generator. As can be seen most models incorporated in our wind power unit model is from either Modelica's standard library or Modelon's Spot library. The models implemented in this project are the turbine model, wind model, the controllers and an interface between our control and the inverter.

3.5.1.1 Inverter Interface

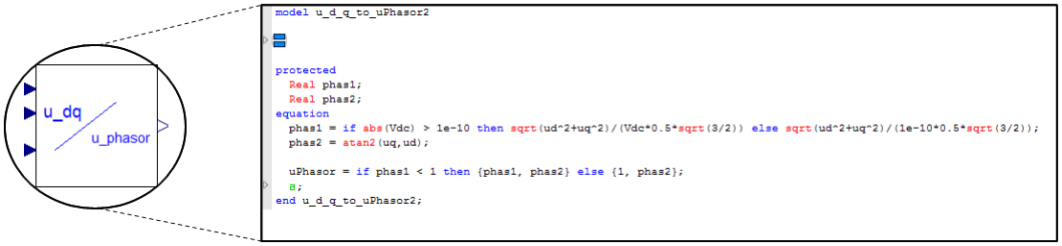


Figure 3.19: Icon for the interface between the control and inverter.

An interface between our control and the inverter in Spot is needed since the output from our control is the desired dq-voltages while the input to the inverter is the percentage of the DC-voltage desired on the AC-side, which can be calculated according to equation 3.21, and a phase angle which can be calculated according to equation 3.22.

$$|V_{AC}| = \sqrt{u_d^2 + u_q^2} = u \cdot \sqrt{\frac{3}{2}} \cdot \frac{V_{DC}}{2} \tag{3.21}$$

$$Phase = \arctan(\frac{u_q}{u_d}) \tag{3.22}$$

In order for the interface to do this conversion a measurement of the current DC-voltage level is done, and in order to avoid division by zero a limit for the minimum allowed $|V_{DC}|$ is set.

3.5.1.2 Wind Model

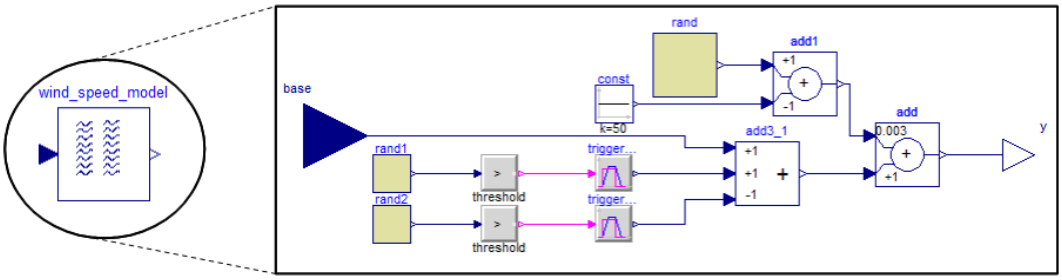


Figure 3.20: Overview of the wind model implemented in Modelica, with the icon to the left and the implementation to the right.

The wind model is implemented as a gust component plus a small noise, as described in section 3.2.3. In order to decide when the gust component should occur a random value is compared with a threshold value every ten seconds. If a random value is higher than the threshold a positive or negative gust component is added to the base wind speed in form of a trapezoid. The rand-block used to calculate the random value is generating a uniformly distributed value between 0 and 100. After the gust component a uniformly distributed noise is added to the base signal in the form of a random value scaled to ± 0.15m/s.

3.5.1.3 Wind Turbine

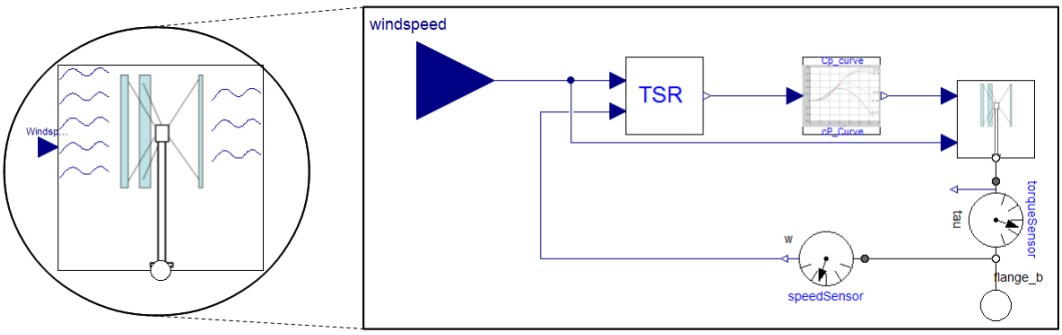


Figure 3.21: Overview of the Wind turbine model in Modelica, with the icon to the left and the implementation to the right.

The Implementation of the wind turbine was done in three different blocks. One calculating the tip speed ratio, TSR, according to equation 2.4, one calculating the C_p-value using equation 2.2 and 2.3 and one calculating the output torque according to equation 2.1, 3.2 and adjusted with the tower shadow effect described in equation 3.3.

All the three models were implemented according to the equations with some small variations in order to avoid numerical instability. In the model for the tip speed ratio a limit for the minimum allowed wind speed, $windspeed_{switch}$, in order to avoid division with very small values and/or by zero. When the

wind speed is below the allowed limit the tip speed ratio is instead calculated according to equation 3.23. In the model of the C_p-curve a lower limit for the C_p value was set to zero so that the efficiency never is below zero. In the model which calculates the generated torque acting on the shaft a lower limit for the rotational speed of the rotor was set, ω_{switch}, this in order to avoid division by zero. When the rotational speed is below the allowed limit the output torque is instead calculated according to equation 3.24.

$$\lambda = \frac{\omega_t R}{windspeed_{switch}} \tag{3.23}$$

$$T = \frac{P}{\omega_{switch}} \tag{3.24}$$

3.5.2 C/C++

There are two possibilities when trying to implement C/C++ objects into Dymola/Modelica. The first is using an External Static Library and the second is using the External Object function in Modelica. Both of the methods have its pros and cons. The External Static Library is disconnected from the modeling tool to a greater extent while the External Object is more interconnected with Modelica. In the end the decision was made to mainly use the External Static Library for implementation of the control structure. This mainly due to two different reasons. Reason one being that the goal was to be able to run the exact same code both in simulations and on the actual plant and for this reason we like to separate the control code and the models as much as possible. The second reason is that the External Object currently only supports code in FORTRAN 77 and C [18], while the static external library supports both C and C++. However the External Object approach was also investigated and tested.

3.5.2.1 External Static Library

The general idea with using an external static library is that Dymola will be using a compiled library containing all the different controls. In this way Dymola is completely disconnected from the control algorithms. This also allows for the control to be designed in a way that just as well could be compiled and downloaded to a control unit as it could be compiled to a static library and used for simulation purposes. In this way the exact same code that would be run on a real plant will be tested in a simulation environment. In order for this to work Dymola needs to know which functions that exists and how to invoke them. Also some tricks needs to be done in Dymola in order to "force" the optimization code for the equations not to remove the function-calls to the library.

In order to enable Dymola to use the external library an interface was written for every class which was needed to interact with Modelica. The interface was written using a header file and a source file, see listing 3.1 and 3.2. The header file is then put in Dymola's Source directory which enables Dymola to

use the functions. The Interface classes use an array with a certain size in order to allow multiple instances of the same object to be used at the same time. When a new instance of the object is created the *init*-function is called which initiates the object, if there is room in the array for one more, and returns the id for the newly created object. When using the object the other functions are then called using the id. If anything would go wrong in a function, or initiation, call the interface will return -1. Since Modelica currently only supports external functions defined in FORTRAN 77 and C [18] the interface classes is defined to be compiled as external C-code, see row five and thirteen in listing 3.1.

Listing 3.1: modelicaInter_ pi.h

```
1  #ifndef MODELICAINTER_PI_H
2  #define MODELICAINTER_PI_H
3
4  #ifdef __cplusplus
5  extern "C" {
6  #endif
7
8          extern int pi_init(double k,double Ti, double Ni, double yMax,double yMin, double
               samplePeriod,double iStart, double yStart);
9          extern int pi_update(int id,double error);
10         extern double pi_get(int id);
11
12 #ifdef __cplusplus
13 }
14 #endif
15 #endif
```

Listing 3.2: modelicaInter_pi.cpp

```
1
2  #include "modelicaInter_pi.h"
3  #include "pi.h"
4
5
6  const int max = 6;
7
8  static int id = 0;
9  static Pi vec[max];
10
11 int pi_init(double k, double Ti, double Ni, double yMax,double yMin, double samplePeriod,
       double iStart, double yStart) {
12         if (id< max) { // check so that array is not full
13                 vec[id++] = Pi();
14                 vec[id-1].init(k, Ti, Ni, yMax, yMin, samplePeriod,iStart, yStart);
15                 return id;
16         } else
17                 return -1;
18 }
19
20 int pi_update(int idIn, double error) {
21         if (idIn > 0 && idIn <=id) { //check so that idIn is valid
22                 vec[idIn-1].update(error);
23                 return idIn;
24         } else
25                 return -1;
26 }
27
28 double pi_get(int idIn) {
29         if (idIn > 0 && idIn <= id) { // check so that idIn is valid
30                 return vec[idIn-1].get();
31         } else
32                 return -1;
33 }
```

The interface classes can then be used as external functions in Dymola. This is done by creating a function with the same name as the interface function and list all the inputs followed by the outputs. It should also be declared that the function is implemented using external C-code. Apart from this the header file-name containing the function declaration and the library-name containing the executables should also be provided, see listing 3.3

Listing 3.3: pi_init.mo

```
within VerticalWindPark.Components.Control.ExternalLib.Functions;
function pi_init
  input Real k;
  input Real Ti;
  input Real Ni;
  input Real yMax;
  input Real yMin;
  input Real samplePeriod;
  input Real iStart;
  input Real yStart;
  output Integer id;
  external "C";
  annotation(Include="#include <modelicaInter_pi.h>",Library="control");

end pi_init;
```

When all the interface functions has been implemented a model using the external code can be written, see listing 3.4. The model starts by importing the functions needed and then continues by as normal defining parameters, inputs and outputs. Since all our control algorithms is sampled/discrete in nature the models are declared to extend the DiscreteBlock Interface in Modelica which provides some basic variables such as **initial()**, which returns **true** if it is the first sampling instant, and *sampleTrigger* which is true when a sample instant occurs.

Since Dymola/Modelica optimizes the system of equation as much as possible in order to accelerate the simulations it will disregard any functions and equations which it deems useless. This means that a function call without any return value such as **pi_update(...)** will be disregarded, since it does not affect the system. Dymola is completely unaware that this function call changes any states. In order to "force" Dymola to include the equation or function call some state needs to be effected by a return value, which is the reason to why both id1 and id2 is used in the code. When calling *id2* = **pi_update(...)** id2 is changed depending on what is returned by the update-function, thus the function call cannot be neglected.

There is also a protection implemented in order to stop the simulation if more than the allowed number of instances of the same object is exceeded or some other input violation is done. This protection is enabled by the fact that -1 is returned by the C-function when an input violation is detected.

Listing 3.4: pi_lib.mo

```
within VerticalWindPark.Components.Control.ExternalLib;
model pi_lib
  extends Modelica.Blocks.Interfaces.DiscreteBlock;

  import init_pi =
    VerticalWindPark.Components.Control.ExternalLib.Functions.
        pi_init;
  import update_pi =
    VerticalWindPark.Components.Control.ExternalLib.Functions.
        pi_update;
  import get_pi =
    VerticalWindPark.Components.Control.ExternalLib.Functions.
        pi_get;
  import Modelica.Utilities.Streams.error;

  parameter Real i_start=0 "Initial value of integrator value";
  parameter Real y_start=0 "Initial value of output signal";
  parameter Real P = 1 "Proportional gain";
  parameter Real Ti = 1 "Integrator time constant";
  parameter Real yMax = 100 "Maximum output";
  parameter Real yMin = -yMax "Minimum output";
  parameter Real k = 1 "Gain of the controller";
  parameter Real Ni = 1 "Ni*Ti is the time constant of antiwindup compensation";

  Modelica.Blocks.Interfaces.RealInput u_s;
  Modelica.Blocks.Interfaces.RealInput u_m;
  Modelica.Blocks.Interfaces.RealOutput y;

protected
```

```
30    Integer id1;
31    Integer id2;
32
33    Real err;
34    Real ySamp;
35
36  initial equation
37    y = y_start;
38  equation
39    when initial() then
40      id1 = init_pi(P, Ti, Ni, yMax, yMin, samplePeriod, i_start, y_start);
41    end when;
42
43    when sampleTrigger then
44      err = u_s-u_m;
45      id2 = update_pi(id1, err);
46      ySamp = get_pi(id2);
47    end when;
48
49    if id2 < 1 or id1 < 1 then
50      error("Error using 'Pi' in the external 'control' library, possibly the 'Pi' array is
          full!");
51    end if;
52
53    y = pre(ySamp);
54  end pi_lib;
```

3.5.2.2 External Object

The *External Object*-function is Modelica's way of achieving a function with an internal memory, that is an object, see [18]. This is done by letting the Modelica class directly extend from the predefined partial class *ExternalObject*, it shall also define exactly two functions, one called "*constructor*" and one called "*destructor*". The *constructor* will be called exactly once, just before the first usage of the object, and the *destructor* will be called exactly once after the last use of the object. The *constructor* should have exactly one output argument in which the *ExternalObject* is returned, and the *destructor* shall have no outputs. The *ExternalObject*-type is mapped in C as a **void**-pointer when used as an input or return value. The *constructor* and *destructor* is called automatically, it is illegal to call any of them explicitly. When created the *ExternalObject*'s internal memory can be operated upon by an external functions by using the *ExternalObject* as an input to the function[18], the general structure of how the *ExternalObject* is used can be seen in figure 3.22.

The actual implementation in Modelica can be seen in listing 3.5 and 3.6. In 3.5 the *ExternalObject* is defined with the required *constructor* and *destructor*, and in 3.6 the function which uses the internal memory of the *ExternalObject* is defined by using the *ExternalObject* as one input.

Listing 3.5: PI2.mo

```
1   within VerticalWindPark.Components.Control.External.Functions.PI2;
2   class PI2
3     extends ExternalObject;
4
5     function constructor
6       input Real P;
7       input Real Ti;
8       input Real Ni;
9       input Real yMax;
10      input Real yMin;
11      input Real samplePeriod;
12      input Real i_start;
13      output PI2 pi;
14      external "C" pi=  initPI2(P,Ti, Ni, yMax,yMin,samplePeriod,i_start);
15      annotation(Include="#include <PI2.c>", uses(Modelica(version="3.2")));
16
17    end constructor;
18
19    function destructor
```

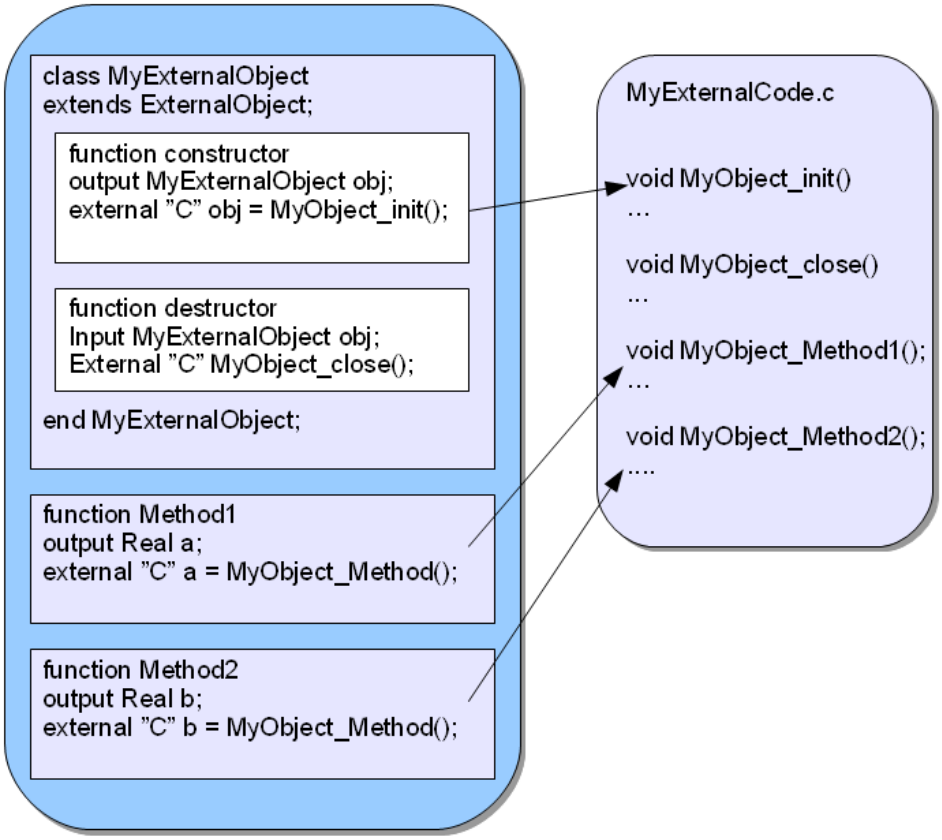


Figure 3.22: Schematic picture of how the external object is used.

```
20      input PI2 pi;
21      external "C" closePI2(pi);
22    end destructor;
23
24  end PI2;
```

Listing 3.6: updatePI2.mo

```
1  within VerticalWindPark.Components.Control.External.Functions.PI2;
2  function updatePI2
3
4    input VerticalWindPark.Components.Control.External.Functions.PI2.PI2
5                  pi;
6    input Real error;
7    output Real y;
8    external "C" y = updatePI2(pi, error);
9
10 end updatePI2;
```

The external object can then be used to implement the external PI-controller according to listing 3.7. As can be seen the initial values for the controller is entered when the external object is declared on row 16 which will later be used when the *constructor* is called. The same problem regarding the optimization of the equations done by Dymola that is present when using the external library, see section 3.5.2.1, is also present here. However this particular function was implemented to return the control signal directly when the *update*-function is called, if the controller where to be implemented as the example in the external library case, where nothing is returned by the *update*-function but instead needs to be fetched by a *get()*-function, then some kind of dummy variable would have to been utilized in order to force the update to be performed. Apart from this the actual model in Modelica looks almost identical to the External Library implementation, see listing 3.4.

Listing 3.7: PI2_external.mo

```
1  within VerticalWindPark.Components.Control.External;
2  model PI2_external "define a new PI-controller"
3  extends Modelica.Blocks.Interfaces.DiscreteBlock;
4    Modelica.Blocks.Interfaces.RealInput u_s;
5    Modelica.Blocks.Interfaces.RealInput u_m;
6    Modelica.Blocks.Interfaces.RealOutput y;
7
8    parameter Real i_start=0 "Initial value of integrator value";
9    parameter Real P = 1 "Proportional gain";
10   parameter Real Ti = 1 "Integrator time constant";
11   parameter Real yMax = 100 "Maximum output";
12   parameter Real yMin = -yMax "Minimum output";
13   parameter Real Ni = 1 "Ni*Ti is the time constant of antiwindup compensation";
14
15 protected
16   VerticalWindPark.Components.Control.External.Functions.PI2.PI2 pi =
       VerticalWindPark.Components.Control.External.Functions.PI2.PI2(P,Ti, Ni,yMax,yMin,
       samplePeriod, i_start);
17
18   Real error;
19   Real ySample;
20
21 equation
22   when sampleTrigger then
23     error = u_s - u_m;
24     ySample = VerticalWindPark.Components.Control.External.
25               Functions.PI2.updatePI2(pi, error);
26   end when;
27
28   y = pre(ySample);
29
30 end PI2_external;
```

The implementation of the PI-controller in C can be seen in listing 3.8, the constructor, *initPI2*, creates an instance of the PI-controller which is returned as a **void**-pointer. Since no structures that requires any specific memory release is used the *destructor* only needs to free memory used by the pi-controller using

free(). The implementation of *updatePI2()* starts by converting the **void**-pointer into a *PI2*-pointer and checks so that the conversion was successful. After this the implementation of the rest of the code is pretty straight forward.

Listing 3.8: PI2.c

```
typedef struct{
        double I;
        double yMax, yMin;
        double P, Ti;
        double samplePeriod;
        double Ni;
} PI2;

void* initPI2(double P, double Ti, double Ni, double yMax, double yMin, double
    samplePeriod, double iStart) {
        PI2* pi = malloc(sizeof(PI2));

        if (pi == NULL)
                ModelicaError("Not enough memory!");

        pi->P = P;
        pi->Ti = Ti;
        pi->Ni = Ni;
        pi->yMax = yMax;
        pi->yMin = yMin;
        pi->samplePeriod = samplePeriod;
        pi->I = iStart;
        return (void*) pi;
};

void closePI2(void* obj) {
        PI2* pi = (PI2*) obj;
        if (obj == NULL)
                return;
        free(pi);
};

double updatePI2(void* obj, double error) {
        PI2* pi = (PI2*) obj;
        double y, temp;

        if (pi == NULL)
                ModelicaError("Not enough memory!");

        temp = pi->P * (error + (pi->samplePeriod / pi->Ti) *pi->I);

        if (temp > pi->yMax) {
                y = pi->yMax;
        } else if (temp < pi->yMin) {
                y = pi->yMin;
        } else {
                y = temp;
        }

        pi->I = pi->I + error + (y - temp) / (pi->P * pi->Ni);

        return y;
};
```

3.5.3 Control

All the controls was implemented in *Microsoft Visual C++ 2010 Express* and compiled to an external static library. This way the control code was written completely independent of the models. The control code was then compiled and linked to an external static library which was placed inside Dymola's *lib*-folder. As reflected in figure 3.18 the control was divided into two main control algorithms, one called *TurbineControl* and one called *GeneratorSideController*. The reason to why this was done was that it was desirable to separate the active control algorithms, such as PI- and current controllers, and the more inactive, managing control, which makes decisions such as when to start and stop the wind power unit, from each other. The different control and code parts are described below.

3.5.3.1 Turbine Control

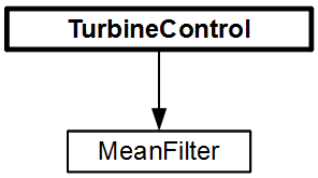


Figure 3.23: Class structure of the Turbine Control.

The Turbine Control is the so called top control of the turbine, it decides when the turbine should start or stop and which mode to operate in, depending on orders, as well as providing useful information to the plant controller. The full implementation is presented in appendix B.1. The Turbine Control is initiated with limits for low and high wind speed, low rotor speed, filter lengths for wind and rotor speed, the air density, the area swept by the rotor, estimation of the maximum C_p-value and the nominal power of the turbine. The limits for low and high wind speed are used in order to be able to decide when the wind speed is too high or low for operation. The low rotor speed is used to decide when the brake should be applied. The air density, area swept by the rotor and the maximum C_p-value is used in order to calculate an estimate of the available power capability of the turbine.

The Turbine Control is implemented using both one internal state and one external state command, the inner state is used in order change state due to too high or low wind speed while the external state command is used in order to select which state that should be used by the power controller. In order to not turn of the turbine due to a small wind gust or an erroneous measurement value the wind speed is filtered using a mean filter, see section 3.5.3.4, the same goes for the rotor speed. The Turbine Control can order the turbine to shut down for three different reasons. The first reason is that the external command tells it to shut down, the second reason is that the wind speed is too low for operation and the third is that the wind speed is too high for operation. When the Turbine Control has decided to shut down the turbine a check to see if the rotor speed is below the low rotor speed limit is performed. If so is the case the brake is applied, if not, and the reason for shut down was due to too high wind speed or an external order, and the rotor speed is below twice that of the low rotor speed limit, the break is applied with 50%.

In order for the Turbine Control to allow operation again the external control should no longer order shut down and the wind speed must be between the limit for low wind speed plus the hysteresis and the limit for high wind speed minus the hysteresis. The hysteresis is there in order to make sure that the turbine is not constantly turned on and off when the wind speed is on the limit of too high or too low.

If the wind speed is OK and the external state order is not shut down the controller allows operation and in all cases except one just passes the control mode and desired power to the power controller. If the control mode should

be delta-control the Turbine Control passes on the mode "POWTRAC" with 80% of the maximum potential power.

The maximum potential power output is calculated in the *getMaxEff()*-function. In order to do this the controller needs to make a difference between ordering shut down due to the wind speed being too low or high and ordering shut down due to the external command. If the turbine has not ordered shut down due to too high or low wind speed the maximum potential power output is calculated according to equation 3.25, otherwise it should be zero.

$$maxEff = 0.5 \cdot \rho A \mathrm{w}^3 \cdot C_{p,max} \tag{3.25}$$

3.5.3.2 Generator Side Controller

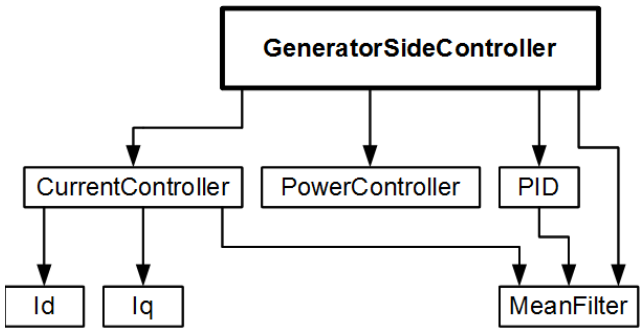


Figure 3.24: Class structure of the generator side controller.

The "GeneratorSideController"-class is not a controller in its own, but a class used in order to update all the controls, except the turbine control, on the generator side. The full C++-code is presented in appendix B.2. The controllers managed by the class are the Power Controller, Speed Controller and the Current Controller. The implementation of the Power Controller is described in section 3.5.3.3, the implementation of the Current Controller is described in section 3.5.3.7 and the Speed Controller is implemented as a PID-controller which is described in section 3.5.3.6. The overall control structure can be seen in figure 3.12.

The initiation of the class has been divided into three parts, one for each of the controllers managed by the class, this in order to keep the number of variables sent to the class in a single function call manageable. The class's main task is to manage the different controls' sampling interval, this since the different controls do not have the same sampling time. To do this a counter for each of the controllers was implemented and the sampling time of the class is set to the same as the lowest of the three controllers. In this case that will always be the current controller since a cascaded control structure is used.

When the initialization of the class is made the counter limits is calculated and the counter is initialized to its counter limit. This in order to make all the controls update during the first sample instant, the counters are then incremented at each sample instant. When the *update()*-function is called the motor angle is derived in order to obtain the rotational speed of the rotor. The class

then starts to check which controllers that should be updated, starting with the controller that is in the uttermost loop. When a control gets updated the corresponding counter is reset to zero. Before the speed controller is updated a filter is implemented in order to filter the speed reference. This is done in order to remove large step changes in the reference. The effect of this filter is that the Speed Controller will not request high, or low, torque spikes from the Torque Controller, which in turn will generate spikes, positive or negative, in the output power from the generator.

3.5.3.3 Power Controller

The "PowerController"-class is the implementation of the power control algorithm described in section 3.4.2. The full C++-code is presented in appendix B.3. The Power Controller is initiated by all the different gains and limits together with the sample period, note that the nominal power, nomEff, is considered to be the maximum allowed power, the nominal DC-voltage is considered to be the setpoint for the DC-control and the w_ref is used as an initial rotational speed reference every time the controller orders a startup of the turbine.

The operation of the controller is divided into four different parts or functions, the main *update()*, the *updateOutput()*, the *leftSlope()* and the *rightSlope()*. In the main *update()*-function the first part of the control update is done, here the controller sets the power reference which later shall be used by the *updateOutput()*-function. The power reference is set in different ways depending on which mode the controller is ordered to work, see section 3.4.2. An extra feature was implemented in order to allow for a faster deceleration of the rotor when the control is ordered to a halt, apart from setting the power reference to zero an extra variable has been implemented indicating that a complete stop of the rotor is ordered. When the power reference has been set it is limited by zero and the nominal power of the turbine, then the current power error is calculated and the *updateOutput()*-function is called.

The *updateOutput()*-function is the implementation of the control algorithm described in section 3.4.2.2 with some modifications. A special algorithm has been implemented for the shutdown of the unit. When the shutdown of the unit is started the rotational setpoint is decreased by 30 rad/s every sample in which the actual rotational speed is below the previous output until the turbine is at standstill. When the turbine is restarted after a shut down it resets the parameters associated with the startup. The action taken when a change in the power reference has been detected was modified so that instead of always doing an update with the "*leftSlope()*" the update is done using the previously used slope. An exception from the condition that a minimal rotational speed change has to have happened before an update is made has been added. If the power error is more than ten times higher than dp_lim, the limit for a change in power, an update is forced in the same way as if the power reference has been changed. Apart from this a step has been added in the middle step where the PI-controller of the left side is updated but never actuated. This in order for the integral part of the controller to keep track of the error during

the relative slow process of changing the rotational speed. Note is that since a maximum step change was desired to be used in the update function the PI-controller described in section 3.5.3.5 could not be used. Instead a similar PI-type controller was implemented where both the maximum step size and the max/min-limits of the output was implemented. The limitation of the step size was implemented by calculating the desired change in the control signal, limit it and add to the previous control signal. The PI-control also implements a simple anti-windup function consisting of calculating the appropriate integral value for obtaining the desired output and set the integral to this value, see equation 3.26, where I is the integral value, y the current output, e the control error and k a control constant. The only change made in the "*leftSlope*" and "*rightSlope*" is that a variable "*last_ state*" which keeps track of what "branch" was last used to update the control was added.

$$I = y - e \cdot k \tag{3.26}$$

3.5.3.4 Mean Filter

In order to filter signals and measurements an averaging filter was implemented. The code used for the implementation is presented in appendix B.4. The mean filter is a simple, yet effective, way to discreetly filter signals. The mean filter works by averaging the signal over a user defined number of samples. In our implementation the mean filter is initiated with the desired length and is then updated using the update function, which also returns the updated average value. There are two additional functions implemented, the *get()*-function, which returns the current average value and the *reset()*-function which is used to reset the memory of the filter, the length of the filter is unchanged by this action, however the average calculations are restarted.

The mean value algorithm is implemented to be working in two different modes. The first mode is when it has enough values to calculate the average over the requested number of values and the second mode is when it does not have enough values to do this. When the algorithm does not have enough values to calculate the average over the requested number of samples the algorithm instead calculates the average over the values it currently got stored.

3.5.3.5 PI-Controller

The PI-controller is used in a lot of different parts of the control structure. The PI-control implemented in this project was quite straight forward. The code used for the implementation is presented in appendix B.5. The PI-controller is initiated with the controller gains, k, Ti and Ni, output limits, sample period and start values for the output and integrator. The PI-controller is implementing an anti windup which the Ni-time constant is used for. The *update()*-function is called using the error of the control signal. The function starts by calculate and limit the output before it applies the anti windup algorithm. The output is calculated according to equation 3.27, where T_s is the sampling period. The output is saved in a temporary variable before the output is lim-

ited, this since both the limited and unlimited output is needed by the anti windup algorithm.

$$y = k \cdot \left(e + \frac{T_s}{T_i} \cdot I \right) \tag{3.27}$$

The reason for why an anti windup is implemented is to prevent the integrator from growing indefinitely when the output value is limited and, if necessary, force the integral value to track the actual output signal. In our implementation this is done by comparing the unlimited output with the limited and adds this to the update of the integral value with a time constant equal to $k \cdot N_i$, see equation 3.28.

$$\begin{aligned} antiWindup &= \frac{y - temp}{k \cdot N_i} \\ I &= I + e + antiWindup \end{aligned} \tag{3.28}$$

Apart from the *init()*- and *update()*-functions the PI-controller also implements a function to directly reset the integral value, *setI()*, as well as a function to set and get its parameters through the *"PiParam"*-class. The *"PiParam"*-class is a container-class used to make it easier to store and pass the PI-parameters to other functions, for the full implementation see appendix B.5. The *setParam()*- and *getParam()*-function sets the gains and other parameters in the PI-controller to the provided ones, alternatively returns the current parameters stored in the *PiParam*-object. The *setI()*-function is called with the desired output and current error, based on this the controller sets the integrator to the appropriate value to achieve the output, see equation 3.29. The idea with this function is to provide for a bumpless control, or parameter, change.

$$I = \frac{T_i}{T_s} \cdot \left(\frac{y_{desired}}{k} - e \right) \tag{3.29}$$

3.5.3.6 PID-Controller

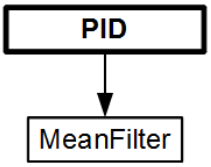


Figure 3.25: Class structure for the PID-controller class.

Also the PID-Controller is used in a number of places of the control structure. The PID controller is in essential a PI-Control with the addition of a derivative part. The difference in the implementation between the PI-Controller and the PID-controller is quite minor. The full implementation is presented in appendix B.6. In the initialization a derivative gain has been added as well as a filter length. Apart from this the *setParam()* and *getParam()*-functions has been removed, and the *setI()*-function has been replaced by *setOutput()*. The PI- and anti windup- part of the control is implemented in the same way

as it was in the PI-Controller, see section 3.5.3.5. The only thing that differs between the PI- and PID-control is the addition of the derivative gain in the calculation of the output.

When implementing derivatives in discrete systems it is always recommended to apply some kind of filtering before the derivation is done, this since all signals are piece wise constant and large step changes are common, in this case the MeanFilter was used, see section 3.5.3.4. The input of the *update()*-function is filtered before it is derived using equation 3.30, where T_s is the sample period. The old error, e_{old}, is initiated to zero.

$$\frac{de}{dt} = der = \frac{e - e_{old}}{T_s} \tag{3.30}$$

Using the derivative of the error the control output is calculated using equation 3.31. The output is then limited, the anti-windup is performed and the current error is stored as e_{old} before the *update()*-function is finished.

$$y = p \cdot \left(e + \frac{T_s}{T_i} \cdot I + K_d \cdot der \right) \tag{3.31}$$

When the *setOutput()*-function is used the current output of the controller is set to the demanded output, the current error provided is stored as e_{old} for calculation of the next derivative and the derivative filter is reset. Apart from this the integral value is updated to fit the new output according to equation 3.32, this in order to provide bumpless control switches.

$$I = \frac{T_i}{T_s} \cdot \left(\frac{y}{p} - e \right) \tag{3.32}$$

3.5.3.7 Current Controller

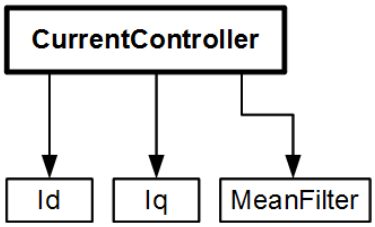


Figure 3.26: Class structure of the Current Controller.

The complete Current Controller for the PMSG[5] was implemented into one single class using the I_d and I_q classes, see section 3.5.3.8, as shown in figure 3.26. The code in its whole is presented in appendix B.7. The Current Controller is initiated with the generator parameters, min/max allowed output value, start values, sample period and a filter length, which the Current Controller uses to initiate the I_d/I_q controller. When the update is performed the

[5]Permanent Magnet Synchronous Generator

current measurements, the rotational speed of the generator and the torque reference is provided. The input torque reference is filtered using the mean filter in order to make the control calmer, if this would not be desired the filter length can be set to zero. The Current Controller starts by calculating the current references based on the desired output torque. There are a number of different ways to do this see section 3.4.3. The method implemented in this project was chosen according to equation 3.33, where pp is the number of pole pairs in the generator. The Current Controller then updates the I_d and I_q controllers.

$$\begin{aligned} Id_{ref} &= 0 \\ Iq_{ef} &= \frac{\tau_{ref}}{pp \cdot \psi_m} \end{aligned} \tag{3.33}$$

3.5.3.8 Id/Iq-Controller

In order to achieve a good control of the permanent magnet synchronous generator specialized current controllers was implemented. The full code is presented in appendix B.8. As described in section 3.4.3.2 the current controllers are a modification of the normal PI-Controller, the modifications consists of parametrization of the controller gains and an addition of a compensation for the back electromagnetic force, because of this the implementation is quite similar to that of the PI-controller, see section 3.5.3.5. Because of the parametrization the *init()*-function was changed so that instead of the controller gains the machine parameters are provided as an input, the optimal gains are then calculated according to equation 3.34. In order to make the control more robust the gains were decreased, the proportional gain halved and the integral time constant doubled. This is needed in order for the system to not be unstable because of deviations in the machine parameters.

$$\begin{aligned} P_{id} &= \frac{L_d}{T_s} + \frac{R_s}{2} \\ Ti_{id} &= \frac{L_d}{R_s} + \frac{T_s}{2} \\ P_{iq} &= \frac{L_q}{T_s} + \frac{R_s}{2} \\ Ti_{iq} &= \frac{L_q}{R_s} + \frac{T_s}{2} \end{aligned} \tag{3.34}$$

The update of the control signal is managed in the same way as for the PI-controller with the difference that the *update()*-function, apart from needing the error also needs to know the electrical rotational speed of the generator as well as the "other" current, the I_d controller needs to know the I_q current and vice versa. The controller output can then be calculated in the same way

as the done in the "normal" PI-controller with an addition of the back emf-compensation, see equation 3.35, the control output is then limited and the anti windup is performed.

$$\begin{aligned} y_{id} &= P_{id} \cdot \left(e + \frac{T_s}{Ti_{id}} \right) - \omega_e \cdot L_q I_q \\ y_{iq} &= P_{iq} \cdot \left(e + \frac{T_s}{Ti_{iq}} \right) + \omega_e \cdot (\psi_m + L_d I_d) \end{aligned} \tag{3.35}$$

3.6 Simulations and Results

In order to test how well our different controls and optimization algorithms perform a couple of test cases was simulated, see section 3.1. The different test cases were designed to either test different aspects of the control or how well the controllers cope with a specific task. The different controllers have been divided into three different subsections, the Speed Controller, Optimization Algorithm and Turbine Control.

3.6.1 The Speed Controller

The Speed Controller's task is to control the speed with which the PMSM rotate. The control of the PMSM is identical for both the case of regular operation as a motor and that of a generating operation as a generator, the only difference between the two modes is the direction of the power flow. For this reason two tests were performed. One for running the machine as a motor and one for running the machine as a generator.

3.6.1.1 Used to Run a Motor

The test bench used for this test can be seen in figure 3.27. In this case we have that an optimal DC-grid was connected to the inverter. The Speed Controller uses the inverter to control the PMSM. The speed reference is set to a constant 10 rad/s, and after 30 seconds the load torque is increased from 10 Nm to 60 Nm. The result from the simulation is shown in figure 3.28 and 3.29. In figure 3.29 the Speed Controller was tuned to be quite gentle in order to not generate any large negative power spikes in the consumed power, while in figure 3.28 the Speed Controller was tuned in order to obtain as good result as possible, not taking the power consumed by the motor into consideration. When the controller was tuned for performance it can be seen that it follows its reference strictly, while the gentler controller is a bit slower. However when observing the power consumed by the motors, the motor controlled by the aggressive control consumes very large spikes of energy, $> 20kW$, while the motor controlled by the more gentle controller do not have any large spikes in the consumed energy at all and in total never exceeds a power consumption of $2kW$.

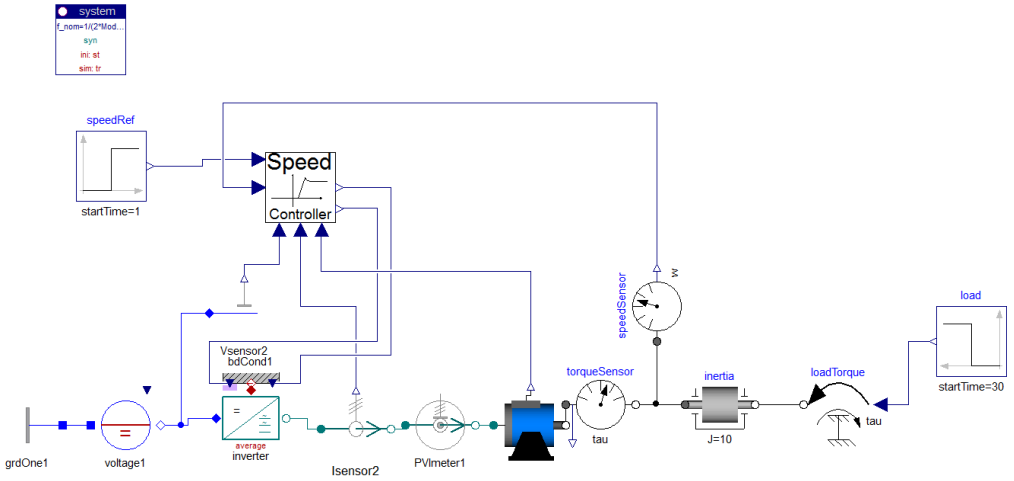


Figure 3.27: Test-bench used for testing the Speed Controller when used to run a motor.

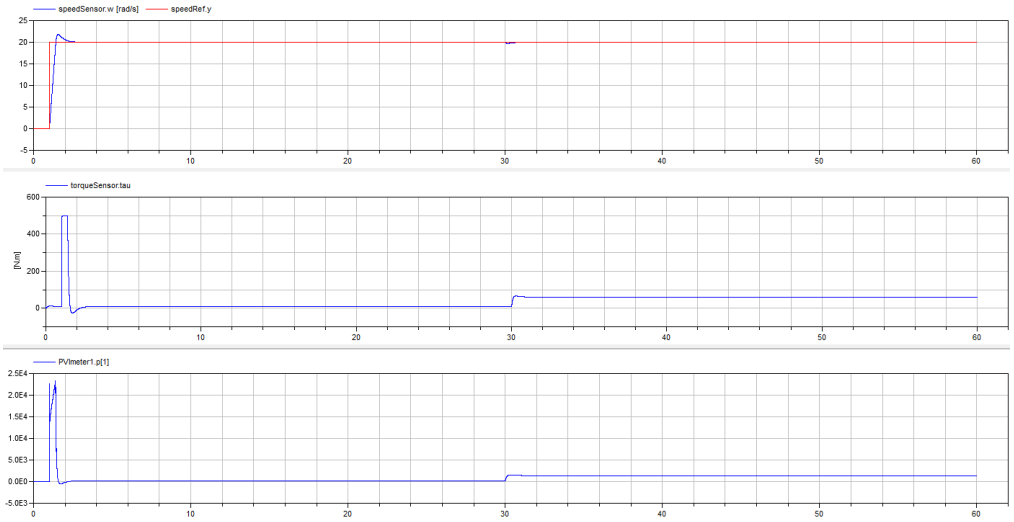


Figure 3.28: Results when testing the aggressively tuned Speed Controller used to run a motor. In the top subplot the rotational speed and its reference are plotted, in the middle subplot the actuated torque from the motor is plotted and in the bottom subplot the power flow directed into the motor is plotted.

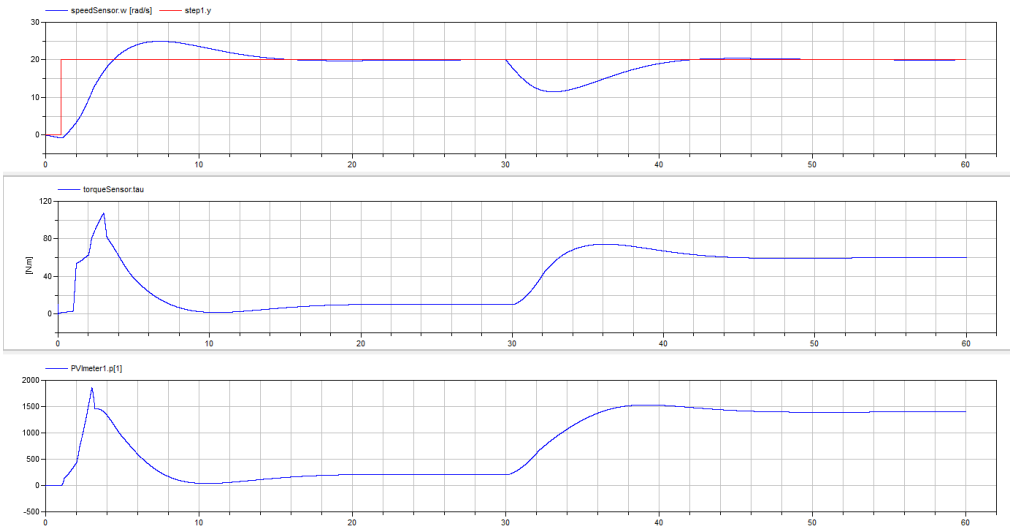


Figure 3.29: Results when testing the more gently tuned Speed Controller used to run a motor. In the top subplot the rotational speed and its reference are plotted, in the middle subplot the actuated torque from the motor is plotted and in the bottom subplot the power flow directed into the motor is plotted.

3.6.1.2 Used to Run a Generator

The test bench used for this test can be seen in figure 3.30. In this case we have assumed that an optimal torque source is acting as the power source on the rotor. The Speed Controller is using the inverter to apply a negative torque on the shaft in order control the rotational speed of the rotor. By doing this power is generated to the DC-grid, which consists of a resistive load. In this test the Speed Controller was tuned the same way as for the gentle control in the motor test, 3.6.1.1.

As can be seen in the top graph of figure 3.31 the generator follows its reference value with a bit of an overshoot and some small oscillation to begin with but settles and follows the reference. During the ramp the controller successfully tracks the reference, and when the driving torque is increased the controller reacts and drives the rotational speed back to its reference. The fact that the machine is running as a generator is shown when observing the power flow into the generator, this since the power flow is negative. The benefit of having a gentler controller is more obvious when running as a generator since it is very desirable to have a constant power output from the generator.

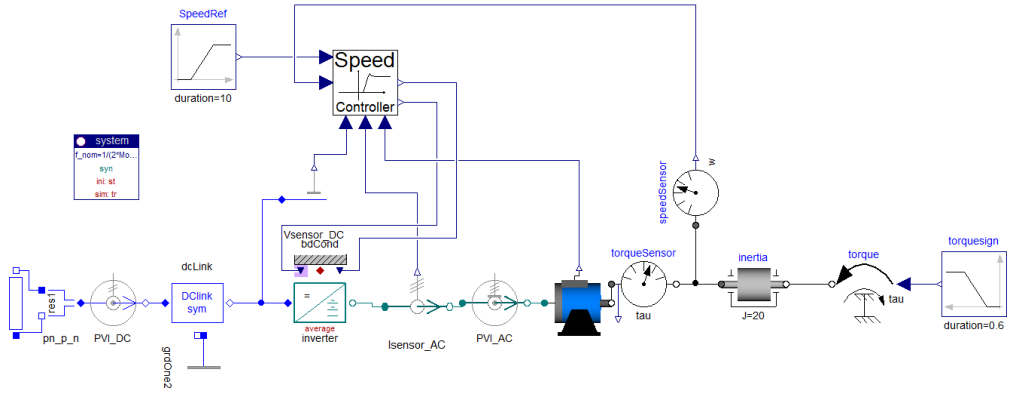


Figure 3.30: Test-bench used for testing the Speed Controller when used to run a generator.

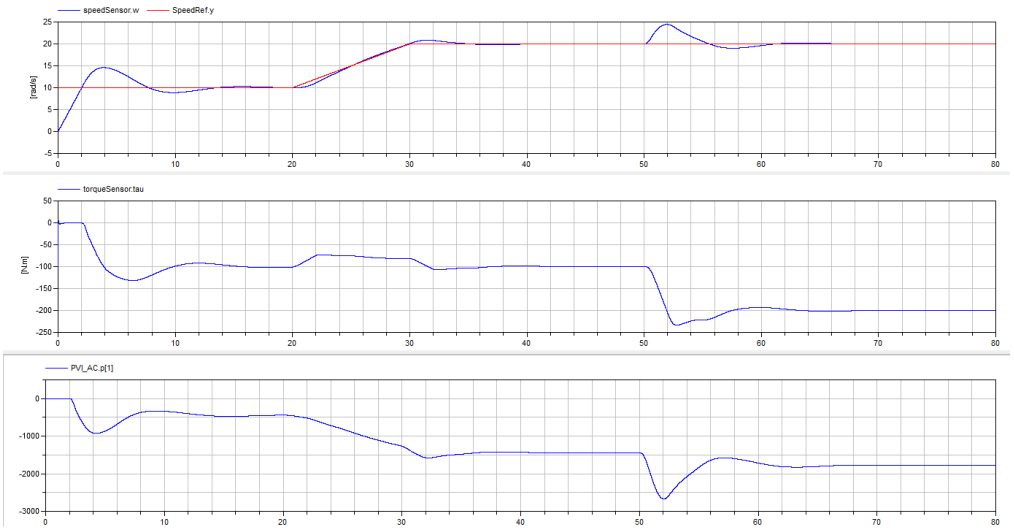


Figure 3.31: Results when testing the Speed Controller used to run a generator. In the top subplot the rotational speed and its reference are plotted, in the middle subplot the actuated torque from the motor is plotted and in the bottom subplot the power flow directed into the motor is plotted.

3.6.2 Optimizing Algorithms

The optimizing algorithms' task is solely to optimize the power output from the power unit, this is done by two different algorithms. The first algorithm only measures the power that is extracted from the wind and converted to electrical power whilst the other algorithm also measures the rotational speed of the generator. For both case 1 and 2 the wind speed profile was chosen to start at 8m/s and increase after 200s to 11m/s over 100s.

3.6.2.1 Case 1 - MPPT Sensorless

This case is designed in order to test the performance of the Sensorless MPPT algorithm, see section 3.3.2. The Sensorless MPPT's task is to maximize the power output from the wind power unit by maximizing the C_p-curve, see section 2.2, without using any other measurements then the power measurements.

In figure 3.32 the result from the simulation is presented. As can be seen the Sensorless MPPT successfully finds the optimal operation point and keeps the unit there during most of the simulation. However it takes some time for the algorithm to find the optimal operation point, as well as recover after a wind gust. The maximum power output plotted together with the power output is not in any way communicated to the algorithm but is only calculated by the turbine control in order to visualize the maximum output. The algorithm is quite sensitive to changes in wind speed, such as gusts, which can be observed after about 100s. When a gust occurs the power measurement is largely affected which forces the algorithm to wrongfully adjust the rotational speed reference.

Since the algorithm only measures the power output and assumes the wind power unit has reached its previous rotational speed reference it requires a relatively long sampling period[6]. If the rotational speed reference should not have been achieved, the assumption might cause the power measurement to be less than accurate and the algorithm runs a risk of making wrong decision. In worst case scenario the algorithm gets "lost" and ends up outputting a completely erroneous value.

[6]In this simulation the sampling period was chosen to 15s.

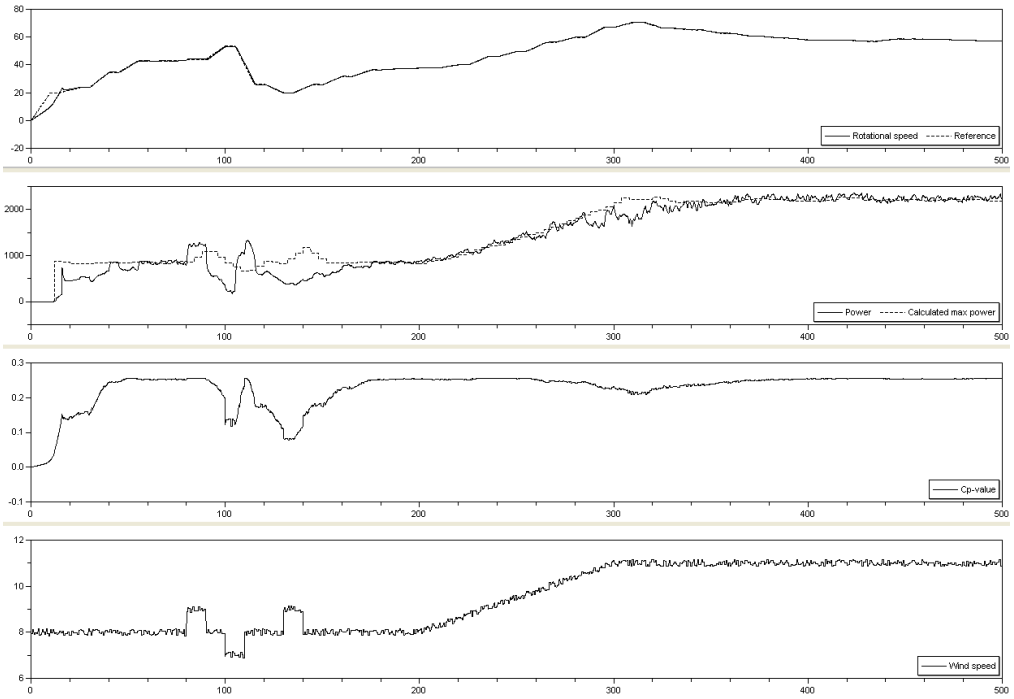


Figure 3.32: Simulation results for case 1, in the first subplot the rotational speed of the rotor and its reference are plotted, in the second subplot the power delivered to the DC-grid together with the maximum potential power output calculated by the Turbine Control are plotted, in the third subplot the C_p-value is plotted and in the fourth subplot the wind speed is plotted.

3.6.2.2 Case 2 - MPPT

This case is identical to case 1 with the difference that the component optimizing the power output is the MPPT, see section 3.3.3. The MPPT's task is to maximize the power output from the unit by maximizing the C_p-curve. The difference between this algorithm and the previous one is that this algorithm utilizes both power measurements and measurements of the generator's rotational speed.

In figure 3.33 the result from the simulation is presented. When observing the C_p-value in the diagram it can be seen that the algorithm successfully controls the wind power unit to its optimal operational point, and maintains it there during most of the simulation.

When comparing with the results from case 1 it can be seen that the MPPT requires much less time to recover from changes in wind speed, such as gusts, than the Sensorless MPPT. In fact the overall disturbances in the C_p-value due to changes in wind speed is drastically decreased, and the overall performance of the operation is improved.

By also measuring the rotational speed the sampling period can be reduced drastically[7], and the risk of getting "lost" is removed.

[7]In this simulation the sampling period was chosen to 2s.

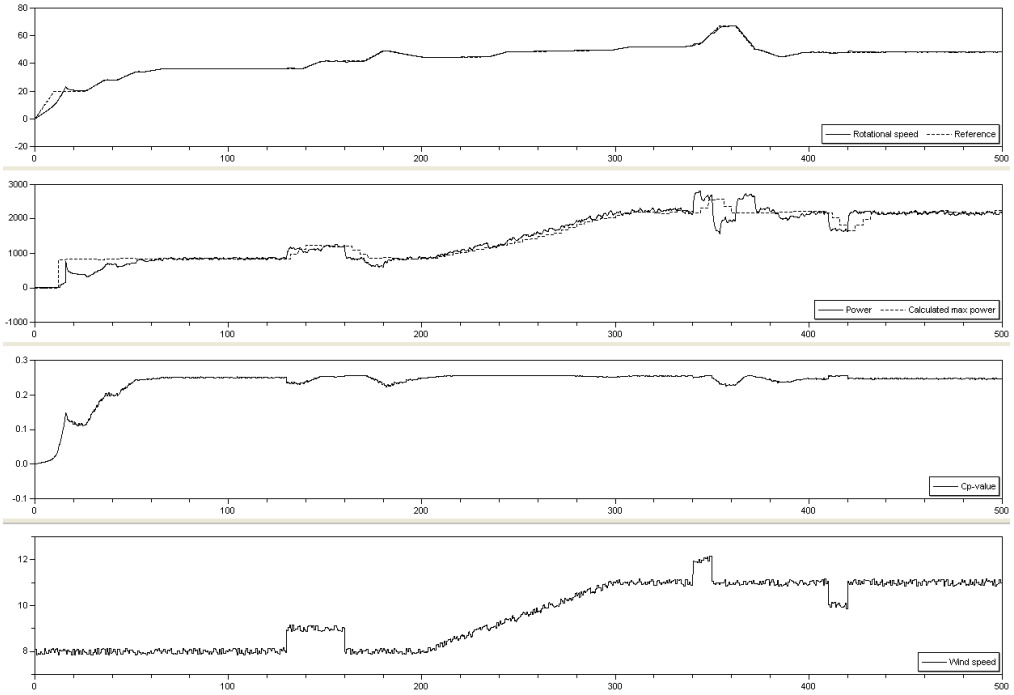


Figure 3.33: Simulation results for case 2, in the first subplot the rotational speed of the rotor and its reference are plotted, in the second subplot the power delivered to the DC-grid together with the maximum potential power output calculated by the Turbine Control are plotted, in the third subplot the C_p-value is plotted and in the fourth subplot the wind speed is plotted.

3.6.3 Turbine Control

The Turbine Control's task is to manage the wind power unit, deciding when it should operate and, in combination with the Power Controller, at what speed. In order to test all operation modes of the control system a number of different cases have been designed, see section 3.1.

3.6.3.1 Case 3 - To Low for Nominal Power

This test is designed to test how well the Power Controller performs as an optimizing algorithm. The task in this case is to maximize the power output from the unit when the wind speed is not high enough to achieve the requested output power. The unit starts at standstill.

The results from the simulation are shown in figure 3.34. As can be seen the wind speed starts at 8 m/s and after 200s it increases to 11 m/s over 100s, which is the same wind profile which was used for testing the optimization algorithms. When observing the power output from the unit it can be noticed that the output is zero until the rotational speed reference has been reached, this since the generator is not applying any negative torque. After this the power output is approximately 1 kW, and when observing the C_p-value in figure 3.34 it can be seen that the value is very close to its maximum, which for this

test was ~ 0.26. When the wind speed is increased the Power Controller reacts and adjusts the rotational speed of the rotor, after some time the C_p-value has been returned to its maximum. It takes some time for the algorithm to recover after the increase in wind speed, however the value is during this time maintained in the proximity of the maximum. The wind gusts are reflected both in the output power and the C_p-value. The main reason to why the effect is so visible in the output power is that the wind power unit which was simulated is quite small, a larger rotor, with higher inertia, would do a better job of filtering these "bumps". The effect of the gusts on the C_p-value is quite large, however the goal of the optimizing algorithm should not be to maximize the output power over these gusts, but instead to maximize the power output during a long period of time. If the algorithm was to try maximizing the output power of the gusts the control is forced to be very aggressive and "nervous" which is not desirable.

This case is in fact identical to case 1 and 2. When comparing the result with case 2 it can be seen that the performance of the two algorithms is quite similar, which not that surprising considering that the optimization algorithm is implemented in the power controller is based upon the MPPT.

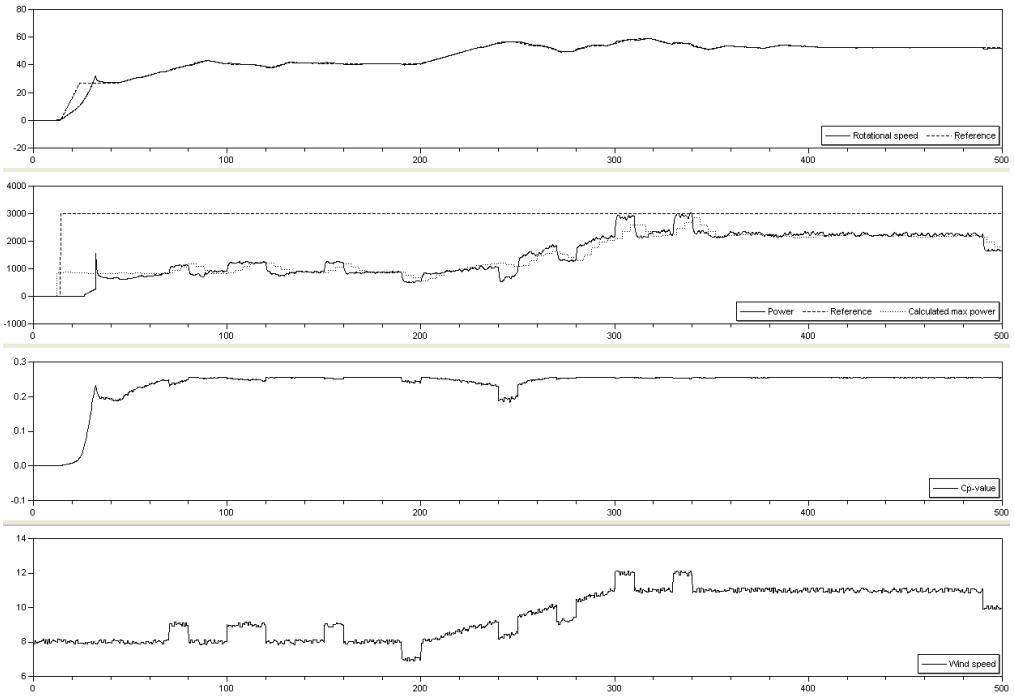


Figure 3.34: Simulation results for case 3, in the first subplot the rotational speed of the rotor and its reference are plotted, in the second subplot the power delivered to the DC-grid together with the power reference and the maximum potential power output calculated by the Turbine Control are plotted, in the third subplot the C_p-value is plotted and in the fourth subplot the wind speed is plotted.

3.6.3.2 Case 4 - Nominal Power

This test is designed to test the power control part of the Power Controller, which is to test if the Power Controller can make the wind power unit output the requested power when there is enough wind to do so. The unit starts at standstill.

The results from the simulation are shown in figure 3.35. As can be seen the wind starts at 14 m/s and after 150 seconds the wind speed increases to 18 m/s over 50 seconds and at 400-500 seconds it slowly decreases again to 14 m/s. Just as in case 3, see section 3.6.3.1, the output power from the unit is zero until that the generator applies a negative torque. The power output from the rotor is steadily increased until the reference value is reached. After 250 seconds, when the wind speed increases, the power also increases briefly but is then controlled back to its reference. Also when the wind speed is decreased slowly the output power is slowly decreased, however the Power Controller reacts accordingly and the output power is controlled back to its reference value. During the whole simulation the power is kept, approximately at its reference value. There are however small deviations from the references which occurs due to the wind gusts. It can also be noticed that the maximum power output calculated by the Turbine Control is limited by the nominal power.

In figure 3.36 the torque output from the wind turbine is shown. In the figure the tower shadowing effect, see section 3.2.2 is clearly visible. It can also be noticed that when the wind speed decreases at 400-500s the torque output also is decreased, however the power controller increases the rotational speed in order to achieve the requested power output.

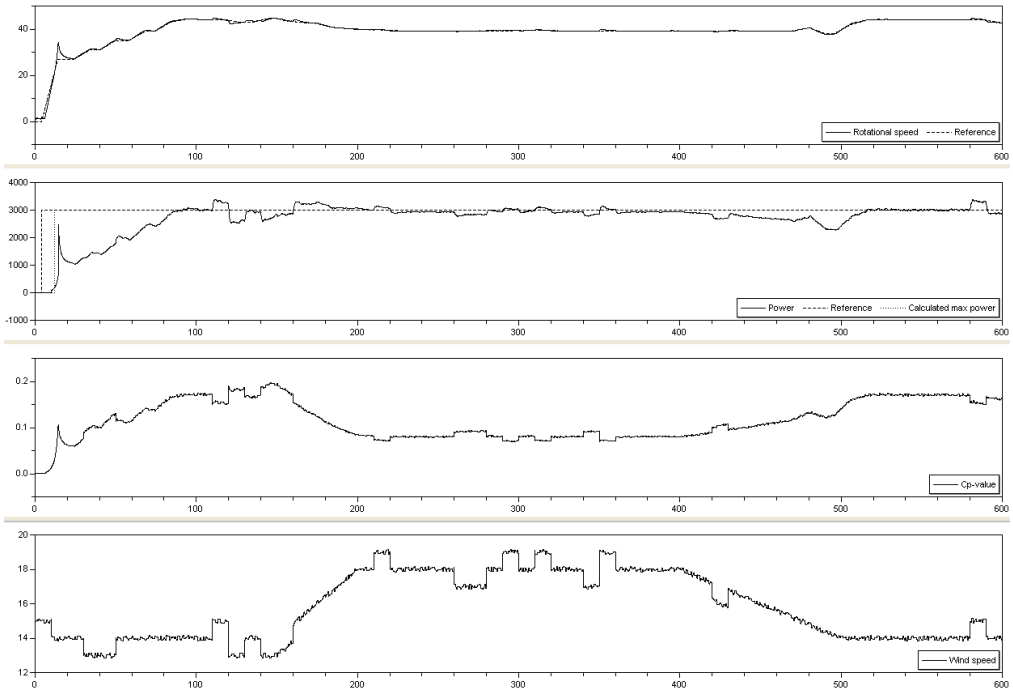


Figure 3.35: Simulation results for case 4, in the first subplot the rotational speed of the rotor and its reference are plotted, in the second subplot the power delivered to the DC-grid together with the power reference and the maximum potential power output calculated by the Turbine Control are plotted, in the third subplot the C_p-value is plotted and in the fourth subplot finally the wind speed is plotted.

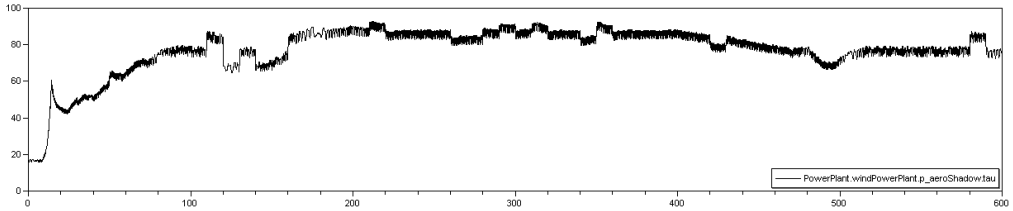


Figure 3.36: Simulation result for case 4, in this plot the torque output from the wind turbine is plotted in order to show the tower shadow effect.

3.6.3.3 Case 5 - Power Point Tracking

This test is designed in order to further test the power control part of the Power Controller. In this test the power output from the wind power unit is supposed to follow the reference value provided. The unit starts at standstill.

The results from the simulation are presented in figure 3.37. The power reference chosen for this test was 2675W at the start and after 300s the reference is changed to 2140W. The wind speed is varied between 13 and 18m/s, after 150s the wind speed is increased from 14 to 18m/s over 50s and after 400s it is decreased from 18 to 13m/s over 100s. The control quickly reaches the reference value and keeps it there during the simulation. The changes in wind speed are reflected as a small overshoot when the wind speed increases

and a small drop when the wind speed decreases. When the change in the power reference occurs the power output is decreased in a controlled way, by decreasing the rotational speed of the rotor, to the new reference value. One observation that can be made when comparing the effects of the wind gusts in the power delivered from the unit, is that the effect of the gusts is much smaller when not maximizing the output. The reason for this phenomenon can be found in the equation for the power absorbed from the wind, see equation 3.1. When maximizing the power output a high C_p-value is obtained, while in this case a relatively low C_p-value is obtained. Because of this a relatively lower portion of the wind speed is transformed to rotational energy which in turn makes the relative wind speed sensibility lower.

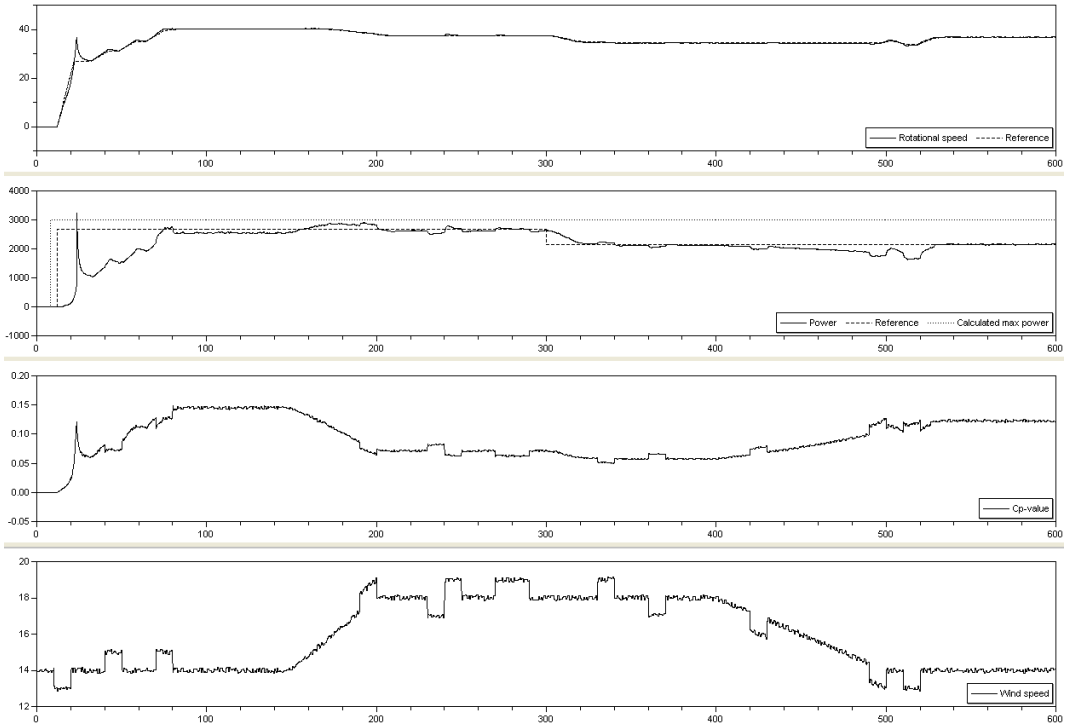


Figure 3.37: Simulation results for case 5, in the first subplot the rotational speed of the rotor and its reference are plotted, in the second subplot the power delivered to the DC-grid together with the power reference are plotted, in the third subplot the C_p-value is plotted and in the fourth subplot the wind speed is plotted.

3.6.3.4 Case 6 - Delta Control

This case has been designed in order to test the delta-control aspect, see section 4.2.1. The unit starts at standstill.

The results from the simulation are presented in figure 3.38. For the simulation the wind speed varies between 8 and 15m/s in order to test the delta control both when the wind speed is too low for nominal power and when the wind speed is sufficient for nominal power. The delta control is achieved by calculating the potential maximum power output from the unit and set the

power reference to 20%[8] below the current maximum output. In the diagram it can be seen that the power reference constantly is 20% below the calculated maximum value. When the wind speed is high enough for allowing operation at the nominal power the control does a very good job of controlling the output power to 20% below the maximum. When the wind speed is too low for nominal effect the control has a harder time tracking the power reference. One of the reasons to this is that the power reference is not changing when the wind speed is high enough for nominal power, this since the maximum power is limited by the nominal power. When the rotational speed is too low for nominal power the maximum power is increased/decreased by the gusts, and since the wind speed is filtered using an averaging filter the effect of the gusts in the maximum power is slightly shifted in time, which makes it even harder for the power control to track the reference.

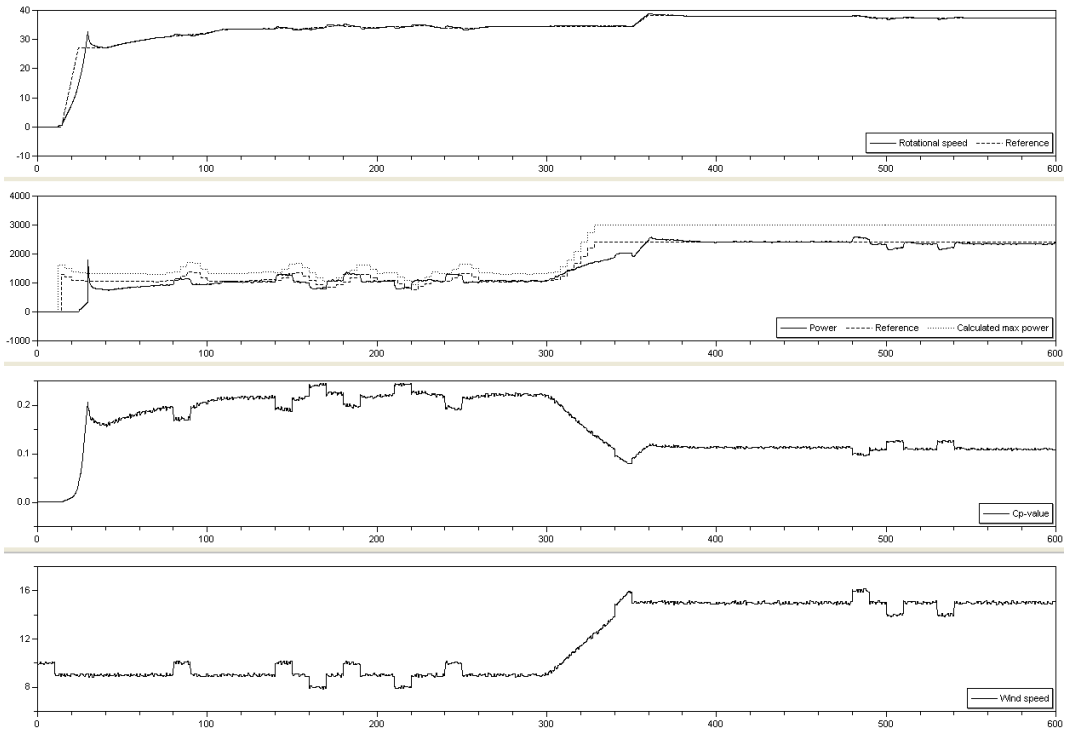


Figure 3.38: Simulation results for case 6, in the first subplot the rotational speed of the rotor and its reference are plotted, in the second subplot the power delivered to the DC-grid together with the power reference are plotted, in the third subplot the C_p-value is plotted and in the fourth subplot the wind speed is plotted.

3.6.3.5 Case 7 - Total Case

This case has been designed in order to test as many features of the Power Controller as possible. To do this the wind speed varies from 0 - 23 m/s during

[8]In this project it was chosen to set the power reference 20% below the maximum, however this value is interchangeable and can even be swapped to a control signal from the Plant Control.

the simulation, this should cause the power controller to shut down the wind power unit both for too high and to low wind speeds, it should also optimize the power output when the wind speed is too low for nominal power and control the power output when the wind speed is sufficient. More importantly it should also confirm that the transition between the different control modes works well. The limits for low and high wind speed were chosen to 6 m/s respectively 20 m/s.

The simulation has been divided into 7 different zones, depicted in figure 3.39, in zone:

1. The wind speed starts at 0 m/s and increases to 5 m/s. During this period the rotor's rotational speed and the power reference are both zero. This since the wind speed is too low for operation of the unit.

2. In this zone the wind speed is increased from 5 m/s to 10 m/s which is enough to allow operation of the unit. The power reference is raised to its nominal value, 3000 W, and the rotor starts to rotate. Since the wind speed is not enough for nominal power the Power Controller tries to optimize the unit's power output, as can be seen in the graph of the C_p-value in the third subplot of figure 3.39.

3. In this zone the wind increases further and is now high enough to allow operation at nominal power.

4. In this zone the wind speed increases to 23 m/s, which is more than the maximum allowed for operation. Hence the power reference is decreased to zero and the generator decelerates the rotor speed. When the rotational speed is low enough the brake is applied and the rotor stops.

5. The wind speed is now reduced to 18 m/s, which is lower than the maximum allowed wind speed for operation, and the power reference is increased to its nominal value and the rotor starts to rotate again. Since the wind speed is high enough for nominal power the Power Controller controls the power to its nominal value.

6. The wind speed is now reduced to 11 m/s which is too low for nominal power and the Power Controller tries to optimize the unit's output power, which can be seen by looking at the C_p-value in the third subplot of figure 3.39.

7. In this zone the wind speed is decreased to 3 m/s which is well below the minimum allowed wind speed, this causes the Turbine Control to give orders to shut down the wind power unit. The power reference is set to zero and the generator decelerates the rotor. When the rotor's rotational speed is low enough the brake is applied.

As can be seen in zone 1, 4 and 7 the Turbine Control successfully makes the decision to turn off the unit when the wind is either too high or too low for operation. The transition between the different control modes, which can

be seen in the transition from zone 2 to 3 and 5 to 6 [9], is working correctly. In the transition from zone 2 and 3 a large power overshoot is present. The main reason for this is that the increase in wind speed is causing the rotor to rotate faster, which the Speed Controller is counteracting, however a small error is left during the increase of the wind speed. This error is larger than the minimum error in rotational speed allowed for updating the Power Controller's algorithm. When observing the C_p-value in zone 7 a large spike can be observed, the reason for this is that when the rotor decelerates it passes through its optimal rotational speed, see section 2.2.4.

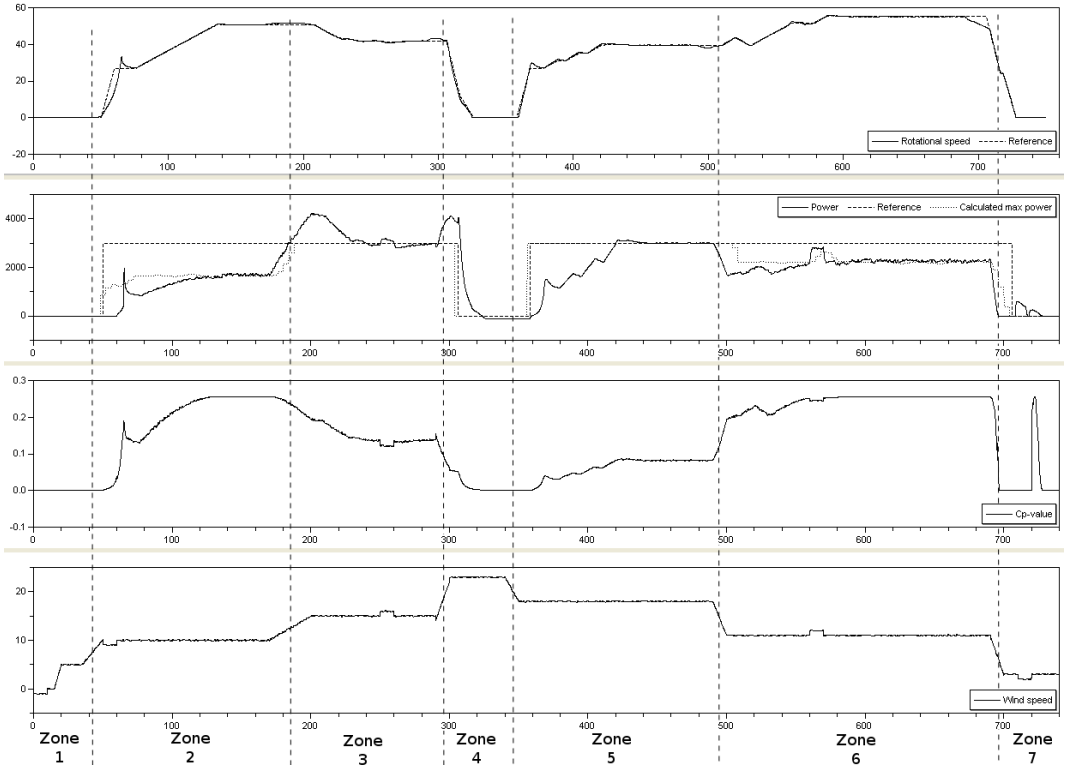


Figure 3.39: Simulation results for case 7, in the first subplot the rotational speed and rotational speed reference of the rotor are plotted, in the second subplot the power out to the DC-grid, the reference value for the power output and the estimated maximum power output are plotted, in the third subplot the C_p-value is plotted and in the fourth subplot the wind speed is plotted.

3.6.4 Losses

As discussed in section 2.3 losses are always present during power transfers and conversions. In figure 3.40 the power output from the different parts of the unit is presented. This in order to show the losses modeled into our system. The powers plotted is kinetic wind energy absorbed by the rotor and converted to rotational energy, the electrical power delivered from the generator to the

[9] in zone 2 to 3 the transition between optimization and nominal power can be observed, and in zone 5 to 6 the transition between nominal power and optimization can be observed.

inverter and the electrical power delivered from the inverter to the DC-grid. As expected there exist substantial losses inside the wind power unit. In this case the transfer loss and generator loss is by far the greater of the two with a loss of approximately ∼15%, these losses are probably mostly resistive losses in the generator, see section 2.3.2. The loss from the inverter is approximately ∼5% and overall the losses in the wind power unit sums up to ∼19%.

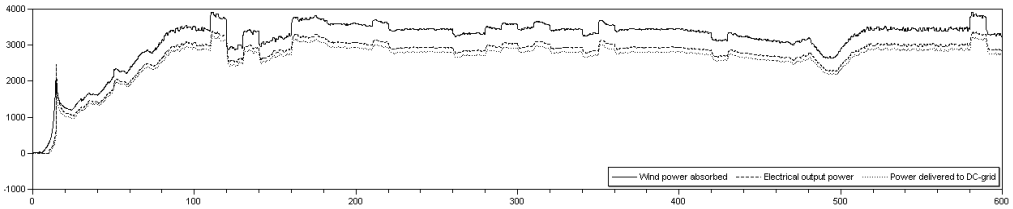


Figure 3.40: In this plot the losses for the wind power unit are presented. In the diagram the power outputs from the rotor to the electrical power output on the DC-grid are plotted.

3.7 Discussion

After some initial issues there were no real problems when implementing our models. Also the implementation of the control code was done without any major issues. The biggest problem encountered was that Dymola disregarded the function calls since they did not alter any states in the models. Overall the *External Static Library* implementation proved to be a good way of implementing the controls. The biggest inconvenience with the implementation was that when an input or output was to be changed the C-code, interface and Modelica-code needed to be changed.

In the beginning we had some problems to control the generator without generating large spikes when the rotor was decelerated, and large drops in the power output when trying to accelerate. This was finally solved by first trying to use less aggressive parameters for the speed controller, which greatly improved the results, however not enough to achieve our goals. To further improve the control a filter was introduced for the speed reference which solved the problem, which is shown by the current results. Retrospectively it feels quite obvious that a step-wise constant speed reference will produce the large power spikes, and drops, observed.

Overall the control algorithms designed in this chapter performed well, however there is always room for improvement. For example the sensor-less-MPPT could maybe be improved by filtering the power measurement in a smart way. One filter-strategy that comes into mind is a median filter which have the possibility of "removing" a power measurement that probably is caused by a gust.

In our implementation of the delta-control the delta value was chosen to 20%, retrospectively it probably would have been better to have used a variable delta value.

Since we did not have any access to the real wind power unit the parameters used for the generator, shaft and inverter was not based on real equipment. Instead parameters from laboratory equipment were used. The effect of this is that the loss estimation is not accurate, however the overall operation should not be affected in a significant way. For example the optimization algorithms are strongly dependent on the shape of the C_p-curve, and even though the absolute values of the curve is not accurate the general shape is, which in turn allows testing of the control algorithms.

Chapter 4

Grid Connection

The grid integration part of a wind power plant is an important issue, the grid transports the electrical energy produced by the wind power plant to the consumers. In order to connect the wind plant to the grid, the power plant has to meet the grid codes set up by the grid owner, to ensure the quality of the produced power and the safety of the grid. Multiple paths between the power sources and the loads increases the reliability in the system since a failure of one line will not cause a failure of the whole system. However if a failure of the system occurs a part of the electrical network might be disconnected from the grid, in this case a power plant can be ordered to single handedly power up a part of the grid on its own, in a so called Island operation.

In this chapter the transportation of the power from the DC-grid to the consumers will be modeled and controlled. The model of the grid and the control system controlling the grid connection is described. The grid is modeled in two different variants - Strong and Island. Apart form this the grid codes for a wind power plant in Denmark will be listed and compared with grid codes from Sweden.

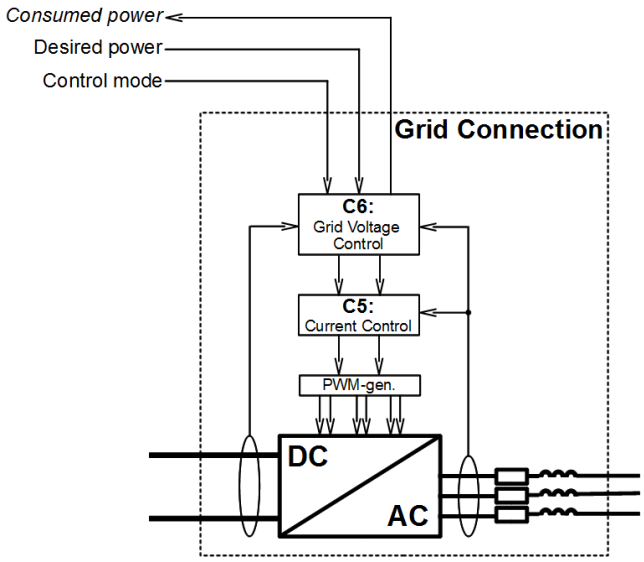


Figure 4.1: Schematic overview over the grid connection.
Note: In the figure an impression that an external control is controlling the power output to the grid, due to the Desired power and Consumed power signals. This is **not** the case, which will later be explained.

4.1 Objectives

One of the objectives in this project is to model the grid connection, including the grid, the grid connection is defined according to figure 1.1, as one inverter connected directly to the point of connection. A more detailed figure with a proposed control structure is presented in figure 4.1. However to accurately model and control the grid connection the grid also needs to be modeled, as visualized in figure 1.1 the grid will be modeled as a transformer connected via a grid to the consumers. Two different grid models will be done, one modeling a strong grid, used for normal operation and one modeling the case of Island operation. The objective in this chapter is to model the grid connection, power transportation and control structure. The control structure designed should be able to achieve a number of different control tasks listed below.

- Deliver a specific amount of active power from the DC-grid.
- Control the DC-level on the DC-grid by delivering active power to the grid.
- Simultaneously control the DC-level on the DC-grid and the voltage level on the AC-side by delivering both active and reactive power to the grid.
- Automatically decide, depending on the DC-level, when delivering power to the grid is desired.
- Control both the voltage level and frequency on the AC-side, which is support island operation.

The grid connection which is to be controlled is configured as a DC-grid connected through an inverter to the point of connection, as proposed in figure 4.1. In the figure a potential control structure is also proposed.

As a part of the project the control designed for the grid connection will be evaluated and compared with grid codes, mainly from Denmark.

4.1.1 Test Cases

In order to test the models and different control modes that will be developed in this chapter a number of different test cases should be evaluated. The test cases are constructed in way so that they will test different realistic control scenarios and/or a specific control algorithm. The results from the test cases should be evaluated and compared with appropriate grid codes. Some control modes, such as the DC-control, requires the whole system to be operational, these control modes will therefore instead be tested in chapter 5.

1. Test to deliver a specific amount of active power from the DC-grid
2. Test to deliver a specific amount of active power while simultaneously controlling the voltage level on the AC-side.
3. Test to simultaneously control the AC-voltage level and frequency, island operation.
4. Test a "total case" where the DC-level is varying and a varying amount of active power is delivered to the grid with and without control of the voltage level of the AC-side.

4.2 Grid Codes

In order to connect a power plant to the grid the power produced by the plant must be of a certain quality and fulfill a number of rules, grid codes. The purpose of the grid codes is to maintain a secure operation of the grid and minimize risks of damages and disturbances. There are a number of different grid codes in Europe, each country has its own, but the general grid codes for each country are more or less the same. However there are different grid codes for different sort of power plants. For example in Sweden there are no requirements for reactive power compensation for a wind power plant. This is because the total wind power production only makes up 2.4% of the total power production in Sweden [3]. The grid codes in Sweden for wind power will in the near future very likely change concurrently with increasing extension of wind power. In Denmark on the other hand the grid codes for wind power are stricter. This because a large investment in the wind power production already has been made. In the following sections follows a selection of grid codes for a wind power plant in Denmark and a short comparison with the grid codes in Sweden, the reason to why the grid codes of Denmark are explored to a larger extent is that they are more developed and thus also more interesting.

4.2.1 Terminology and Definitions

In [19] and in [20], the grid codes for Denmark respectively Sweden are described. The grid codes in Denmark are governed by Energinet.dk, the Swedish counterpart is Svenska Kraftnät. The following definitions are taken from [19].

The Electricity Supply Undertaking

The electricity supply undertaking is the enterprise to whose grid a wind power plant is connected electrically. For voltage levels up to 100 kV it is the local distribution network operator, and for voltage levels greater than 100 kV it is the regional transmission operator.

Wind Power plant

A wind power plant is one or several wind turbine generator systems with a total rated power greater than 25 kW which has been connected to the public electricity supply network.

Small Wind Turbine (SWT)

A small wind turbine is one or several wind turbine generator systems with a total rated power of up to 25 kW which has been connected to the public electricity supply network.

Point of Connection

The point of connection (POC) is the point in the public electricity supply network where the wind power plant is or can be connected.

Point of common coupling

The point of common coupling (PCC) is the point in the public electricity supply network where consumers are or can be connected. Electrically the point of common coupling and the point of connection may coincide. The point of common coupling is always placed closest to the public electricity supply network.

Frequency Control

The control of active power with the goal of stabilizing the grid frequency at its reference is called frequency control.

Absolute Power Control

An absolute production constraint is used to constrain the active power from a wind power plant to a predefined power limit in the point of connection. An absolute production constraint is typically used to protect the public electricity supply network against overloading. See figure 4.2.

Power Gradient Control

A power gradient constraint is a device controlling active power with a fixed increase/decrease (gradient) of the active power. See figure 4.2.

Delta Control

A delta production constraint is used to constrain the active power from a wind power plant to a required constant value in proportion to the possible active power. A delta production constraint is typically used to establish a control reserve for control purposes in connection with frequency control. See figure 4.2.

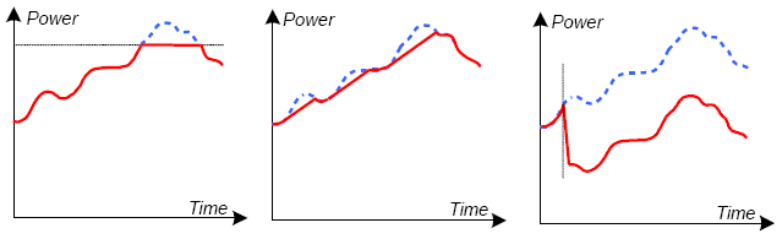


Figure 4.2: Different control functions, from left: absolute power control, power gradient control and delta control.

Q-control

Q control is a control function controlling the reactive power independently of the active power in the point of connection. See figure 4.3.

Power factor control

Power factor control is a control function controlling the reactive power proportional to the active power in the point of connection. The gradient of the line is known as the power factor. See figure 4.3.

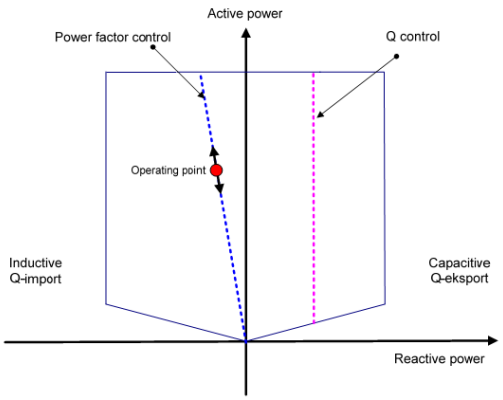


Figure 4.3: Different control functions, from left: power factor control and Q-control. [19]

Different Plant Size

In both Denmark and Sweden there are different codes depending of the size of the wind power plant. In Denmark the definitions are described below [19]:

- Wind power plants with a power output range of 11 kW to 25 kW (typically called "small wind turbines") - requirements and power limits have been harmonized with future/existing European and other international standards.
- Wind power plants with a power output range of 25 kW to 1.5 MW - requirements and power limits are defined in accordance with Danish legislation, which requires that local development plans is prepared.
- Wind power plants with a power output range of 1.5 MW to 25 MW - requirements and power limits have been harmonized with other technical regulations for electricity-generating plants.
- Wind power plants with a power output greater than 25 MW - requirements and power limits have been harmonized with other technical regulations for electricity-generating plants.

In Sweden the classification is narrower, for example the smallest size is 2.5MW. The definitions for the different plant size in Sweden are [20]:

- small plants: wind power plant with 2.5MW to 25MW installed active power.
- medium plants: wind power plant with 25MW to 100MW installed active power.
- large plants: wind power plant with over 100MW installed active power.

4.2.2 Tolerance of frequency and voltage deviations

In Denmark the requirements on tolerance of frequency deviations are:

"A wind power plant must be able to withstand frequency and voltage deviations in the point of connection under normal and abnormal operating conditions while reducing the active power as little as possible."

In figure 4.4 the active power production requirements for a wind power plant with an output 11kW - 25kW and for wind power plant with an output of 25kW - 1.5MW are presented.

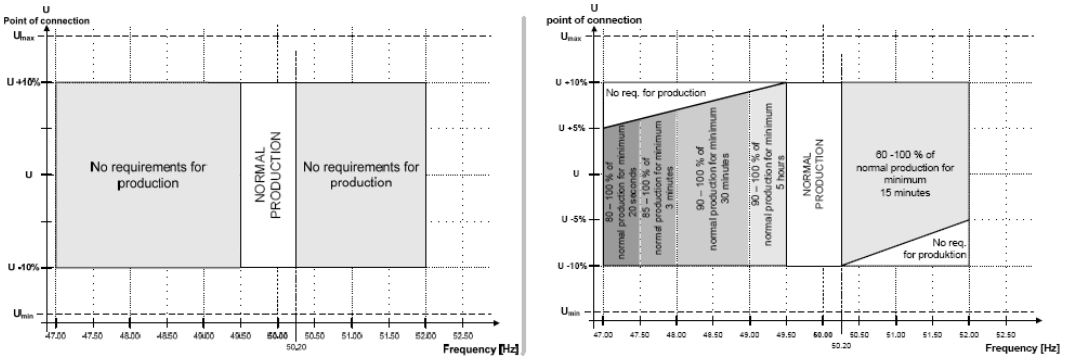


Figure 4.4: Active power production requirements at frequency/voltage variations for a wind power plant with a power output 11 kW to 25 kW (left) and 25 kW to 1.5MW (Right) [19]

As can be seen to the left in figure 4.4, small wind turbines are exempt from power production if the frequency is lower than 49.5 Hz or higher than 50.20Hz. Larger wind power plant should on the other hand be able to lower their power production according to the right subplot in figure 4.4.

When it comes to voltage deviation:

"In the point of connection a wind power plant must be able to withstand a voltage drops down to 20 % of the voltage in POC over a period of 0.5 s without disconnecting and after 1.5 s voltage in POC should be up to 90 %." [19]

This is illustrated in figure 4.5.

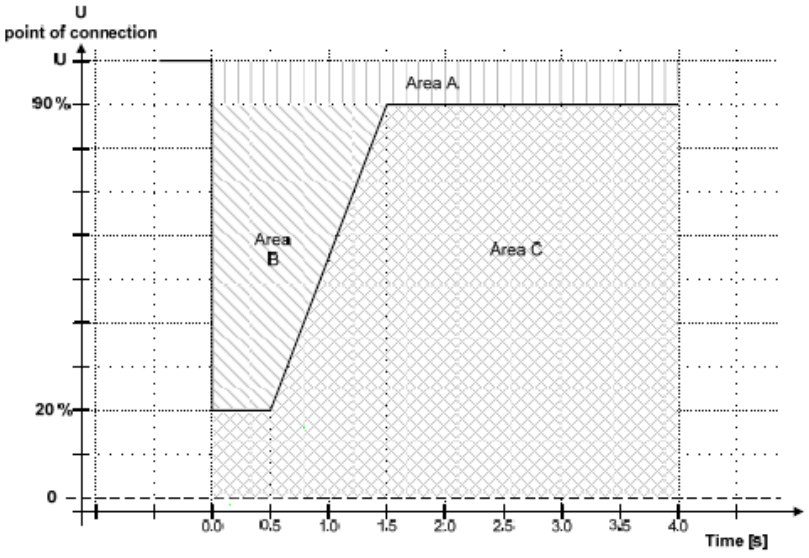


Figure 4.5: Requirements for tolerance of voltage drops for a wind power plant with greater output than 1.5MW [19]

where the different areas are:

- Area A: The wind power plant must stay connected to the network.

- Area B: The wind power plant must stay connected to the network. The wind power plant must provide maximum voltage support by supplying reactive power to the network
- Area C: Disconnecting is allowed.

comparison

Wind power plants in Sweden should also be able to keep and maintain active power production during frequency variations, however the condition is lighter and easier to achieve.

When it comes to voltage deviations in the point of connection, the requirements for different plant size can be seen in figure 4.6. Large plants should be able to withstand voltage drops down to 0% during 2.5s, follow by a leap to 25% and after 0.75s the voltage level should be up to 90% without disconnecting, which is illustrated in the top subplot in figure 4.6. The lower subplot in figure 4.6 illustrates the requirements for medium and small plants which should be able to withstand voltage drops down to 25% and after 0.25 s, the voltage level should then be over 90% without disconnecting.

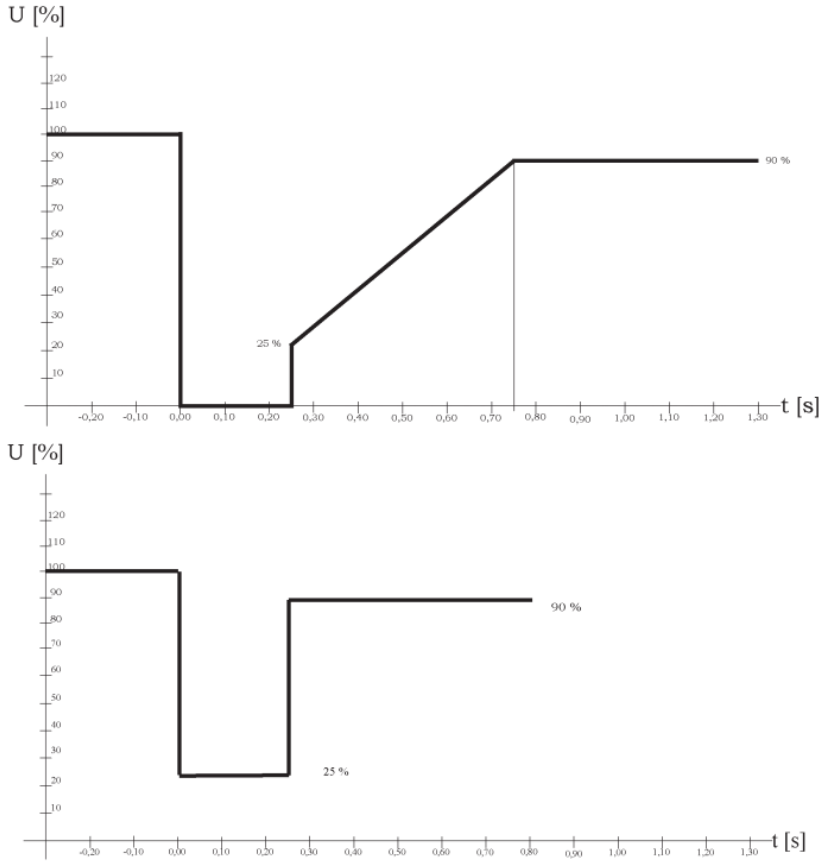


Figure 4.6: Requirements voltage drops in Sweden, the top subplot shows the requirements for large plants and the lower plot the requirements for medium and small plants[20]

4.2.3 Active Power Control

A wind power plant in Denmark should be able to control the produced amount of active power in the point of connection. The control functions listed below are requirements for wind power plants with an output larger than 1.5MW.

- **Absolute Power Control** - *"If the setpoint for the absolute production constraint is to be changed, such change must be commenced within two seconds and completed not later than 30 seconds after receipt of an order to change the setpoint. The accuracy of the control performed and of the setpoint must not deviate by more than ±2% of the setpoint value or by ±0.5% of the rated power, depending on which yields the highest tolerance."* [19]

- **Power Gradient Control** - *"If a setpoint for the power gradient constraint is to be changed, such change must be commenced within two seconds and completed not later than 30 seconds after receipt of an order to change the setpoint. The accuracy of the control performed and of the setpoint must not deviate by more than ±2% of the setpoint value or by ±0.5% of the rated power, depending on which yields the highest tolerance."* [19]

This requirement is only for wind power plants with an output larger than 25MW

- **Delta control** - *"If the setpoint for a delta production constraint is to be changed, such change must be commenced within two seconds and completed not later than 30 seconds after receipt of an order to change the setpoint. The accuracy of the control performed and of the setpoint must not deviate by more than ±2% of the setpoint value or by ±0.5% of the rated power, depending on which yields the highest tolerance."* [19]

Wind power plants should be able to automatically shutdown if the wind speed is too high. In order to not disconnect too much power at the same time and prevent instability in the electricity network, wind power plants should be equipped with an automatic downward regulation. The downward regulation band must be agreed by the enterprise who owns the grid where the wind power plant is connected to.

Comparison

In Sweden the total power from a wind power plant should only be able to control active power to a fixed reference value. The reference value should be able to be changed during runtime. The total production in the plant should be able to be reduced to 20% of the maximum produced power in 5 seconds.

When a wind power plant is disconnected because of to high wind speed, it is not allowed to disconnect more than 30MW/min, and more than 30MW/min should not be connected when the wind plant is reconnected to the grid.

4.2.4 Reactive Power and Voltage Control

A wind power plant in Denmark should be equipped with control functions that are capable of controlling the reactive power and the voltage in the point of connection. The control functions listed below are requirements which should be implemented in wind power plants larger than 1.5MW:

- **Q-control** - *"If the Q control setpoint is to be changed, such change must be commenced within two seconds and completed not later than 30 seconds after receipt of an order to change the setpoint. The accuracy of the control performed and of the setpoint must not deviate by more than ± 2% of the setpoint value or by ± 0.5% of the rated power, depending on which yields the highest tolerance."* [19]

- **Power factor Control** - *"If the power factor setpoint is to be changed, such change must be commenced within two seconds and completed not later than 30 seconds after receipt of an order to change the setpoint. The accuracy of the control performed and of the setpoint must not deviate by more than ± 2 % of the setpoint value or by ± 0.5% of the rated power, depending on which yields the highest tolerance."*[19]

In terms of voltage control, a wind power plant should be equipped with voltage control only if there is a specific agreement with the electric network company and if the power output is greater than 25MW. The condition for the voltage control is:

"If the voltage setpoint is changed, the change must be commenced within two seconds and completed not later than 10 seconds after. The accuracy of the control performed and of the setpoint must not deviate by more than ± 2% of the setpoint value or by ± 0.5% of the rated power, depending on which yields the highest tolerance." [19]

Comparison

Svenska Kraftnät has no requirements of reactive compensation for wind power plants. However wind power plants should be designed so that the reactive power should be controlled to zero. In case of controlling the voltage is stated that large, and medium wind power plants should be equipped with automatic voltage control. The voltage should be controllable to ± 5% of the plants nominal voltage level. Small wind plants, except plants with asynchronous generators connected directly to the grid, should have automatic voltage control to be able to contribute to stabilize the voltage if any disturbances occur.

4.2.5 Frequency Control

A wind power plant must not perform frequency control without having entered into a specific agreement with the electricity supply undertaking. If there is an agreement and if the wind power plant output is greater than 25MW it must be possible to set the frequency to any value in range of 50.00Hz ± 3.00Hz with an accuracy of 10 mHz.

Comparison

In Sweden there are no requirements for wind power plants to perform frequency control.

4.3 Modeling

The grid connection will be modeled in two different ways, strong grid and island grid. The strong grid is supposed to be used during normal operation, the wind power plant will be connected to a grid where the voltage level and the frequency are constant. Island operation will occur if there is a fault in the grid network and the wind power plant is cut off from the rest of the grid, i.e. the wind power plant is the only plant that generates power to the load which will lead to that the voltage level and the frequency is not constant. Both of the grid models will however have some components in common and a base model can be divided into three parts:

- A line from the inverter to the point of connection
- A transformer
- A transmission line

The line between the inverter and point of connection is modeled as an RX-line, see next section. After the point of connection a transformer is placed in order to transform the voltage to the voltage level of the grid, this since the produced voltage from the generating plant is dependent on its design and is not necessary the same as the grid voltage level. The transformer is then connected to the transmission line, modeled as an RX-line, which is connected to the point of common coupling. The thing that differ the two the lines is the length, the transmission line is significantly longer.

4.3.1 Strong Grid

The strong grid model is supposed to model the grid during normal conditions. During normal conditions the grid can be considered to have a constant frequency and voltage level, which is not affected by the output from one single plant. For this reason the grid can be modeled as a slack-bus, that is an optimal voltage source with constant amplitude and frequency, connected to the transmission line.

4.3.2 Islanding Grid

The Islanding grid model is supposed to model the system during island operation. In this scenario the wind power plant is the only power source in the system. the grid can the modeled as a load, for instant as a pure resistive load connected to the transmission line and ground connection. In order to make the load vary over time a small variable voltage source can be connected

between the resistance and ground in order to change the power consumption of the resistance.

4.3.3 RX-line

Figure 4.7: Icon for Spot's RX-line

Both the line to the point of connection and the line to the point of common coupling are modeled using the component RX-line from the Spot library. The RX-line is a simple model of a line, the impedance, Z, is described by equation 4.1

$$Z = R + jX \tag{4.1}$$

where R is the resistance and X is the reactance of the line. In Dymola the user can set the resistance and the reactance per kilometer as well as the length of the line.

4.3.4 Transformer

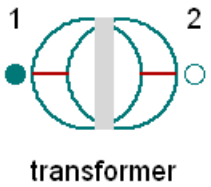


Figure 4.8: Icon for Spot's transformer

Spot has a number of complete models of transformers in different advanced levels. A transformer consists of two windings, one primary and one secondary. By using a different number of turns on the primary and secondary winding a voltage increase or decrease can be achieved. The transformer used in this project was a model which assumes ideal magnetic coupling. The model does however implement resistances and stray inductances in the windings.

4.4 Control Design

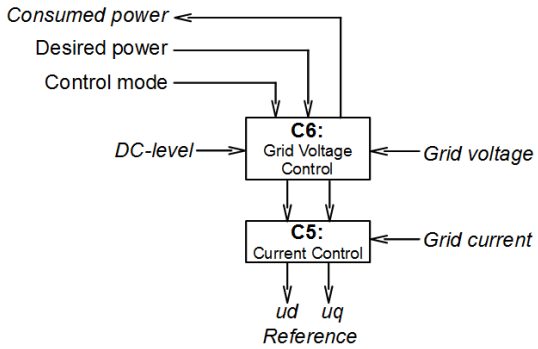


Figure 4.9: Overview of the control structure for the grid

To effectively and safely transfer power, produced by the wind power plant, to the point of connection a grid side controller was developed. The overall control scheme for the grid side can be seen in figure 4.9. The tasks of the grid side controller are listed below

- Be able to control the DC-level of the DC-link
- Be able to control the AC-level
- Be able to do reactive power compensation
- Current control, needed to control the grid side inverter

The grid current and grid voltage are measurements taken from the point of connection.

4.4.1 Grid Side Controller

An overview of the grid side can be seen in figure 4.10, where R and L are resistances and inductances of the line to the point of the connection, $e_{a,b,c}$ is the grid voltage of each phase in the point of connection, $v_{a,b,c}^{swt}$ is the voltage of each phase out from the switched inverter. The voltage equation for all three phases can be written as:

$$\begin{bmatrix} e_a \\ e_b \\ e_c \end{bmatrix} = R \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} + L\frac{di}{dt} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} + \begin{bmatrix} v_a^{swt} \\ v_b^{swt} \\ v_c^{swt} \end{bmatrix} \tag{4.2}$$

If the grid voltage is assumed to be symmetrical and balanced the zero component can be neglected when equation 4.2 is transformed into the rotating dq-plane, using the transformation presented in appendix A.1. The rotating plane rotates with the velocity w_{el}, which gives a dynamic model of the grid

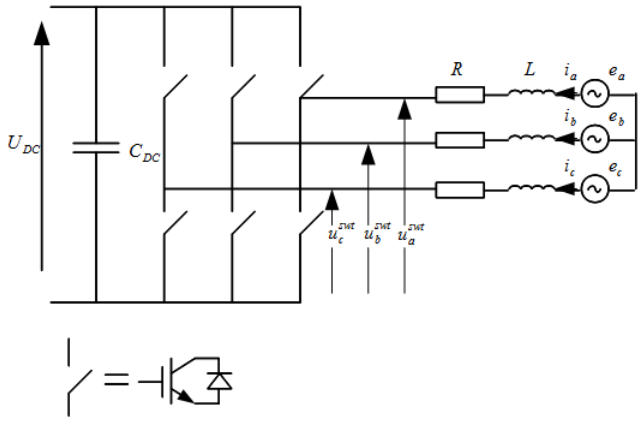


Figure 4.10: Overview of the grid side model in the abc-reference frame

connection in the dq-plane described by 4.3.

$$\begin{aligned} u_d &= Ri_d + L\frac{di_d}{dt} - w_{el}Li_q + u_d^{swt} \\ u_q &= Ri_q + L\frac{di_q}{dt} + w_{el}Li_d + u_q^{swt} \end{aligned} \tag{4.3}$$

where R and L are the grid resistance and inductance, u_d^{swt} and u_q^{swt} are inverter voltages components. The active and reactive power of the grid can be written as:

$$P = u_d \cdot i_d + u_q \cdot i_q \tag{4.4}$$

$$Q = u_q + i_d - u_d \cdot i_q \tag{4.5}$$

The grid side controller is vector controlled in the grid voltage reference frame according to equation 4.3. The grid side controller consists of a cascaded control structure, with an outer voltage loop and an inner current loop. The grid side controller is designed to be able to run in and switch between multiple modes, the different control modes are listed below.

- INACTVE - Inactive mode, this mode is active if the DC-voltage is to low or if the wind power plant should be disconnected from the grid.
- DCCONTROL - DC-level control mode, the grid side controller is ordered to control voltage level of the DC-link.
- ISLANDING - The wind power plant is not connected to a strong grid. The grid side controller is ordered to control the phase and amplitude of the AC-voltage. The wind power plant is controlling the DC-level at the DC-link.
- RCTVCOMP - Reactive power compensation mode, this mode is active at the same time as DCCONTROL, which results in that the grid side controller is controlling both the DC-level and AC-level simultaneously.

The different control modes uses the same control structure, two PI control loops in cascade, which can be seen in figure 4.11 to 4.13

- A slower outer voltage control loop.
- A faster inner current control loop, see section 4.4.1.4.

The different control modes are presented in the next sections.

4.4.1.1 DC-Level Controller

The aim of the DCCONTROL control mode is to keep the DC-link capacitor voltage steady at a reference level. The DC-level changes relative the balance between the power generated by the wind power units and the active power delivered to the grid. The grid side controller should be able to achieve the DC reference value regardless of the magnitude and direction of the power produced by the plant. Unless the reactive power compensation is active the grid side controller should only deliver active power to the grid, i.e the reactive power should be controlled to zero, according to grid codes.

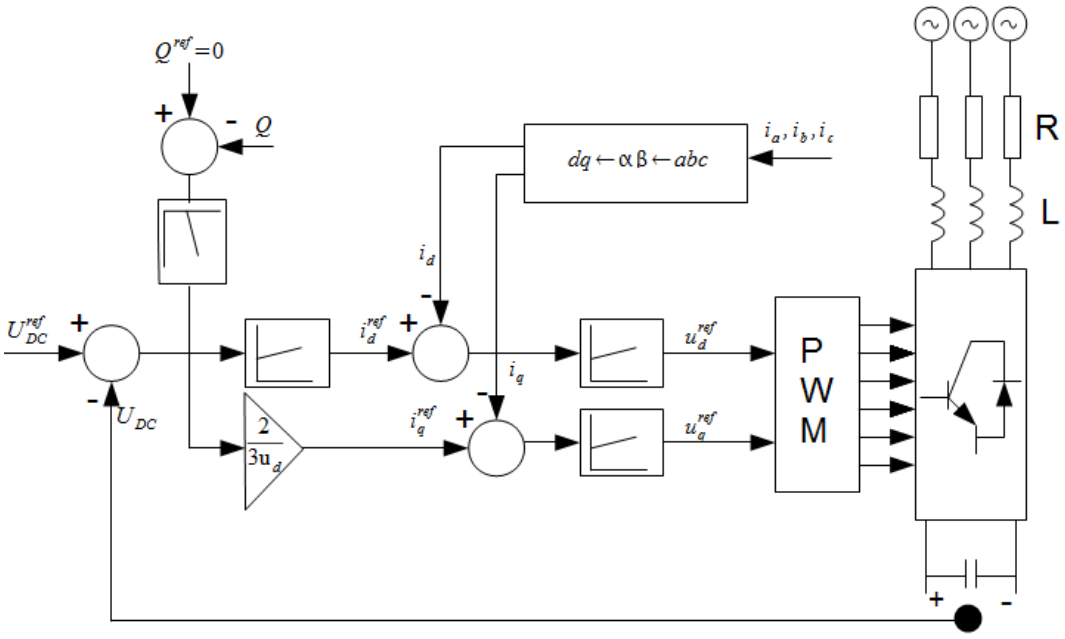


Figure 4.11: Block diagram over the control structure in DC-level mode

The outer control loop is used to control the DC-link voltage level to its reference value U_{DC}^{ref}. The outer control loop generates reference values for the direct current i_d^{ref}. This will in turn make sure that the active power is transfered from the inverter out to the grid. If the inverter is not ordered to do reactive compensation, the reactive power will be controlled to zero.

4.4.1.2 Islanding

In this mode the grid side controller's task is to keep AC-level at its reference value. The control of the voltage of the DC-link is controlled by the wind power units. The overall control structure can be seen in figure 4.12. The reference values for the direct current and the quadrant current, i_q^{ref} and i_d^{ref} for the inner current controller, is achieved from the outputs of the PI-controllers in the outer loop.

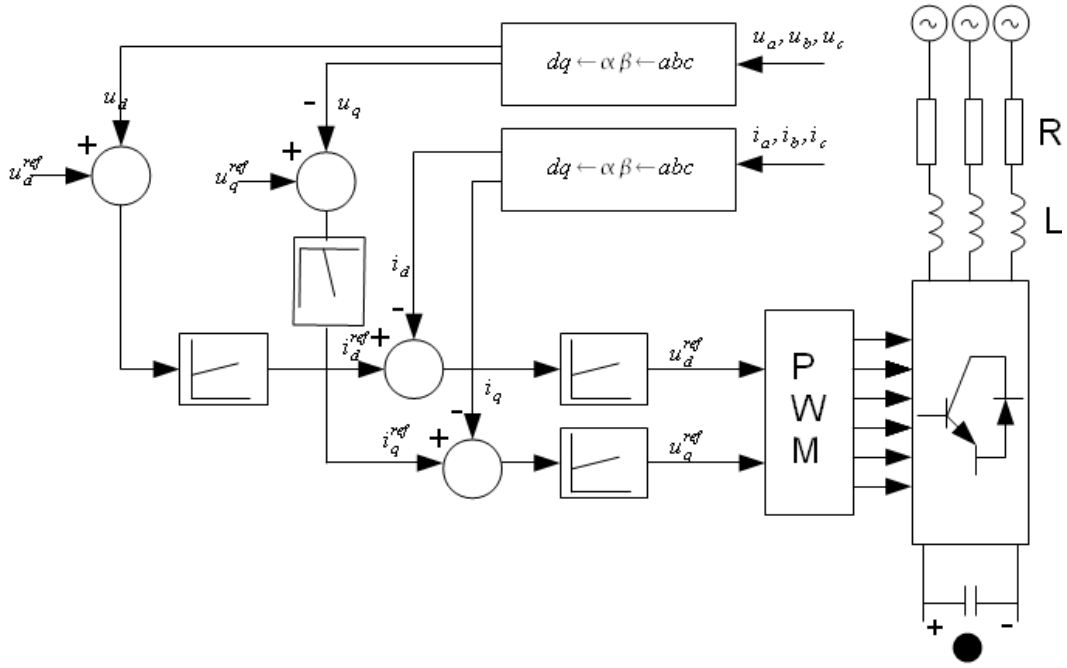


Figure 4.12: Block diagram over the control structure in island mode

4.4.1.3 Reactive Power Compensation

Instead of just controlling the reactive power exchange of the grid to zero, reactive power compensation in the point of connection can be done. The overall control structure can be seen in figure 4.13. The reactive power reference, Q_{ref}, is obtained by controlling direct voltage, u_d to its reference value. The difference between Q^{ref} and the actual reactive power, Q, is the input for the next PI-controller, which generates the unscaled reference value for the quadrant current, i_q^{ref}. The actual i_q^{ref} is scaled with $\frac{2}{(3 \cdot u_d)}$. The rest of the control is performed in the same way as in DC-control, see section 4.4.1.1

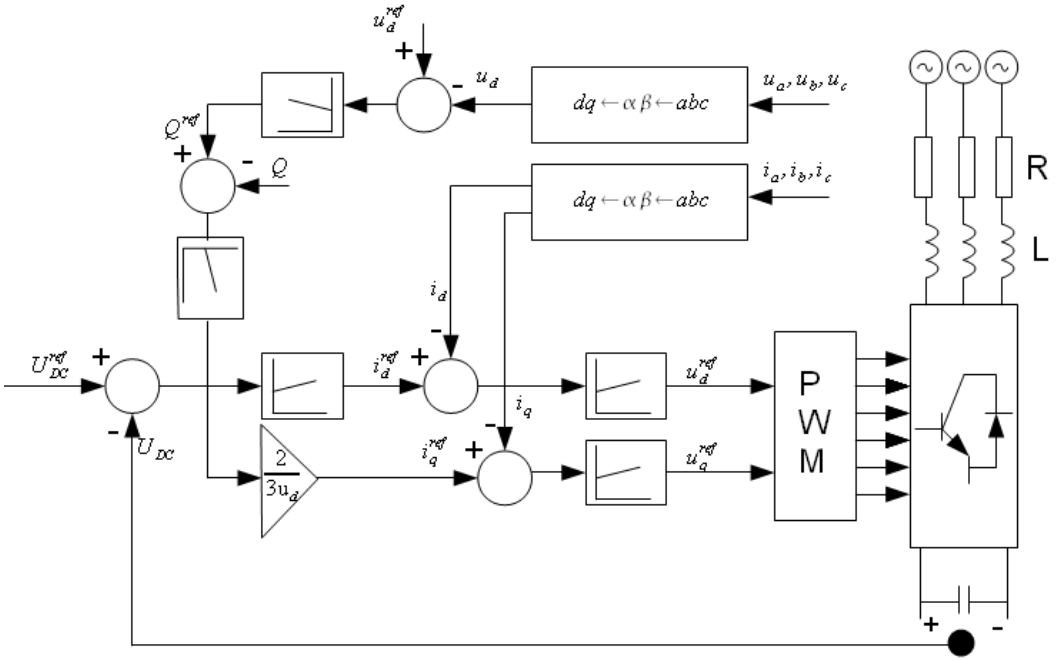


Figure 4.13: Block diagram over the control structure when reactive power compensation is active

4.4.1.4 Current Controller

The current-controller consists of two PI-controls. One controlling the direct current and one controlling the quadrature current. The current references are received from the outer loops. The currents are controlled by outputting a reference voltage to the transistor controls, modulator, which generates the PWM-signal used for controlling the transistors.

4.5 Implementation

4.5.1 Modelica

The model of the grid is constructed by using components from spot 3.2.4.1. The grid model was implemented in two ways according to 4.3. The two implementations have the following components in common:

- A line from the inverter to the point of connection
- A transformer
- A transmission line, from the transformer to point of common coupling

Both the line to the point of connection and the transmission line are implemented using the spot-component RX-line. What differs the two lines is the length, the transmission line is significantly longer. The transformer transforms the produced voltage of the wind power plant to the grid voltage.

4.5.1.1 Strong Grid

In this case the wind power plant is connected to a strong grid where the voltage amplitude and the phase are constant. This is implemented by connecting the grid line to a slack bus, which is a component with constant voltage amplitude and frequency. The implementation in Dymola can be seen in figure 4.14. The logic and the switch to the left in figure 4.14 is due to the fact that the average inverter in the Spot library is modeled as a voltage source. The effect of this is that when the inverter should be inactive, i.e. disconnected from the grid, it is instead acting as a ground connection. The inverter will be active when the DC-level of the DC-link is sufficient and close the switch.

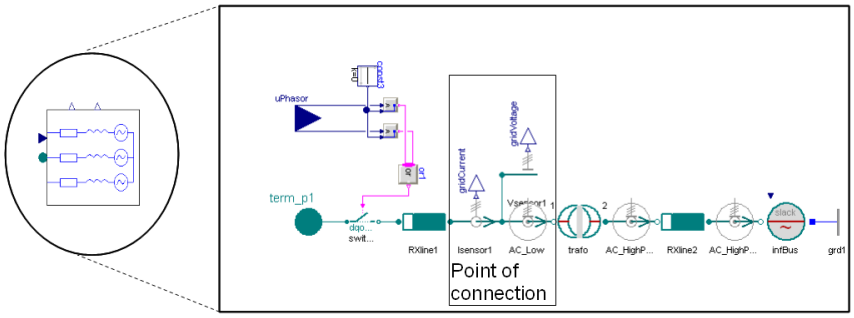


Figure 4.14: Overview of the strong grid model in Modelica, with the icon to the left and the implementation to the right

4.5.1.2 Island Operation

In Island operation the wind power plant is the only power plant that supplies the load with power. The implementation in Dymola can be seen in figure 4.15. The transmission line is directly connected to the load. The load will consume both active and reactive power and be time varying. The amplitude and frequency of the load is controlled by changing the inputs to the load before the simulation. As can be seen in the figure, after specific time a capacitance in serial with a resistance is connected in order to modify the load.

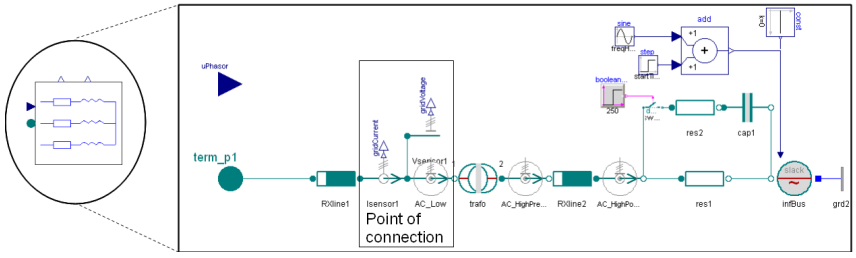


Figure 4.15: Overview of the islanding model in Modelica, with icon to the left and the implementation to the right

4.5.2 C/C++

The grid side controller is implemented in C/C++ and integrated with Dymola in the same way as the generator side controller described in section 3.5.2.1.

The grid side controller is completely implemented in the dq-reference plane.

4.5.3 Control

The class structure of the C++-implementation for the control system at the grid side has been divided into three parts:

- GridSideControllerOuter
- GridVoltageController
- GridCurrentController

The different control parts are described below.

4.5.3.1 GridSideControllerOuter

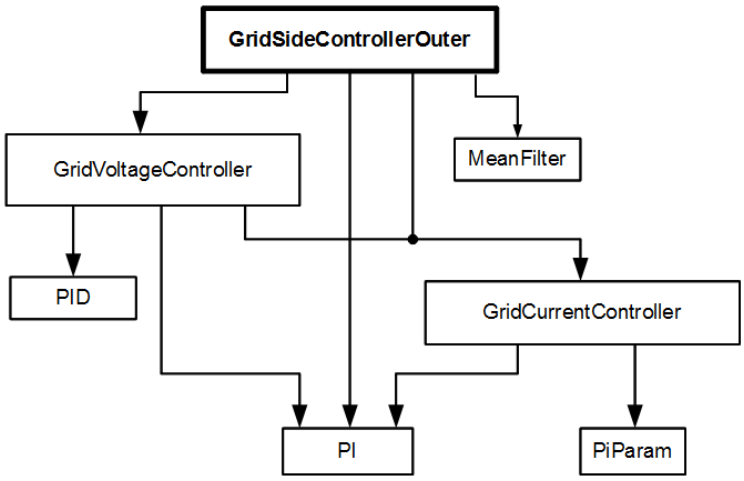


Figure 4.16: class structure of control system at the grid side

The first part of the grid side controller is called "GridSideControllerOuter". The full code is presented in appendix B.9. As can be seen in figure 4.16, the "GridSideControllerOuter"-class uses four classes, the "GridVoltageController"-class, the "GridCurrentController"-class, the "PI"-class and the "Meanfilter"-class. When a simulation starts the GridSideControllerOuter initialize the voltage controller and the current controller. The class has basically two assignments. The first assignment is to perform the overall control of the grid side. This includes handling the communication link between the grid side and the Plant Control and to monitor and provide the voltage controller and the current controller with measurements and control modes during operation. In order to solve the problem that the different control loop does not have the same sampling time, two counters was implemented. The first counter is used to keep track when it's time to update the slower outer voltage loop and calculate the mean value of the active power and send it back to the Plant Control. The second counter is used for reactive power compensation, see the

second assignment below. The class also mediate control modes to the voltage controller and the current-controller when it is time to switch between the different control modes described in 4.4.1. When there is a control mode switch to either DCLEVEL or ISLAND, the class gives order to the current-controller to update the control parameters, this since control parameters for the current controller in DCLEVEL are more aggressive and unstable in ISLAND mode. During each update-cycle the class also have the possibility to calculate the mean value of the active power and send it back to the Plant Control.

The second assignment is to calculate the reference value for the reactive power, Q_{ref}, this is only done if the control mode is the RCTVMODE. If reactive compensation is ordered the control for the quadrature voltage and current goes through three PI-controllers, see figure 4.13.

4.5.3.2 GridVoltageController

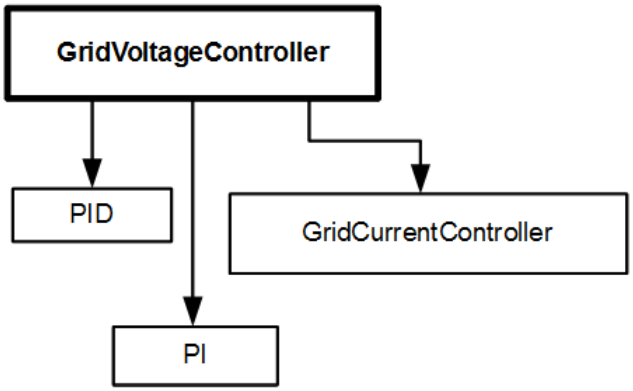


Figure 4.17: class structure of the voltage controller of the grid side

The second part of the grid side is the voltage controller called "GridVoltageController". The full C-code of the voltage controller can be seen in appendix B.10. As can be seen in figure 4.17, the voltage controller is using the "GridCurrentController"-class, PI-class and the PID-class. The voltage controller is using two PI-controllers, to control both the direct and quadrature voltage. The PID-controller is used for controlling the DC-level. The voltage controller is initialized with parameters for the PI-controllers, the PID-controller and two limits, the lowest allowed DC-voltage before the inverter should stop delivering power and what voltage level the DC-link capacitor must have in order to start delivering power out to the grid. The task for the voltage controller is to generate current references to the current controller.

When updated the *update()*-function is provided with a control mode, the actual DC-voltage, the DC-voltage reference, the grid voltages, the grid voltages references, the grid currents and a power reference of how much power the inverter should deliver. Note that the power reference is only used when the Plant Control has ordered the wind power plant to deliver a specific amount of power. The voltage controller is implemented using an internal state and the control mode sent by the Plant Control. In every update cycle the voltage

controller always check if the control mode differs from the internal state. If there is a difference the voltage controller first checks if there is enough DC-voltage on the DC-link, if so the internal state will transition to the control mode and the integrator part of the involved controllers are updated in order to achieve a bumpless control-switch.

The voltage controller uses "GridCurrentController" to get access to the present direct voltage reference. This is needed in order to scale the quadrature current reference when the control mode is operating in the RCTVMODE.

4.5.3.3 GridCurrentController

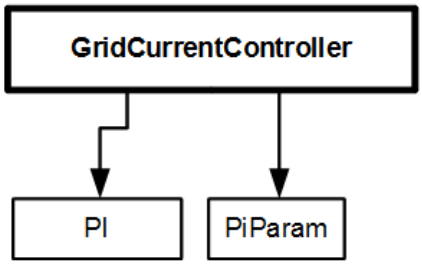


Figure 4.18: class structure of the current controller of the grid side

The current controller of the grid side was implemented into one class called the "GridCurrentController". The full C-code can be seen in appendix B.11. As can be seen in figure 4.18 the current controller is using two classes, the "PI"-class and the "PiParam"-class. The current controller uses two PI-controllers one for the direct and one for the quadrature current. The current controller is initialized with parameters for the two PI-controllers and reference values for the voltage of the grid. When updated the *update()*-function receives the current measurements, current references and what control mode the grid side controller is currently working in. The procedure is dependent of the control mode, if the control mode is INACTIVE the current controller will set the integral-part to zero and return zero in both d- and q-voltage reference. If the control mode differs from INACTIVE the PI-controllers will be updated with the present error in the currents and the output will be the output from the PI-controllers added together with reference voltage in the d- and q-component. The class has two functions *setDcLev()* and *setIsland()*, these functions are used to change the parameters for the PI-controllers when the control mode is changed to DCLEVEL respectively ISLAND.

4.6 Simulations and Results

In order to test how well our different control algorithms perform a couple of test cases was simulated, see section 4.1.1. The different test cases were designed in order to either test different aspects of the control or how well the controllers cope with a specific task. For all cases an ideal DC-source was used.

4.6.1 Grid Side Controller

4.6.1.1 Case 1 - Deliver Specific Amount of Active Power

This test was designed in order to test how well the grid side controller can deliver a specific amount of active power to the point of connection. For this test the DC-level was kept at 1000V. The power reference for this test was chosen to be a sinusoidal with a frequency of 0.01Hz, amplitude of 1000W and offset of 4000W.

The results from the simulation are presented in figure 4.19. Since the DC-source is ideal the DC-level is unaffected by the power extraction. As can be seen in the diagram the controller increases the power output until the reference is reached. When the power reference is reached the controller tries to track the reference value. The controller requires a couple of seconds to ramp up the power output before the reference is reached. The controller's task is in reality to control the DC-voltage level to its reference, which is set to 900V, and is limited by the power output reference. This means that the power reference control in reality is a limited power output. The reason to why the control is done this way is that even though a specific power output is desired it is desirable to control the DC-voltage with the inverter and not with the wind power unit. When the whole system is interconnected the requested power will be produced by the wind power units, and any overshoots in the power production, due to for example gusts, will be filtered.

The controller has no problems to track the requested power output. In order to deliver the active power the direct voltage is increased above its normal value of 230V to˜250V, how to deliver the same active power without increasing the voltage level is shown in case 2.

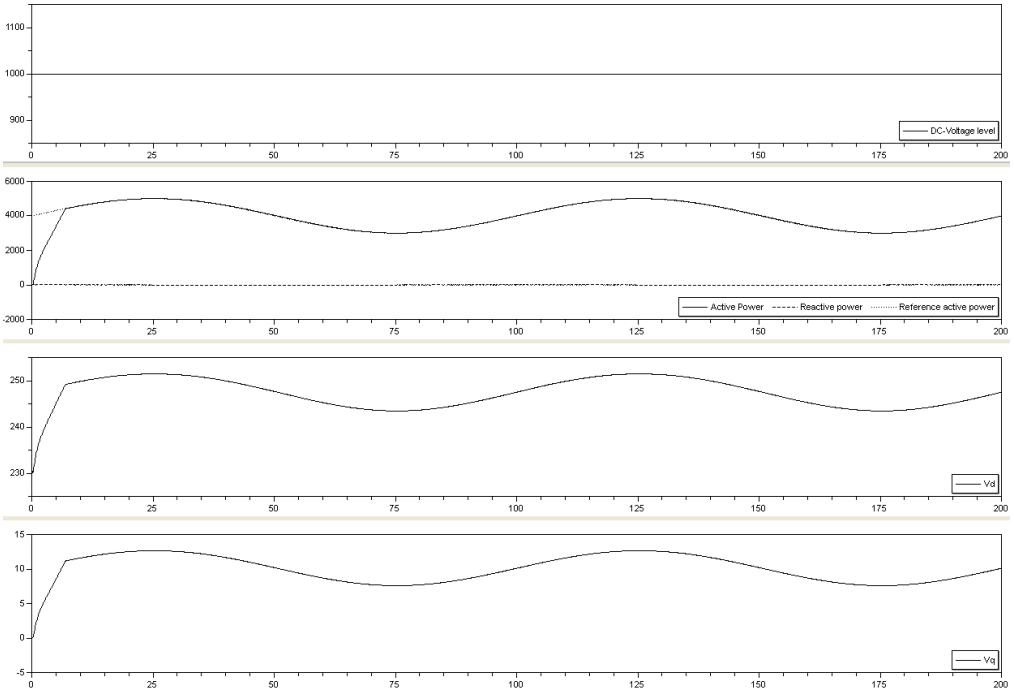


Figure 4.19: Simulation results for case 1, in the first subplot the DC-Voltage level is plotted, in the second subplot the active power reference is plotted together with the active and reactive power output, in the third subplot the direct voltage is plotted and in the fourth subplot the quadrature voltage is plotted.

4.6.1.2 Case 2 - Deliver Active Power with Reactive Power Compensation

This test was designed in order to test the reactive power compensation control. This test is identical to case 1 with the difference that reactive power compensation is performed and the simulation is extended 100s during which the power output reference is constant. During the whole simulation the DC-voltage level is 1000V.

The results from the simulation are presented in figure 4.20. As can be seen in the diagram the active power reference and output is identical to that of case 1 during the first 200s. At 200s the power reference is changed to be constant at 8000W, the controller needs some time to perform the power output increase that is need to achieve the reference.

The goal of the reactive power compensation is to keep the direct voltage at a constant 230V. When the load is constantly varying the control is not quite able to achieve the goal of keeping the direct voltage at 230V. However when comparing the direct voltage level with the voltage results in case 1 in figure 4.19 a large improvement has been made. Instead of a direct voltage level around 250V the direct voltage varies between 228-232V which is below 1% and also well below the grid codes[1], see section 4.2.4.

[1]The limits for the voltage deviation during the reactive power control is in Denmark is ±2% and in Sweden ±5%

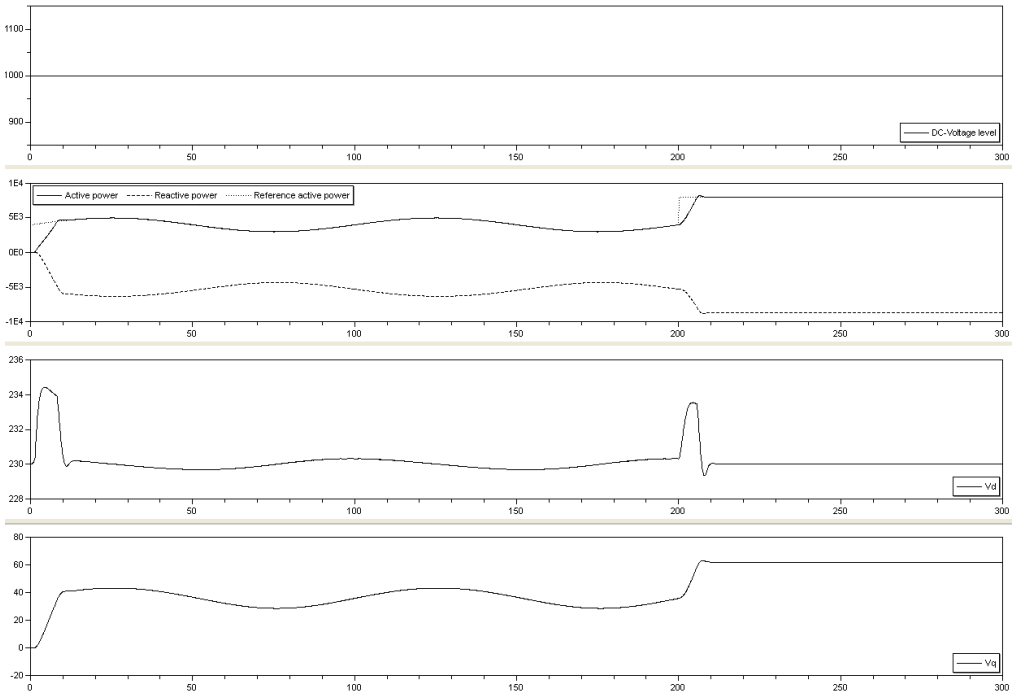


Figure 4.20: Simulation results for case 2, in the first subplot the DC-Voltage level is plotted, in the second subplot the active and reactive power reference are plotted together with the active and reactive power output, in the third subplot the direct voltage is plotted and in the fourth subplot the quadrature voltage is plotted.

4.6.1.3 Case 3 - Test Island Operation

This test was designed in order to test the island operation mode. This test was divided into three parts. The first part tests the island operation with a constant load, the second part tests the island operation with a varying load and the third part tests the island operation with a constant load using a switched inverter.

Constant Load

This test case was designed in order to test the basics of the island operation. The results from the simulation are presented in figure 4.21. The constant load is consisting of a simple resistance directly connected to the ground, this is generating a load of a constant 2000W. The grid controller successfully manages to power up the grid and after an initial transient it does not have any problems to maintain the direct and quadrature voltages. One possible reason for the small initial transient is explained in the last part of this case.

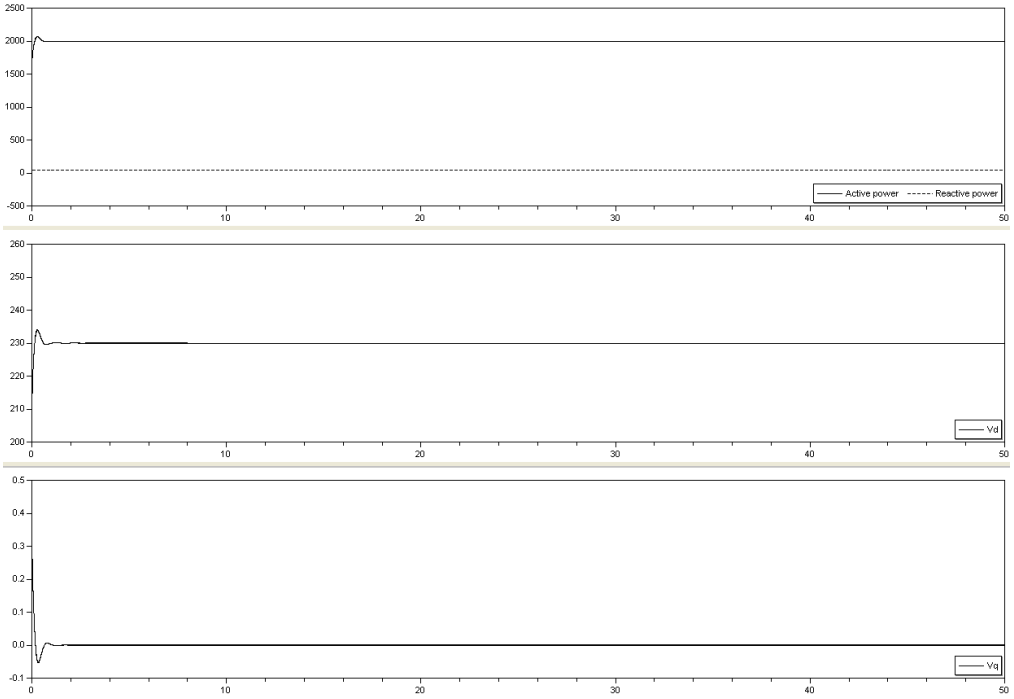


Figure 4.21: Simulation result for the first test of case 3, in the top subplot the active and reactive power are plotted, in the middle subplot the direct voltage is plotted and in the bottom subplot the quadrature voltage is plotted.

Varying Load

This test case was designed in order to test the performance of the Island operation by applying a varying load, see section 4.5.1.2. The results from the simulation are presented in figure 4.22. The load was chosen to vary as a sinusoidal with a frequency of 0.1Hz, an amplitude of approximately 600W and an offset of approximately 1600W. Also in this case the grid controller successfully powers up the grid, and after some initial transients the direct and quadrature voltage level is kept around their reference values. The power variations are reflected in the direct and quadrature voltages as a very small oscillation around their reference. The oscillation is, however, well below the limits set by the grid codes. One possible reason for the initial transient is explained in the last part of this case.

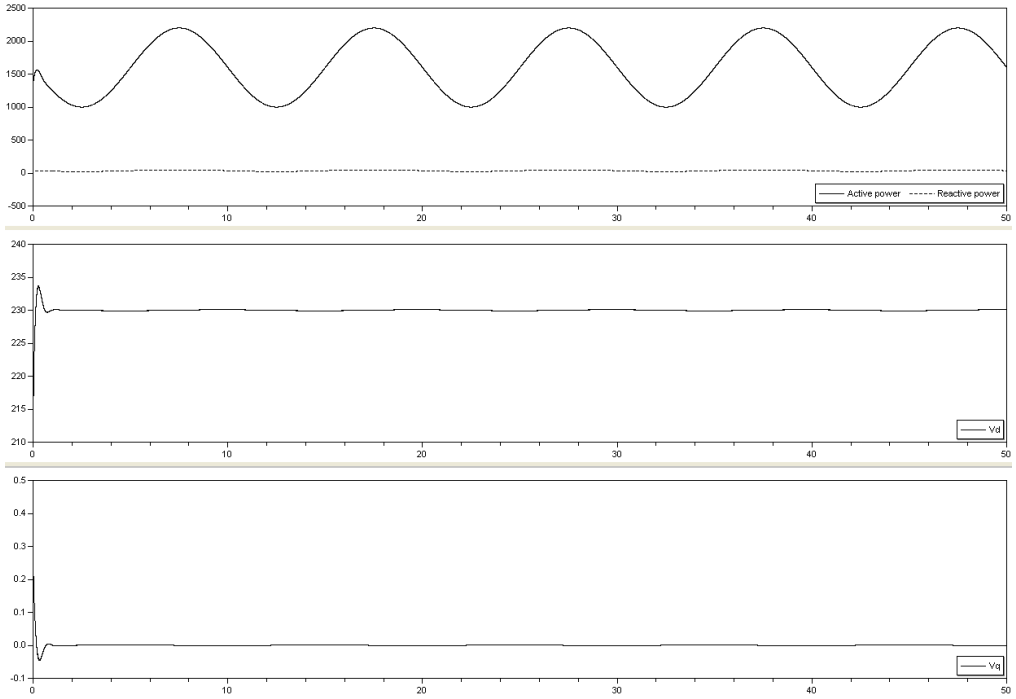


Figure 4.22: Simulation result for the second test of case 3, in the top subplot the active and reactive power are plotted, in the middle subplot the direct voltage is plotted and in the bottom subplot the quadrature voltage is plotted.

Constant Load Using a Switched Inverter

This test case was designed in order to assure that the average model is not deviating largely from the switched model. The general results from the simulation are shown in figure 4.23, the calculated direct and quadrature voltage are shown in figure 4.24 and a close-up of the switched voltage output from the inverter together with the filtered voltage level. In this case the model is simulated using a switched inverter model powering the same grid as in the first test of this case for one second. As can be seen in figure 4.24 the control does a good job in controlling the direct and quadrature voltage to 230V respectively 0V despite the disturbances introduced by the switching.

As can be seen in figure 4.23 and 4.24 the oscillations both in the active power and voltage levels are no longer present. The reason for this is most likely cause of the fact that in this simulation both the active power and voltage levels are not changed instantly.

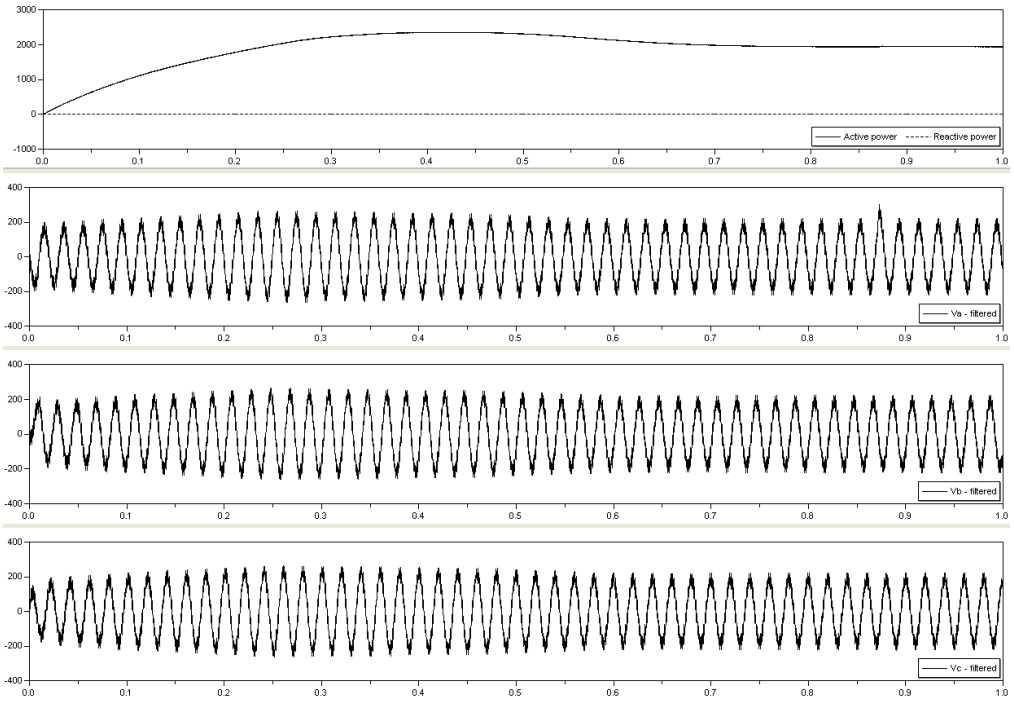


Figure 4.23: Simulation result for the third test of case 3, in the top subplot the active and reactive power is plotted, in the second subplot the filtered voltage level in phase a is plotted, in the third subplot the filtered voltage level in phase b is plotted, in the fourth subplot the filtered voltage level in phase c is plotted.

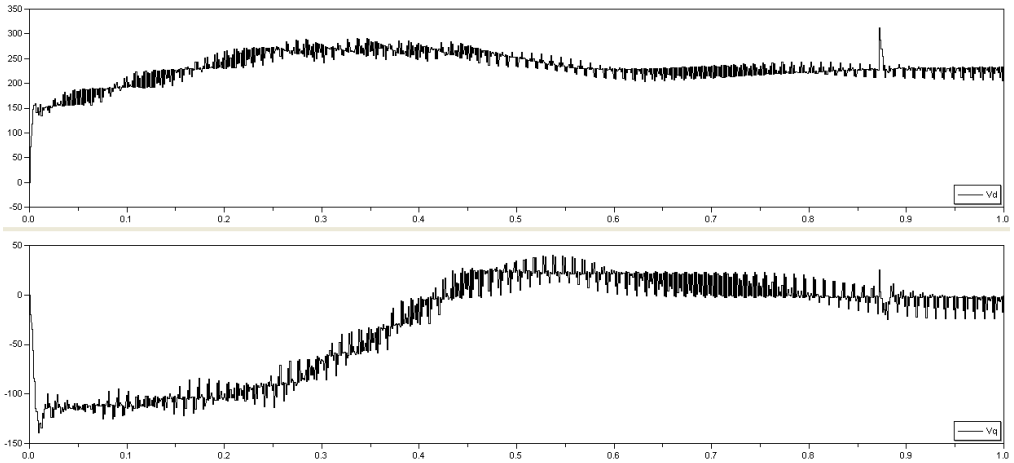


Figure 4.24: Simulation results for the third test of case 3, in the first subplot the calculated direct voltage is plotted and in the second subplot the calculated quadrature voltage is plotted

The model was simulated in the abc-reference frame, and in order to control the system the currents and voltages had to be converted to the dq0-frame. In order for the park-transformation to work the switched voltage shown in figure 4.25 had to be filtered. The filtering of the measurements was in this case performed by a continues-time low-pass Butterworth filter of the first degree.

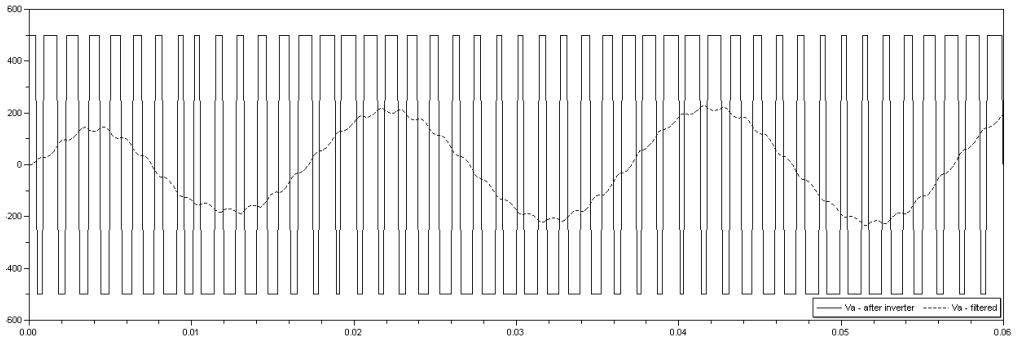


Figure 4.25: Simulation result for the third test of case 3, in this plot the actual voltage level directly after the inverter is shown together with the filtered voltage level measured.

4.6.1.4 Case 4 - Total Case

This test was designed to both test the transition to reactive power compensation and how the control deals with a varying voltage source. The results from the simulation are presented in figure 4.26 and 4.27.

During the simulation the DC-voltage level was chosen to vary as a sinusoidal with a frequency of 0.01Hz, an amplitude of 150V and an offset of 1200V. During the first 100s the grid controller is supposed to output the requested active effect and the last 100s the grid controller should output the requested active power while at the same time keeping the direct output voltage at 230V. As can be seen in figure 4.26 the control of the active power output is unaffected by the change in the DC-voltage level. When the transition to reactive power compensation occurred a small deviation in the active power output from its reference is present, which is expected since the quadrature current is used to control the reactive power which in turn increases the active power. As can be seen in figure 4.27 the control successfully decreases the direct voltage to around 230V, for the same reason as described in case 2 the voltage level is varying slightly around 230V. The control decreases the voltage level to below 234.6V in well below 10s which is required by the grid codes, see section 4.2.4. Apart from this the grid codes regarding Q-control states that when the Q-setpoint is changed the control should be initiated within two seconds and performed within 30 seconds, and the output power should not deviate more than $\pm 2\%$. However a control mode in which a specific reactive output power could be requested was never implemented, but the controls for performing this specific task was, and when studying the reactive power reference and output in figure 4.26 it is quite reasonable to assume that the control would manage to achieve also these goals.

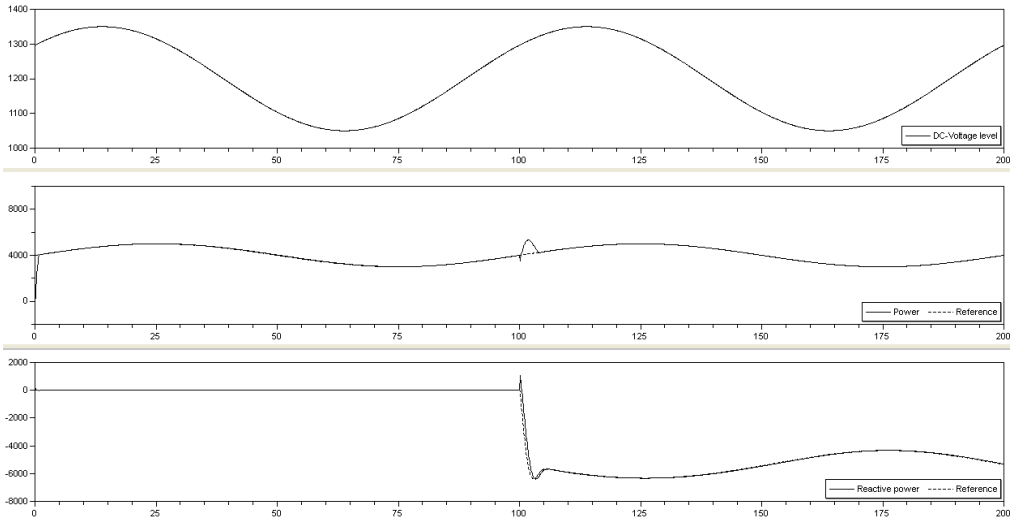


Figure 4.26: Simulation results for case 4, in the top subplot the DC-Voltage level is plotted, in the middle subplot the active power reference is plotted together with the active power output, in the bottom subplot the reactive power reference together with the reactive power output are plotted.

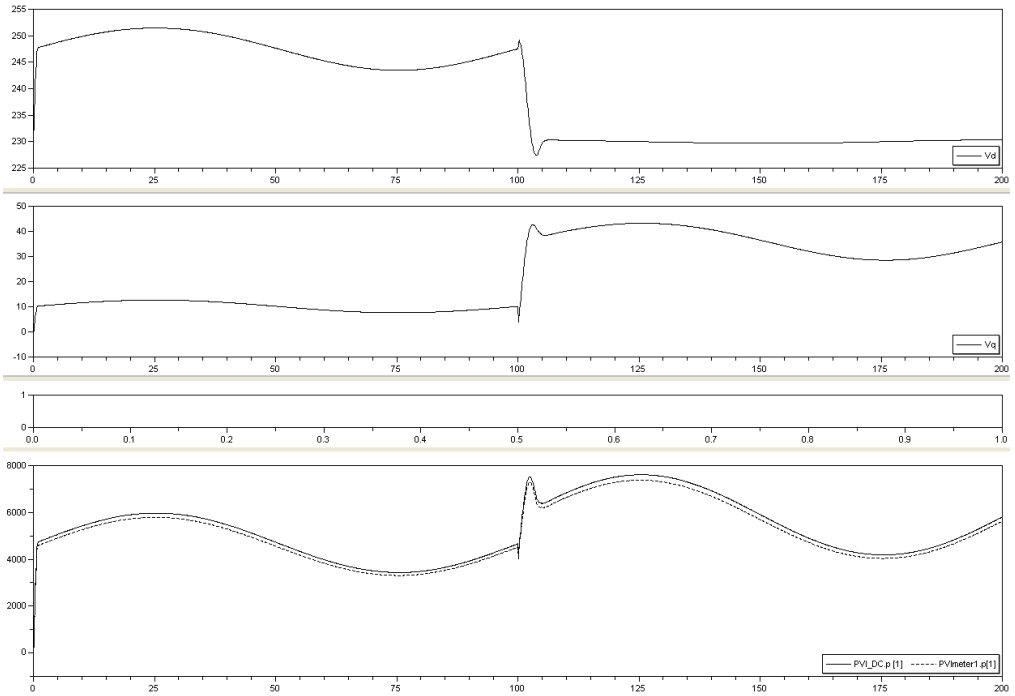


Figure 4.27: Simulation results for case 4, in the top plot the direct voltage is plotted and in the bottom subplot the quadrature voltage is plotted.

4.6.2 Losses

In figure 4.28 the power output in different key parts of the models is plotted in order to show the losses in the system. As can be seen in the figure the system is

exposed to losses in both the inverter and the grid transferring the power to the connection point. In the figure it can also be seen that the losses in the grid is greatly increased by the reactive power compensation, this due to the increase of the reactive power absorbed by the system. The losses caused by the inverter is during the whole simulation about ∼3% and the losses caused by the power transfer to the connection point without the reactive power compensation is ∼12% and with the reactive power compensation ∼30%. In total the overall losses in the system is ∼14% without the reactive power compensation and ∼31% with the reactive power compensation turned on.

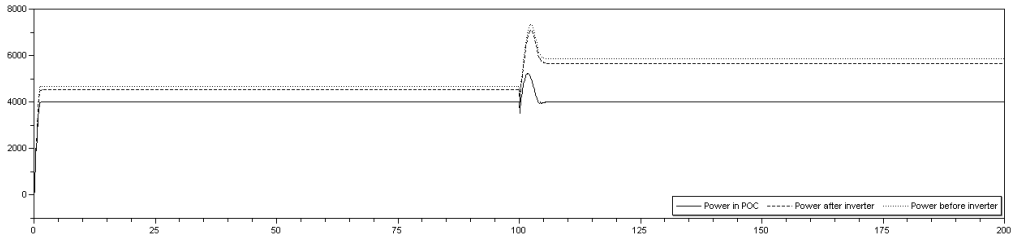


Figure 4.28: This plot is illustrating the losses in the grid connection. In the plot the power flow on the grid side is shown, the top curve shows the active power extracted from the DC-grid, the middle curve the active power delivered to the AC-grid and the bottom curve the active power at the connection point.

4.7 Discussion

As demonstrated in case 3 and shown in figure 4.23 to 4.25 the switched inverter model does not differ a great deal from the averaging inverter model. The largest difference between the models is the effect caused by the switching shown in figure 4.25, usually a filter is implemented after the inverter in order to filter its output to the grid, however due a lack of time this was not implemented in this project. In order to be able to run the simulations anyway continues filters from the Modelica standard library was applied directly on the measurements. The largest modeling error introduced by the averaging model is due to the fact that the implementation of the average inverter model is based upon a model of an ideal voltage source. The implication of this is mainly manifested during the initial transients, which can be seen in the islanding simulations of case 3. Since the model output is based on an ideal voltage source the voltage level is changed instantly when requested. In reality, and the switched model, the voltage, and current, is increased rapidly but not instantly, this is most likely also the cause of the initial transient of the islanding control.

The biggest concern regarding the grid connection is the modeling of the different grids. The fact that we could not find any transfer grid data which we could use to set up the models meant that the parameters of the transfer grid was made according to a qualified estimate. However the goal of this project was not to model the grid accurately, but to model and control the

wind power plant as a system. The possible implications of a change in the grid connection to a more accurate model would most likely only be a retuning of the grid controller's parameters.

Overall the performance of the grid control, tested in this chapter, is good, however the aspect of controlling for instance the active and reactive power output is not a complicated task since they are easily calculated using equation 4.6 and 4.7. The control has no problems tracking both the active and reactive power output simultaneously, however the main operation of the grid controller, DC-level control, is tested in chapter 5 since the entire wind power plant needs to be operational for that to work properly.

$$P = u_d \cdot i_d + u_q \cdot i_q \tag{4.6}$$

$$Q = u_q \cdot i_d - u_d \cdot i_q \tag{4.7}$$

Chapter 5

Wind Power Plant

In this chapter a complete wind power plant consisting of one or several wind power units, back to back inverters and grid connection will be taken into consideration. The modeling in this chapter is quite minor since all the different parts of the plant has been modeled and tested in chapter 3 and 4, thus the modeling essentially consist of connecting the different models. The control part in this chapter mainly consists of managing control, however the DC-control done by the wind power units when in island operation will also be closer examined. The managing control consists of ordering in which mode the grid connection and individual wind power units shall be operating, all depending on in which mode the plant is ordered to operate and the current power output potential from the different units.

In the end the complete system, with control algorithms, will be tested and evaluated against the grid codes described in section 4.2.

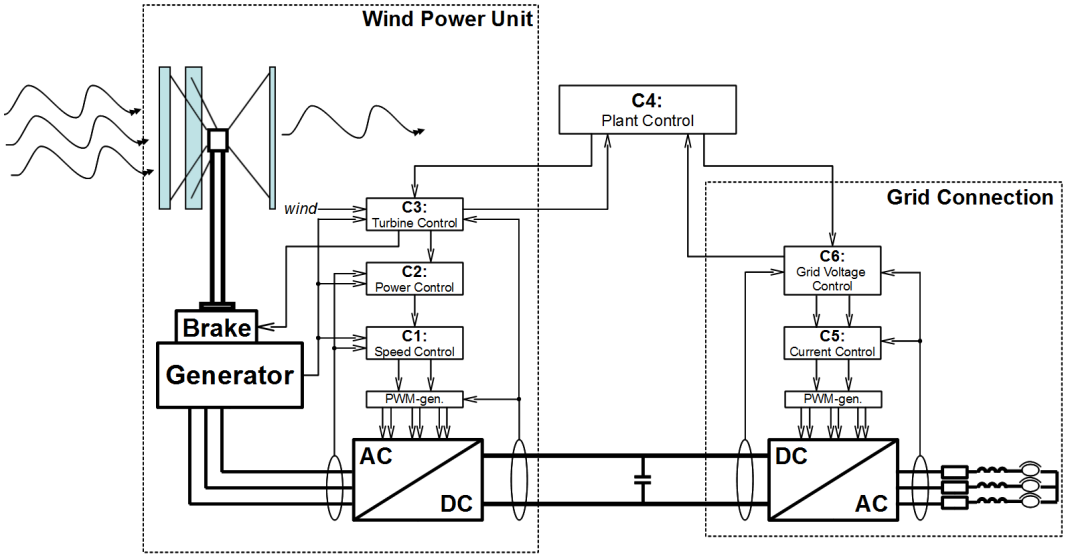


Figure 5.1: Schematic overview of a wind power plant with one wind power unit.

5.1 Objectives

The overall objective of this project is to model and control a small wind power plant, a wind power plant is here defined according to figure 1.1 as one or several wind power units connected by a common DC-link to the grid connection. In this chapter the different parts discussed in chapter 3 and 4 is to be connected and controlled as a whole. Assuming that all individual control algorithms of the grid connection and wind power units is functioning properly a control algorithm should be designed with the objective to manage the wind power plant. The system which is to be modeled, and controlled, is presented in figure 5.1 as one, alternatively three, wind power units connected by a common DC-grid to a single grid connection. The control designed in this chapter should be able to achieve a number of different goals listed below.

- The control should be able to order shut down of the plant at any time.
- The control should not try to turn on a wind power unit when the turbine control, see section 3.4.1, has ordered a shut down due to ill wind conditions.
- The control should be able to order the plant to operate at maximum power.
- The control should be able to order the plant to operate in delta mode.
- The control should be able to order the plant to deliver a specific active power to the grid.
- The control should be able to order the plant to power up a small part of the grid, i.e. order the plant to operate in island mode.

- The control should be able to order the grid connection to do reactive power compensation when in Maximum Power, Delta mode or delivering a specific amount of active power.
- When ordered to deliver a specific amount of power to the grid the control should divide the power-load amongst the wind power units controlled.
- When in island mode the control should be able to decide how many wind power units that should be active depending on the load and potential power output from the different units.

As a part of the project the simulation results should be evaluated and compared to grid codes, see 4.2, and the effect of no longer having an optimal DC-voltage source should be evaluated.

5.1.1 Test Cases

In order to test the control algorithm which will be developed in this chapter a number of different test cases should be evaluated. The test cases have been constructed so that they will test a specific part of the control algorithm and/or different realistic control scenarios. Apart from testing the managing control algorithm designed in this chapter, the DC-control algorithms for the wind power unit and grid connection developed in chapter 3 and 4 should also be tested and evaluated. This since they both require the whole wind power plant to be operational in order for any significant tests to be done. The tests have been divided into two different sections, the first section of tests should be run with a plant using only one wind power unit. This in order to test the different DC-control algorithms and the second section of tests should be run with a plant using three wind power units.

One Wind Power Unit

1. Test running a plant at maximum power with varying wind speed.
2. Test running a plant in island operation using a constant load.
3. Test running a plant in island operation using a varying load.

Three Wind Power Units

1. Test running the plant in delta-control mode with varying wind speed.
2. Test running the plant with the objective of delivering a specific amount of active power to the grid.
3. Test running the plant at maximum power switching between reactive compensation on and off, and order a complete shut down at the end.
4. Test running the plant in island operation using a varying load and varying wind speed.

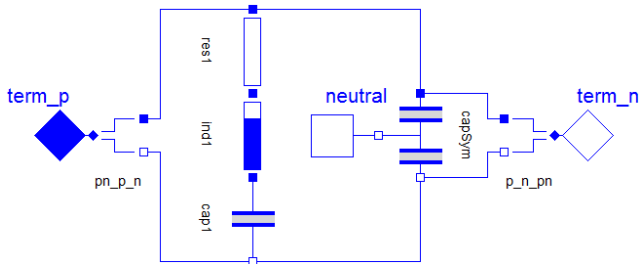


Figure 5.2: Schematic picture of the DC-link used in the system.

5.2 Modeling

The modeling in this chapter consists of connecting the different models developed in the previous chapters into two different systems. The first using one wind power unit and one grid connection and the second using three wind power units and one grid connection. Both of the models utilize a DC-grid in order to connect the wind power units with the grid connection.

5.2.1 DC-Grid

In order to interconnect all the wind power units to the grid connection and decouple the rotational speed of the wind turbines from the frequency of the grid an intermediate DC-grid was chosen to be used. The DC-grid was modeled using the Spot-library's component DC-link. The DC-Link model in Spot is a simple model of a DC-link equipped with symmetric DC-link-capacitors and a simple RLC-filter, see figure 5.2.

5.3 Control Design

In order to achieve the goals stated in section 5.1 an overall control algorithm managing the different components involved in the system is needed. The control's goal should be to achieve the control command received by the customer. The control structure developed for this task consists of one single controller, the Plant Control.

5.3.1 Plant Control

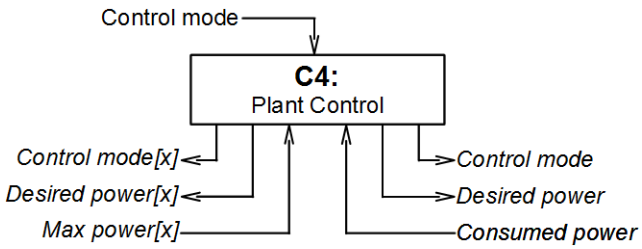


Figure 5.3: Schematic control structure of the plant controller.

The Plant Control's task is to make sure that the wind power plant operate according to the directives it has received, a schematic figure of the control structure proposed in this project is shown in figure 5.3. The Plant Control's main task is supposed to be managing of the power plant, no actual control is supposed to be performed here. The managing control should be able to make the power plant operate in the different modes listed below.

- SHTDWN - Shutdown or inactive mode.
- DESEFF - Desired power.
- MAXEFF - Maximum or nominal power.
- DELTA - Delta mode.
- ISLAND - Island mode.
- RCTVCMP - Reactive Power Compensation.

The management of the plant consists of providing information about in which mode the different system should operate. In most cases it is just a matter of passing down the requested mode to the grid connection and the wind power units.

The shut down mode only requires the controller to order all the system parts to shut down.

In the desired power mode the controller should compensate for the internal losses, decide how many wind power plants is needed in order to provide the power and divide it amongst them. The load should be divided so that a unit with a higher potential power output should get a proportionally bigger part of the load.

In the maximum, or nominal, power mode the controller should order all the units to output nominal power.

In the delta mode the controller should order all the wind power units to operate in delta mode.

When in the island mode the control should order the grid connection to operate in island mode, the Plant Control should also make a decision of how many wind power units that are needed in order to maintain an acceptable voltage level of the DC-grid.

There are many different possible ways for the Plant Control to decide how many wind power units that should operate when in island mode. One of the methods could be to observe the current DC-level as well as how much energy that is currently produced. Another method could be to measure the power consumed by the grid and use this as a basis to decide how many units that should operate. Apart from deciding how many wind power units that should be active the controller also has the possibility to order some of the units to operate at Maximum Power and some to operate in DC-control mode. The method finally chosen in this project was the latter of the two, to use information about how much energy that currently is consumed by the grid as well as the maximum potential energy available from the individual wind power units. The possibility of using some wind power units in DC-control and some at Maximum Power was not explored. In order for the system to operate in this way the grid connection controller is required to provide information about how much active power that is consumed by the grid at the moment.

5.3.2 DC-Control

When the system is operating in Island mode the responsibility of keeping the DC-level at an acceptable level is moved from the grid connection to the wind power units. In order to be able to maintain a good control of the DC-level in this situation a control algorithm was developed which, not only uses the wind power units but also, utilizes the entire wind power plant. The main reason behind why this decision was made was that there was no apparent reason not to utilize the information about how much energy that is consumed by the grid. Instead of neglecting this information and control the DC-level with a PID-controller controlling the output power from the individual units it was decided to use this information in order to control the DC-level.

The information about how much energy that is consumed by the grid is obtained by the grid connection controller through equation 5.1. The power measurement is then filtered in order to remove any medium high to high frequency changes in the consumed power. This since the process of controlling the output power from the wind power units is a relatively slow process and the medium high to high frequency changes in the consumed power only have a minor effect on the DC-level.

$$P = i_d \cdot u_d + i_q \cdot u_q \tag{5.1}$$

The information about the energy consumed by the grid is then sent to the Plant Control which first uses the information in order to decide how many wind power units that should be operating. The Plant Control is also able to compensate for the internal losses and divide the energy consumed by the grid to the units in much the same way that is done when operating in the desired power mode, see section 5.3.1. The desired power which the wind power units receive when in DC-control mode is then used as an offset in the DC-control algorithm, see section 3.4.2.

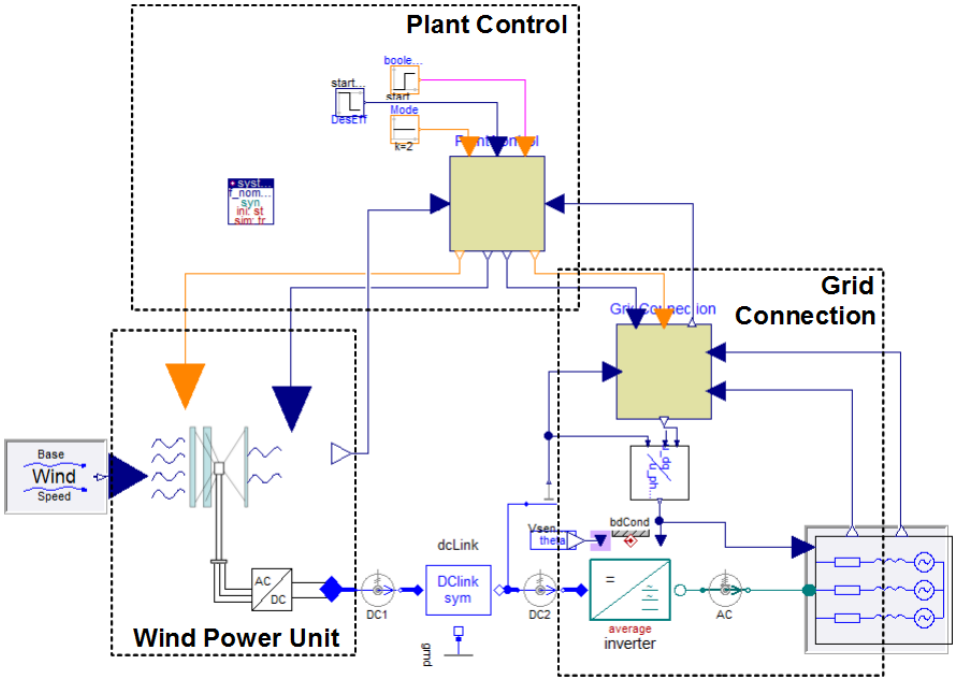


Figure 5.4: Top view of the implementation of a wind power plant with one wind power unit.

5.4 Implementation

5.4.1 Modelica

Two different Modelica models was developed, however the only difference between the two is the number of wind power units that is connected to the DC-grid, see figure 5.4 and 5.5. The models was constructed by connecting one or more wind power unit models to a model of a DC-link, the DC-link was then connected to the grid connection model. In reality it is unrealistic to assume that the wind acting on the different wind power units is identical, in order to reflect this in the model the wind model was moved to inside the wind power unit model. Apart from exposing the different wind power units for a unique set of wind-noise and gusts the units can also be exposed by different base wind speeds. By modeling each of the different units with a separate base wind speed model the different units can be exposed to different wind profiles. Another approach would be to allow the different units to have the same basic wind profiles at different intensities, i.e. use one base wind-block and add a gain in front of each unit.

5.4.2 Control

The Control of the wind power plant was implemented using C++ in a single class called *PlantControl*, the overall design of the control is discussed in section 5.3.

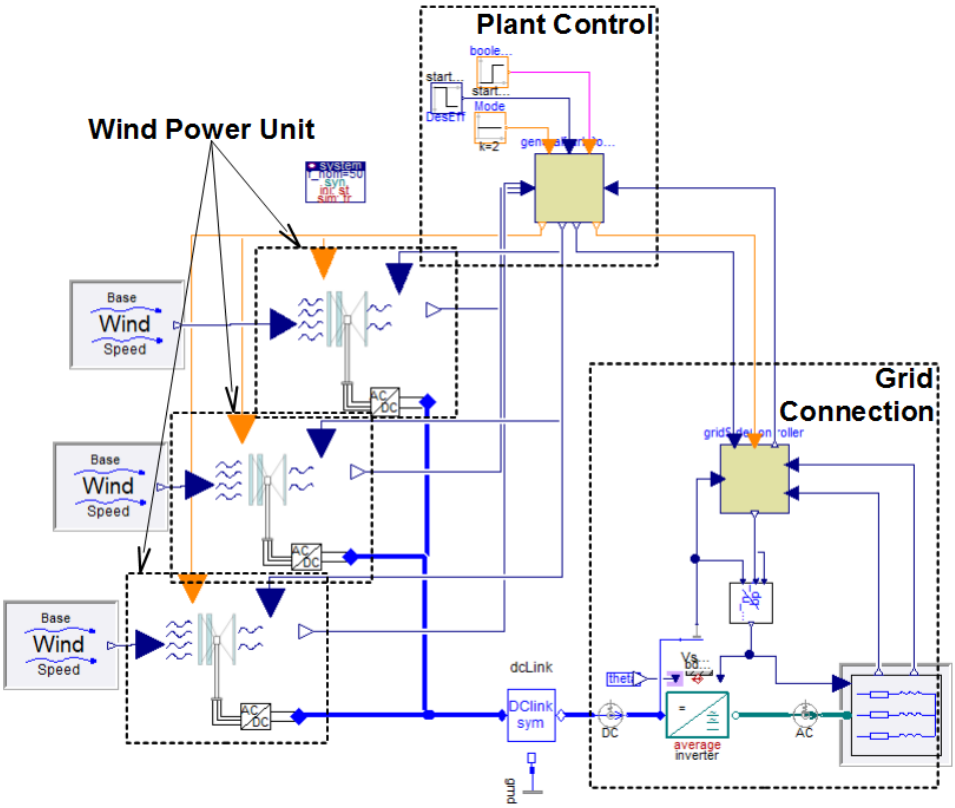


Figure 5.5: Top view of the implementation of a wind power plant with three wind power units.

5.4.2.1 Plant Control

The Plant Control was implemented in order to manage the wind power plant. The C++-code in whole is presented in appendix B.12. The implementation of the Plant Control is partially quite trivial and consists of forwarding the control received.

In order to store in which mode the different wind power units should operate an array structure was utilized, the maximum length is in this case set to 100 units, but can easily be increased to a much higher value. The Plant Control is initiated with the number of units in the system, an estimation of the internal losses in the system, the sampling period and lastly a cool-down time. The cool-down time is used in the initialization of the island operation in order to allow the power measurements from the grid connection control to stabilize before a decision is made regarding the number of units that should be operating. All the units is initialized in the "ShutDown" mode and the desired power is set to zero. If the number of units entered at the initiation would be higher than the maximum allowed units the number of plants is set to -1, this since this is the only available way to indicate an error. The initial grid connection mode is set to "Inactive".

The control is updated by providing the requested mode, the desired effect (only used if the requested mode is "Power Tracking"), the power consumed by the grid (only used if the requested mode is "Island"), a flag indicating that reactive power compensation should be performed and the current maximum power available at the different units. A switch structure is used in order to check which mode is requested, and perform the necessary tasks. If the requested mode would be "Shut Down" the controller sets the grid controller's mode to "Inactive" and calls the internal function *setAllUnitMode()*. If the requested mode would be "Max Effect" or "Delta" the grid controller's mode is set to "DC-Level" or "Reactive Compensation" depending on if the reactive compensation flag is set, also in this case the internal function *setAllUnitMode()* is called.

The *setAllUnitMode()*-function iterates through all the available wind power units and sets the appropriate mode and desired power, in the case of the "SHTDWN" mode the desired effect is set to zero and the mode to "Shut Down". If the mode would be "NOMEFF" the function sets the mode to "Nominal Power" and the desired power to the maximum power, if the maximum power is greater than zero. When the maximum power is zero or less the mode is set to "Shut Down" and the desired power to zero. Finally if the mode entered would be "DELTA" the function sets the control mode to "Delta" and the desired power to 80% of the maximum power[1] if the maximum power is greater than zero. When the maximum power is zero or less the mode is set to "Shut Down" and the desired power to zero.

When the requested mode into the *update()*-function is "DESEFF" the control sets the mode of the grid connection to "DC-Level" or "Reactive Com-

[1]This is done for good measure, the actual reference that will be used is set in the Turbine Control.

pensation" depending on if the flag for reactive compensation is set or not, calls the internal function *splitDesEff()* and sets the variable *gridEff* to the desired power. The variable *gridEff* is then sent to the grid side controller which uses it in order to deliver the desired power, if possible. When in any other mode then "Desired Effect" the *gridEff*-variable is set to -1, this is done in order to signal to the grid side controller that no particular output power is desired.

The *splitDesEff()*-function's task is to calculate how many wind power units are needed for the requested power output and to split the desired effect amongst the wind power units. The function starts by running an algorithm which calculates the number of units which are needed to be active. The algorithm starts by defining two new variables, one for keeping track of how many units that should be activated and one for keeping track of how high their potential power output is. The algorithm iterates through the available units and increments the potential power and number of units needed until either all the units are included or a potential maximum power of 20% higher than the requested power is obtained. The function is now ready to set the modes for the different units which are done by iterating through the units chosen to be active. If the unit's maximum power output is more than zero its mode is set to "Power Tracking" and its power reference is set to a fraction of the requested power relative to the power potential of the unit, see equation 5.2. If the unit's maximum power output is zero its mode is set to "Shut Down" and its power reference to zero. The remainder of the wind power units' modes are set to "Shut Down" and their desired power are set to zero.

$$P_{ref,unit} = \frac{P_{max,unit}}{P_{max,AllSelected}} \cdot P_{ref,plant} \tag{5.2}$$

When in island mode the control's task is to first power up the DC-grid, if necessary, and then decide how many wind power units that is needed. The control starts by setting the grid control's mode to "Island", which means that the grid controller should start to power up the grid if the DC-level is high enough. After this the controller enters the startup sequence. First a check is made to see whether there is a power output on the grid or not, if so is the case another check is made to see if the cool-down time is over. If the cool-down time is not over the counter is increased by the sampling period and the function *setIslandInit()* is called. If the cool-down time is over the function *splitIsland()* is called. If there is no power output on the grid the counter is set to zero and the function *setIslandInit()* is called.

When the function *setIslandInit()* is called the system is in the initiation phase which means that the system should be made ready to take on the "real" control orders. This means that the wind power units should start up and start to deliver a relatively small amount of power. This is done by telling all the units, whose maximum power output is 1000W or more, to enter "Power Tracking" mode with a power reference of 1000W. All the units whose maximum power output is above zero but below 1000W to enter "Power Tracking" mode with a desired power equal to their maximum power. The

units whose maximum power capability currently is zero is set to operate at "Shut Down" with a power reference of zero.

When the function *splitIsland()* is called the system has completed its island initiation mode and is ready to make decisions regarding how many units that should be active. This is done in very much the same way that the power load is split when in the "Desired Effect" mode. The algorithm starts by deciding how many unit's that should be active, this is done in the same way as is done in the *splitDesEff()*"-function, with the difference that the minimum power that is to be achieved is 60% more than the currently consumed power of the grid. The control then sets all the units which should be active to operate in the "DC-Level"-mode and sets the desired power to a fraction of the loss-compensated power consumed by the grid , see equation 5.3.

$$P_{ref,unit} = \frac{P_{max,unit}}{P_{max,AllSelected}} \cdot \frac{P_{consumed}}{1 - losses} \tag{5.3}$$

5.5 Simulations and Results

In order to test the operation of the complete wind power plant some different test cases was designed, see section 4.1.1. The test cases were divided into two sections operating on two different sets of the wind power plant. The first plant setup only contains one wind power unit and the second contains three wind power units.

5.5.1 One Wind Power Unit

In these test cases a plant setup containing one wind power unit and one grid connection, all controlled by the Plant Control. The design and testing of the individual parts can be found in chapter 3 and 4. The purpose of these tests is mainly to test the operation of the DC-control in both the grid connection, see section 4.4.1.1, and in the Power Controller, see section 3.4.2.

5.5.1.1 Case 1 - Max Power Varying Wind

This test was designed in order to test the DC-control of the grid controller. This is done by running the system in the "Maximum Power"-mode while varying the wind speed between 0-23m/s in the same way that is done in case 7 of chapter 3, see section 3.6.3.5.

The DC-level control results from the simulation are presented in figure 5.6, the results for the operation of the plant in a similar test case is presented and discussed in case 7 of chapter 3. The results are divided into seven different zones described below.

1. The wind speed starts at 0 m/s and increases to 5 m/s, during this period the rotor's rotational speed and the power reference are both zero. This since the wind speed is too low for operation of the plant.

2. In this zone the wind speed is increased from 5 m/s to 10 m/s which is enough to allow operation of the unit. The power reference is raised to its nominal value, 3000 W, and the rotor starts to rotate. Since the wind speed is not enough for nominal power the Power Controller tries to optimize the unit's power output.

3. In this zone the wind increases further and is now high enough to allow operation at nominal power.

4. In this zone the wind speed increases to 23 m/s, which is more than the maximum allowed for operation, hence the power production is halted.

5. The wind speed is now reduced to 18 m/s, which is lower than the maximum allowed wind speed for operation, and the power production is resumed.

6. The wind speed is now reduced to 11 m/s which is too low for nominal power and the Power Controller tries to optimize the unit's output power.

7. In this zone the wind speed is decreased to 3 m/s which is well below the minimum allowed wind speed, and the power production is halted.

The results from the simulation are presented in figure 5.6. The grid controller successfully controls the DC-level to its reference with some deviations when the input power from the wind power unit is changed. The DC-control is started when the DC-level is above 850V and turned off when the DC-level is below 600V. The main objective of the controller is to indirectly deliver the power generated by the wind power units to the grid connection. By controlling the DC-level to a constant value the power flow is maintained in balance so that the power generated by the wind power unit is delivered to the grid. Apart from decoupling the rotational speed of the wind turbine the DC-grid also provides a good filter for the power flow as seen in figure 5.6.

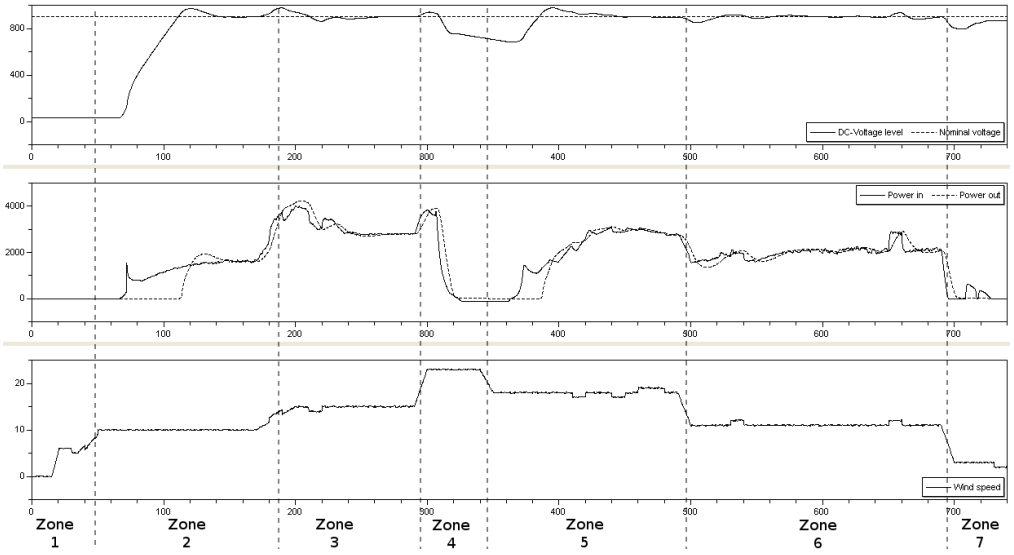


Figure 5.6: Simulation results for case 1, in the top subplot the DC-level and its reference are plotted, in the middle subplot the power in and out of the DC-grid are plotted and in the bottom subplot the wind speed is plotted.

5.5.1.2 Case 2 - Island Constant Load

This test was designed in order to test the DC-control implemented on the wind power unit when a constant load is connected. During this test the wind power was chosen to be between 14 and 18m/s in order for the power plant to be able to operate at its nominal power.

The results from the simulation are presented in figure 5.7 and 5.8. As can be seen in the results the power controller successfully controls the DC-level to 900V with some deviations. At the start the Plant Control orders the wind power unit to output 1000W in order to power up the DC-link capacitor. When the DC-level reaches 850V the grid controller starts powering up the grid. When the power flow out from the DC-grid has started the Plant Control waits until the cool down time has passed before it orders the wind power unit to control the DC-level. The Power Controller manages to relatively fast control the DC-voltage to 900V. After 250s the load is increased which causes the DC-voltage to start decrease, when the DC-voltage drops the power reference is changed in order to restore the DC-voltage. However it takes some time for the relatively slow turbine to change its power output which results in a quite large voltage drop. As depicted in figure 5.8 the grid controller does not have any trouble to power up the grid, and the drop in the DC-level is not so big that it has any effect on the inverter output.

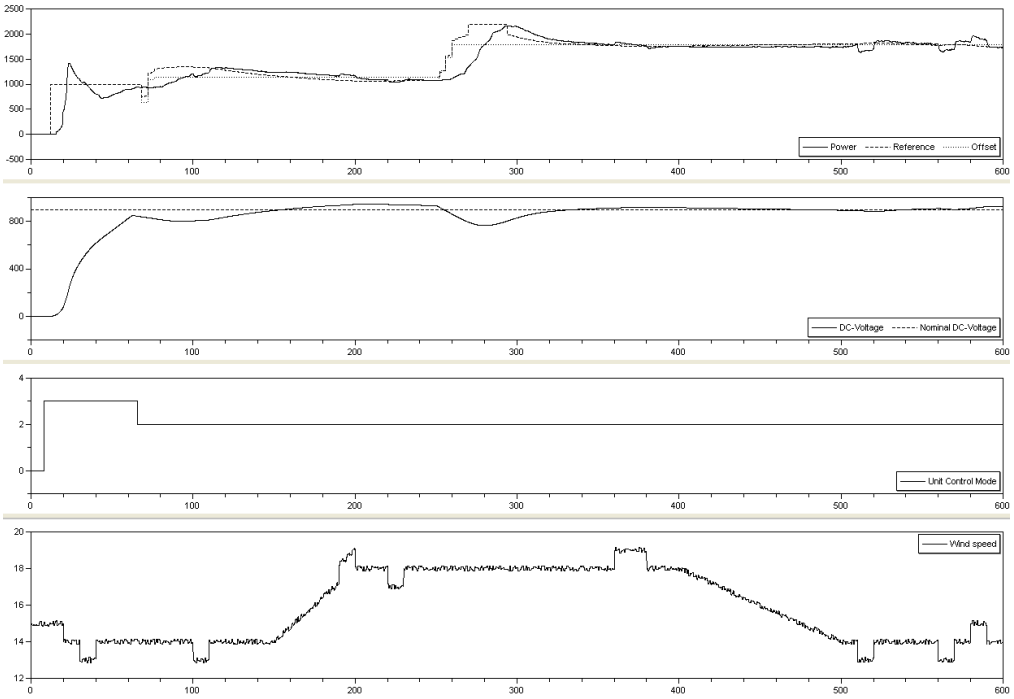


Figure 5.7: Simulation results for case 2, in the first subplot the desired power from the Plant Control, power reference and power output from the wind power unit are plotted, in the second subplot the DC-level and its reference are plotted, in the third subplot the command from the Plant Control is plotted and in the fourth subplot the wind speed is plotted.

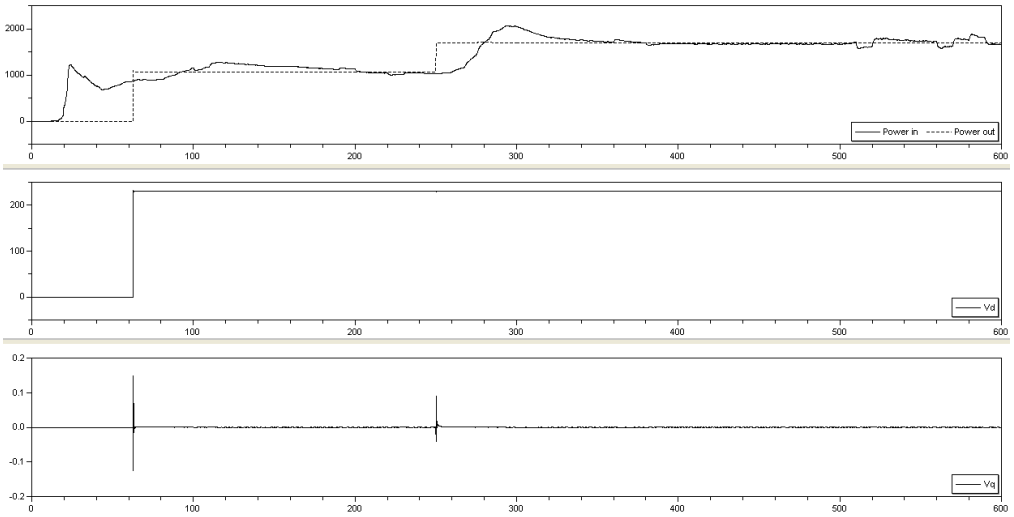


Figure 5.8: Simulation results for case 2, in the top subplot the power flow in and out from the DC-grid are plotted, in the middle subplot the direct voltage is plotted and in the bottom subplot the quadrature voltage is plotted.

5.5.1.3 Case 3 - Island Varying Load

This test was designed in order to test the DC-control implemented on the wind power unit when a varying load is connected. This case is identical to case 2 with the difference that the load is varying.

The results from the simulation are presented in figure 5.9 and 5.10. As in case 2 the power controller successfully controls the DC-voltage to 900V. The effect of the loads oscillation is visible in the DC-voltage which is experiencing a small oscillation. Except from the oscillation in the DC-level and output power the result is almost identical to that of case 2 and the DC-control has no apparent problems controlling the DC-level when a varying load is connected.

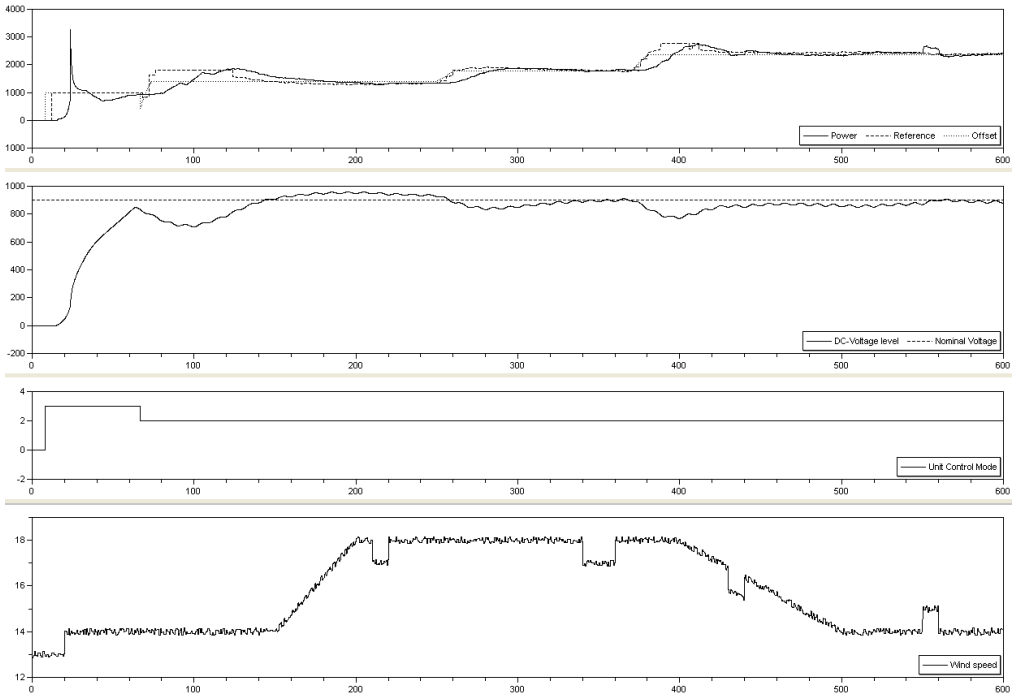


Figure 5.9: Simulation results for case 3, in the first subplot the desired power from the Plant Control, power reference and power output from the wind power unit are plotted, in the second subplot the DC-level and its reference are plotted, in the third subplot the command from the Plant Control is plotted and in the fourth subplot the wind speed is plotted.

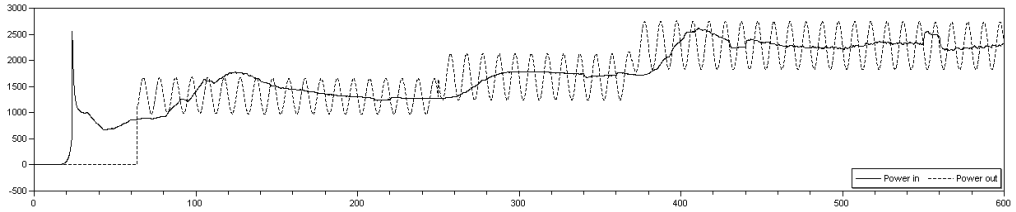


Figure 5.10: Simulation result for case 3, in this plot the power delivered to the connection point and the power output from the wind power unit are plotted.

5.5.2 Three Wind Power Units

In these test cases a plant setup containing three wind power units and one grid connection, all controlled by the Plant Control, was used. The design and testing of the individual parts can be found in chapter 3 and 4. The purpose of these tests is mainly to test how the Plant Control manages the several wind power units connected to the plant.

5.5.2.1 Case 1 - Delta-Control Varying Wind

This case was designed in order to test the delta control when implemented on the complete system. In the test all the wind power units starts in the "Maximum Power"-mode and after 150s, as marked in the figures, the mode is changed to "Delta Control" and after about 400s the wind speed is decreased. The change in wind speed is different for the three wind power units.

The results are presented in figure 5.11 and 5.12. According to the grid codes the delta control mode should be initiated no later than two seconds after it is ordered, see section 4.2.3. There are two ways to achieve this goal. The easiest way is to allow the highest sampling time to be a maximum of two seconds and all other sampling times to be a multiple of this. Another possible, and more reliable, way is to connect the command change as an interrupt which forces an immediate sample instant. In this project the first option of these two was implemented. The grid codes also state that the control command should be completed within 30 seconds and should not deviate more than $\pm 2\%$ from the setpoint or $\pm 0.5\%$ of the rated power, whichever yields the highest tolerance, see section 4.2.3. This constraint is barely achieved, the delta level is reached after $\sim$28s, which is below the allowed limit, however when the wind power units is exposed to a wind gust the power output is increased/decreased more than $\pm 2\%$. The delta control, however, is intended for plants with a rated output power greater than 25MW, and since our plants are very small, 3kW, their rotors are relatively small. Which in turn means that they have a lower inertia and are therefore affected more by the gusts. When the wind speed change occurs the power output is decreased faster than the reference, this since the maximum power output is filtered using an averaging filter which causes it to have a delay that is very visible during fast changes in the wind speed.

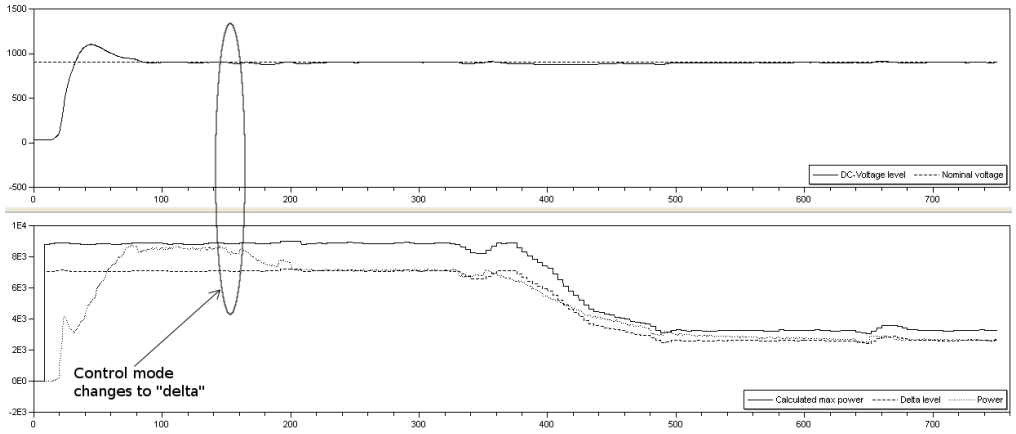


Figure 5.11: Simulation results for case 1, in the top subplot the DC-level and its reference are plotted and in the bottom subplot the total maximum power is plotted together with the calculated delta power level and the measured power into the DC-grid.

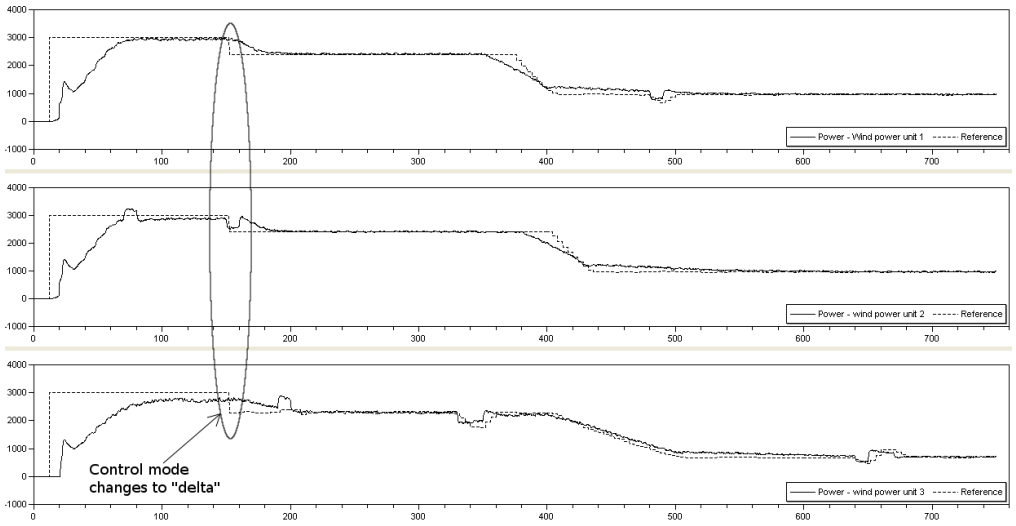


Figure 5.12: Simulation results for case 1, the three subplots are showing the power output and power reference from each of the three wind power units.

5.5.2.2 Case 2 - Deliver Specific Amount of Active Power

This case was designed in order to test how well the plant delivers a specific power to the grid connection. This is done by using both the individual wind power units and the grid control. The wind speed is here kept constant for all of the plants, but at different levels in order to show how the Plant Control divides the load amongst the wind power units. All of the units are started in the "Maximum Power" mode, after 140s the mode is changed to "Desired Power" and after 300s the desired power is increased from 3kW to 5.5kW.

The results from the simulation are shown in figure 5.13 and 5.14. When the mode is changed to "Desired Power" the Plant Control orders one of the wind power units to shut down since only two of them are needed for operation.

The other two units decreases their power output to the setpoint received from the Plant Control, which is the same for the two units since both of them is able to output their nominal power, 3kW. On the grid side when the mode is changed the output power from the grid controller is limited by the reference which causes the power output to rapidly decrease to the desired level. Since the power output from the DC-grid is decreased much faster than the power input the DC-level is increased quite heavily during the transition. When the power input level from the wind power units have achieved their reference the DC-grid voltage is kept constant.

When the desired power is increased to 5.5kW the power output to the grid is increased rapidly to its reference, however not as quick as the previous decrease. This drains some of the energy stored in the DC-link. The increase in the desired power output causes the Plant Control to turn on one additional wind power unit. The additional unit's maximum output power is below its nominal power, and also below the maximum of the other two wind power units, this causes the desired output from the additional wind power unit to be slightly less than the desired output from the other two units. When all of the wind power units have found their desired reference value the DC-level is kept constant.

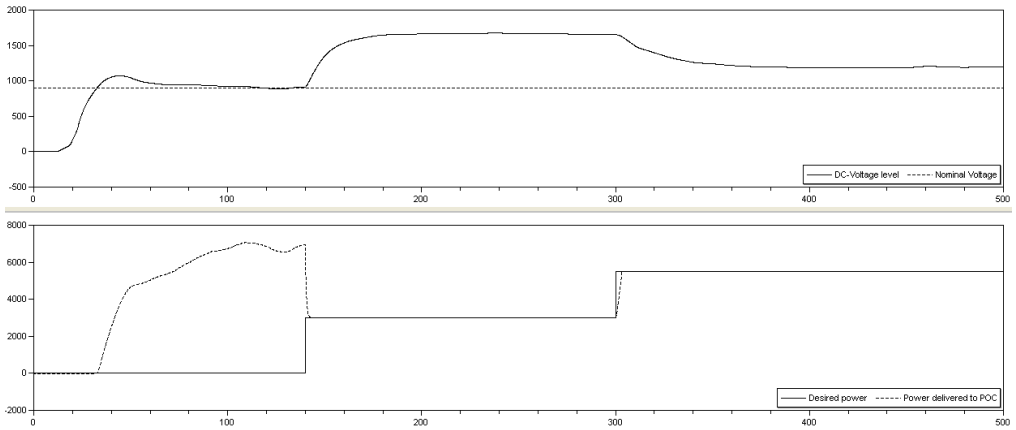


Figure 5.13: Simulation results for case 2, in the top subplot the DC-level and its reference are plotted and in the bottom subplot the power output on the grid is plotted together with its reference.

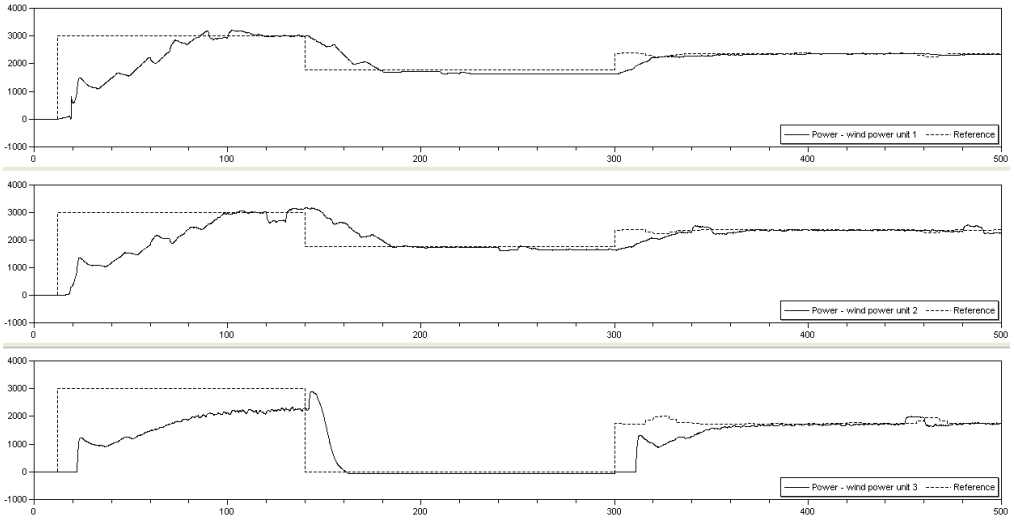


Figure 5.14: Simulation results for case 2, the three subplots are showing the power output and power reference from each of the three wind power units.

5.5.2.3 Case 3 - With/Without Reactive Compensation

This case was designed in order to test the operation through all of the possible changes in wind speed zones possible[2]. The test is similar to test case 7 in chapter 3, see section 3.6.3.5. In the test the different wind power units will be exposed to similar wind profiles to simulate the different conditions in different physical locations.

The simulation has been divided into 7 different zones, depicted in figure 5.15 to 5.18, in zone:

1. The wind speed starts at ∼0 m/s and increases to ∼5 m/s, during this period the wind power units have too low winds to allow operation.

2. In this zone the wind speed is increased from ∼5 m/s to ∼10 m/s which is enough to allow operation of the wind power units.

3. In this zone the wind increases further and is now high enough to allow operation at nominal power.

4. In this zone the wind speed of wind power unit two and three is increased to ∼23 m/s, which is more than the maximum allowed for operation, hence the wind power units shuts down. The wind speed for wind power unit one is increased to ∼19 m/s, which still allows operation of unit one.

5. The wind speed is now reduced to ∼18 m/s, which is lower than the maximum allowed wind speed for operation, and the wind power units resumes operation. In this zone the reactive power compensation has also been turned on.

[2]Too low for operation, not high enough for nominal power, high enough for nominal power and to high for operation.

6. The wind speed is now reduced to ∼11 m/s which is too low for nominal power and the wind power units tries to maximize the power output.

The results from the simulation are shown in figure 5.15 to 5.18. The operation of the wind power units during the simulation is similar to test case 7 in chapter 3, see section 3.6.3.5, for the first 650s. After 650s the Plant Control orders a complete shut down of the whole plant which causes all of the wind power units to shut down.

In figure 5.17 and 5.18 the results from the grid side of the plant is presented. As expected the direct voltage is high above the nominal voltage of 230V during the time the reactive power compensation is inactive, and the reactive power output during the same time period is zero, according to Swedish grid codes. When the reactive power compensation is turned on, after 350s, the direct voltage is reduced from ∼240V to its nominal value 230V in less than 10 seconds, which is well below the requirements of the grid codes, see section 4.2.4. The increase in reactive power consumption results in a small momentary increase of the active power output, which causes the DC-voltage to decrease.

When the Plant Control orders a shutdown of the plant the Grid side controller is ordered to the "Inactive"-mode which causes it to immediately disconnect from the grid. An effect of this is that the DC-level is increased when the generators of the wind power units are stopping the rotors, and no energy can be dissipated by the grid controller. Also the active and reactive power output drops to zero.

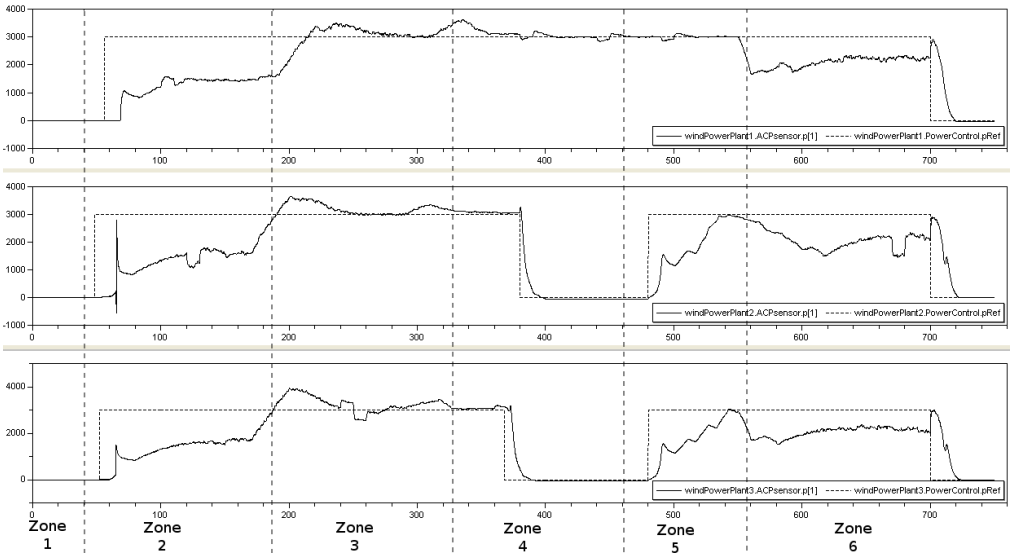


Figure 5.15: Simulation results for case 3, the three subplots are showing the power output and power reference from each of the three wind power units.

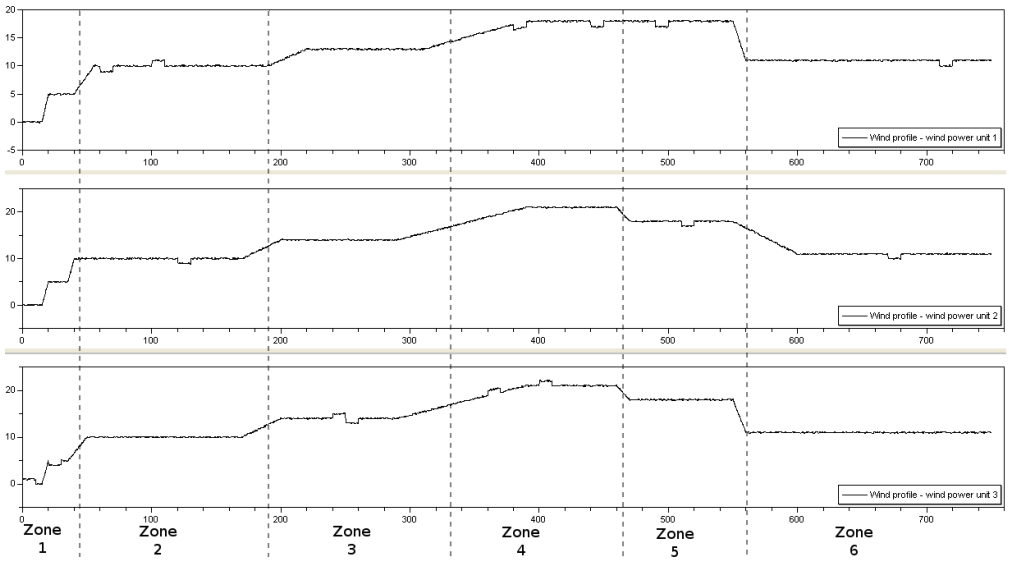


Figure 5.16: Simulation results for case 3, the three subplots are showing the wind speed profiles for the three wind power units.

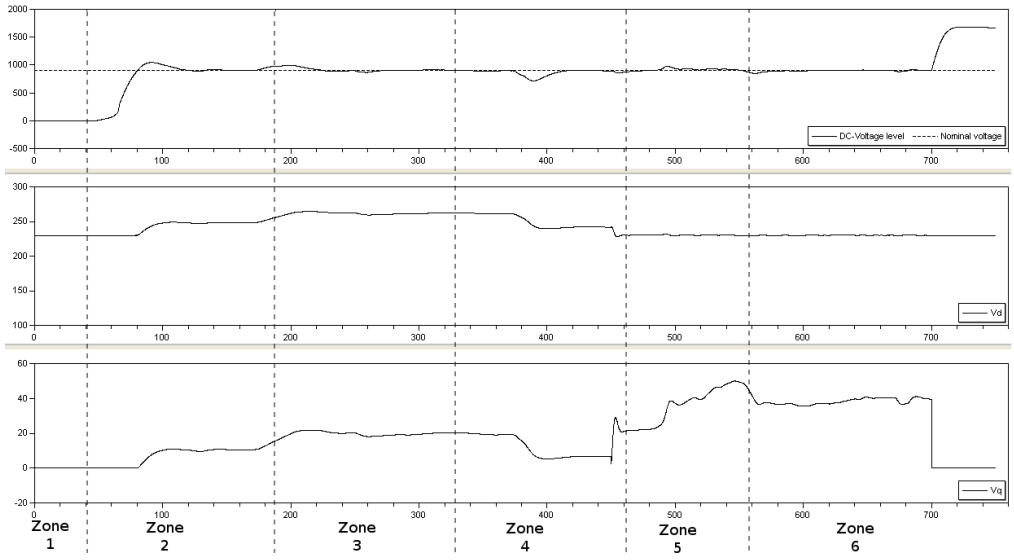


Figure 5.17: Simulation results for case 3, in the top subplot the DC-voltage level is plotted together with its reference, in the middle subplot the direct voltage is plotted and in the bottom subplot the quadrature voltage is plotted.

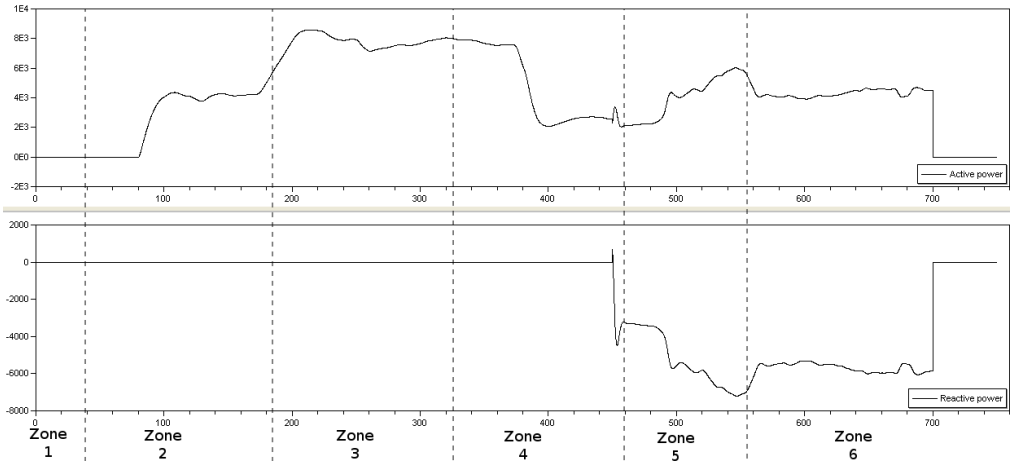


Figure 5.18: Simulation results for case 3, in the top subplot the active power output at the point of connection is plotted and in the bottom subplot the reactive power output at the point of connection is plotted.

5.5.2.4 Case 4 - Island Varying Wind and Load

This case was designed in order to test the island operation when the whole wind power plant is involved.

The simulation has been divided into 6 different zones, depicted in figure 5.19 to 5.21, in zone:

1. The wind speed starts at ∼0 m/s and increases to ∼5 m/s, during this period the wind power units have too low winds to allow operation.

2. In this zone the wind speed is increased from ∼5 m/s to ∼10 m/s which is enough to allow operation of the different wind power units.

3. In this zone the wind increases further and is now high enough to allow operation at nominal power.

4. In this zone the wind speed of wind power unit two and three is increased to ∼23 m/s, which is more than the maximum allowed for operation, hence the wind power units shuts down. The wind speed for wind power unit one is increased to ∼19 m/s, which still allows operation of unit one.

5. The wind speed is now reduced to ∼18 m/s, which is lower than the maximum allowed wind speed for operation, and the wind power units resumes operation.

6. The wind speed is now reduced to ∼11 m/s which is too low for nominal power and the wind power units tries to maximize the power output.

The results from the simulation are shown in figure 5.19 to 5.21. The control successfully powers the grid during the simulation. When the system is started the wind power units are turned on in the "Desired Power" mode when their individual winds are sufficient for operation. When the DC-grid has been charged the Plant Control decides to shut down one unit and change

the mode of the other two to DC-control. This since the two wind power units with their current maximum power output should suffice for the power consumed.

When the wind increases in the beginning of zone 3 the Plant Control decides to shut down one of the wind power units since the one left should be able to maintain the grid on its own. The increase of the consumed power in the middle of zone 3 causes the Plant Control once again to turn on wind power unit two. In zone 4 the wind speed for unit two and three increases to a level which is higher than what is allowed for operation. This forces the Plant Control to try to only operate with one wind power unit. In this particular case it turns out that it was possible to maintain the DC-level with just one wind power unit. In zone 5 the wind speed for unit two and three is decreased to an acceptable level and the Plant Control resumes operation for wind power unit two in order to increase its control margins. The small power increase at ∼380s is not big enough to effect the number of plants that should be operating. The wind speed decrease that occurs in the beginning of zone 6 is not big enough to change the decision about the current number of active wind power units. The large increase of the load which occurs at ∼650s causes the Plant Control to increase the number of active plants to all three. The increase of the load also causes a sudden drop in the DC-level which the DC-control of the wind power units manages to halt.

The power reference shown in figure 5.19 reflects the power reference sent to the power control part of the Power Controller, see section 3.4.2. The control action from the Power Controller is most visible when the DC-level error is very high. The reference is then changed to either zero or nominal power. The oscillation of the direct and quadrature voltage in the beginning of the simulation is a reaction to how the islanding grid model was implemented.

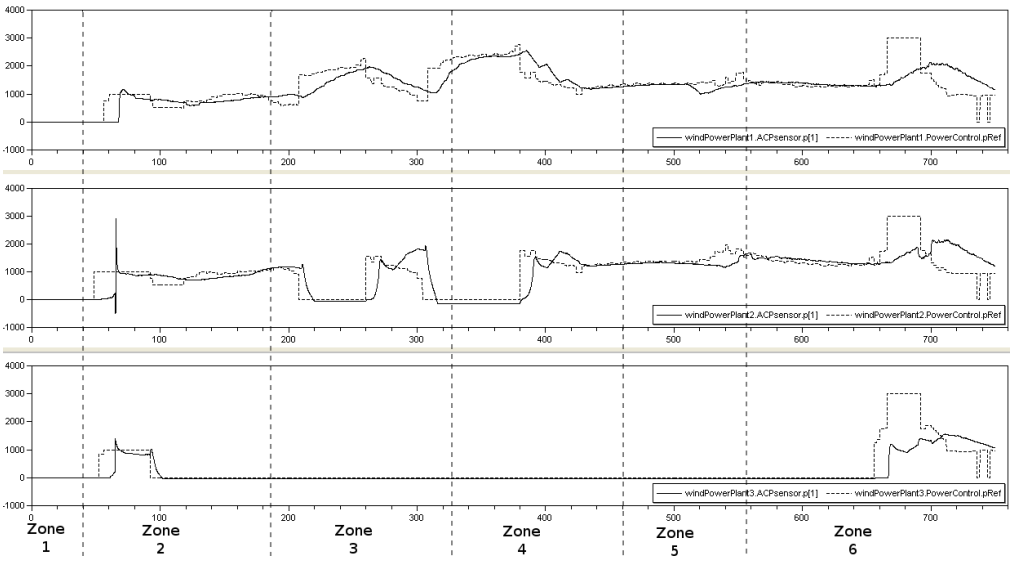


Figure 5.19: Simulation results for case 4, the three subplots are here showing the power output and power reference from each of the three wind power units.

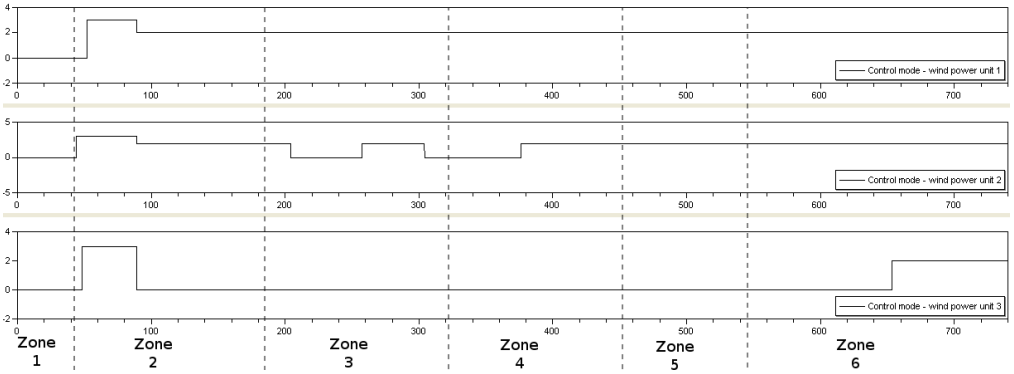


Figure 5.20: Simulation results for case 4, the three subplots are here showing the commanded modes from the Plant Control to the three wind power units.

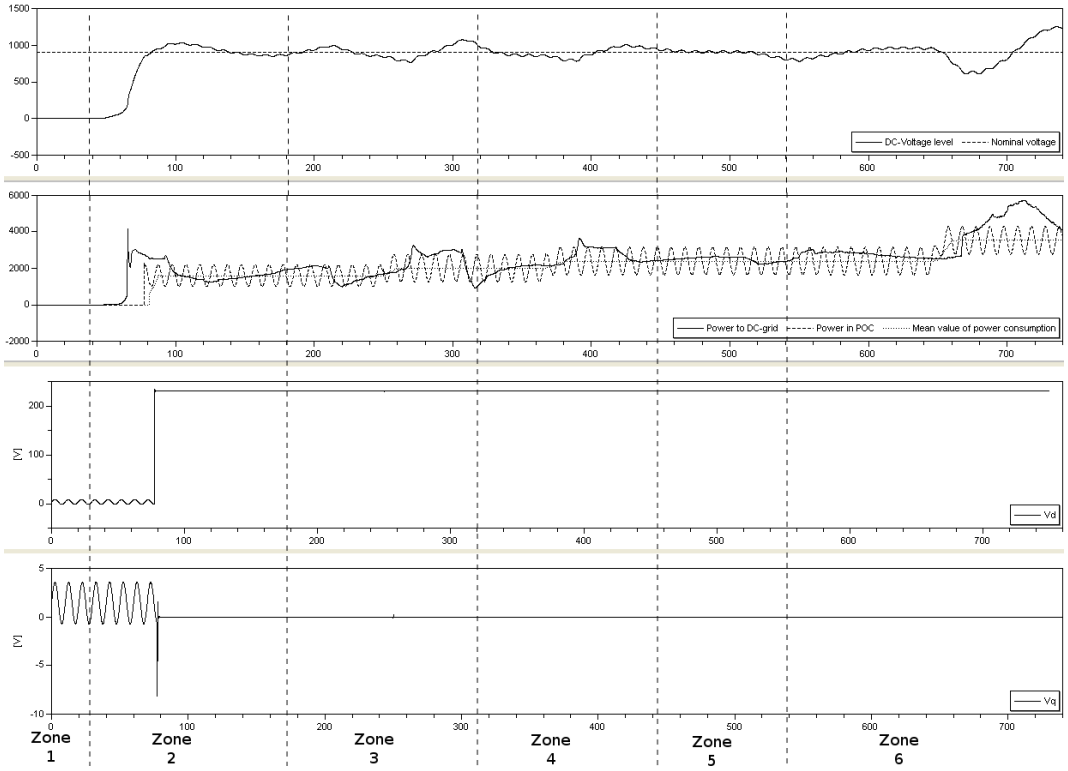


Figure 5.21: Simulation results for case 4, in the first subplot the DC-voltage level is plotted together with its reference, in the second subplot the power flow into the DC-grid, out at the grid connection and the power consumption calculated by the grid side controller are plotted, in the third subplot the direct voltage is plotted and in the fourth subplot the quadrature voltage is plotted.

5.5.3 Losses

The overall losses of the wind power plant are depicted in figure 5.22. The losses in the plant can be divided into four different components. The losses in

the generator and shaft, the losses in the inverter controlling the wind power unit, the losses in the grid side inverter and losses in the grid which transfers the power to the point of connection. All of these losses is approximately a fixed percentage during the different simulations, however the grid losses is varying depending on the reactive power flow. A higher flow of reactive power increases the losses in the grid, as can be seen in figure 4.28. In the system simulated in this project the power loss from the shaft and generator is ∼15%, the losses from the inverter controlling the generator is ∼5%, the losses from the grid side inverter is ∼5% and the losses in the transfer grid is ∼8%. Overall the total loss in this case is ∼28%. The Plant Control should compensate for all losses except the losses in the generator and shaft when requesting a power output from the wind power units. In this case the Plant Control should compensate for ∼17% losses.

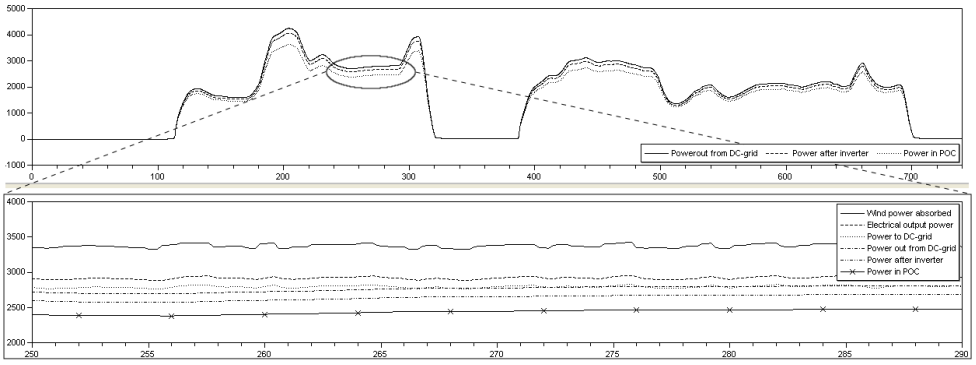


Figure 5.22: In this plot the different power levels of the plant are plotted in order to visualize the losses. In the top subplot the power flow out from the DC-grid, out from the inverter and in the point of connection are plotted and in the bottom subplot the same power flows as in the top subplot is zoomed in together with the power flow into the DC-grid, the power generated by the generator and the power absorbed by the rotor.

5.6 Discussion

The topic of this chapter has been the complete control of the wind power plant, mainly focusing on the managing control by the Plant Control. Due to a lack of time there is room for improvements.

One major improvement is to add functionality in the Plant Control so that when a desired output power is requested the Plant Control makes sure that the requested power is delivered. This is needed in case the loss estimation is wrong due to changes in component, reactive power flow etc. One suggestion of how to do this is to add a measurement of the current power output from the wind power units to the Plant Control. The Plant Control can then use these measurements to see if the wind power units have reached their reference output and if so are the case adjusts their output so that the power output reference on the grid is achieved.

Another improvement that can be made is to add more intelligence in the Islanding functionality. For example all the wind power units does not need to be in either DC-control mode or shut down, some could be running at desired or maximum power. The Plant Control could also be made more intelligent when choosing which units to turn on and off so that one is not turned off in order to turn on another unit. In the long time scope the Plant Control should also try to utilize the different wind power units an equal amount.

There are a lot of work that can be done regarding the Plant Control in general, there are no real limits for how "intelligent" that it can be made.

The total control also lacks a gradient control that is controlling the amount of power that is connected/disconnected, which would be needed in order to achieve some of the grid codes.

Overall we are satisfied with how the Plant Control performs. The Plant Control is capable of supplying a small part of the grid in Island mode by choosing which wind power units that should be operating. It also succeeds in deliver a requested amount of active power to the grid, and while doing so splits the load proportionally over the wind power units that are chosen to be active. There is however some modes that needs to be implemented in order for the plant to pass all the grid codes. However all the grid codes, except delta control, considered in the simulations was achieved with a comfortable margin.

Chapter 6

Conclusion

In this final chapter the results of the project is briefly summarized, some key aspects are brought forward and some future work and improvements are suggested.

6.1 Evaluation of the Project

The purpose of this project was to model a wind power unit in Dymola/Modelica complete with power electronics and to design a control system in C/C++ which should be used to control the model in Dymola. Apart from modeling the single wind power unit a model of a small wind power plant should also be done, together with a simple managing control, complete with a grid model, the project limitations are listed in section 1.2.1.

Overall the project was a success. A simple turbine model was developed and interconnected with power electronics to a DC-grid and control for this was developed. A control system, complete with optimization algorithms, was designed in C/C++, as requested, and interconnected with Dymola in a way which allows the same code, regarding the control algorithms, to be used on a "real" wind power plant. A simple grid-connection model was developed and a control system for power output on the grid was designed. Some simple control strategies for a wind power plant using one or several wind power units were designed. The results from the simulations were compared to actual grid codes. It was chosen to use the grid codes of Denmark instead of Sweden to evaluate the control performance since the Swedish grid codes have relatively low demands regarding wind power plants when compared to the grid codes of Denmark, this since wind power is a relatively large source of power in the Danish grid.

The implementation of the control system was done using an external static library turned out to be very convenient. The control code could be written completely separate from the models. In order to interconnect the control system with the models some interface classes and declarations of the control functions needed to be done. The only drawback of this is that it proved to be a bit inconvenient to change the input to or output from the functions. For example if an input should be added to a function the code needs to be changed in six places, the header and C-code of the class, the header and C-code of the interface class, the function definition in Dymola and the model implementing the function in Dymola.

The wind turbine model was implemented using a C_p-curve estimation with a compensation for the tower shadowing effect. The turbine model was then propelled by a wind model distorting the wind with gusts and turbulence, which are common components in real wind profiles.

In order to model the inverters two different models were used, one switched and one averaging. Because of its fast simulation speed the average model was used for most of the simulations. The switched inverter was tested in the simulation of the system, however only one of the simulations is presented in this project. In the results from the simulation using the switched inverter, see part three of 4.6.1.3, it can be observed that the output from the average model is accurate.

When the plant is running in island mode the DC-level is controlled by the wind power units. This control was first implemented using a PID-controller, however the results using this approach was not satisfactory. The wind power

units managed to power up the grid and keep the DC-level constant but were more sensitive to changes of the load than the final controller. This and the fact that all measurements that is needed for calculating the power consumed by the load is also needed for controlling the grid side inverter was the reason to why the DC-control algorithm of the units were changed. The power measurement is also needed for the Plant Control to decide how many units that should be operating.

During the implementation of the controllers PID-controllers has been utilized in some parts, for instance in the Speed Controller and GridVoltageController. However during the final tuning of the controls it was decided, through testing, to set the derivative gain to zero. This result in that the controls used is in essential PI-controllers.

The overall control performance of the controllers designed during the project has to be considered to be good. The control fulfills most of the grid codes which was examined, and it feels like it should be possible to fulfill all of them if there would have been more time available. This also considering that the wind profiles which were used for the test cases might have been a bit extreme. For example during the total case the wind is changing from none at all to strong gale, almost storm[1] and back again in about 600 seconds, or 10 minutes. The reason to why the wind profiles were chosen like this was in order to test and stress the control algorithms to their maximum. One grid code in particular was never discussed in the result sections. The regulations about when it is allowed for the control to automatically disconnect from the grid. In our case the control system only disconnects from the grid when the DC-level is too low, no specific "intelligence" for when to disconnect from the grid was developed.

6.1.1 Improvements

There are some distinct improvements that could be made to further improve the results of the modeling and control done in this project.

Model Verification

Unfortunately no real measurement data was available for model verification, which meant that the models could not be verified. The main effect of this is that the loss models and the absolute power and rotational values might not be accurate. Especially the loss estimation might deviate a great portion from its actual values. The control and optimization algorithms should not be affected a great deal by this. For the optimization the most important aspect is the C_p-curve, however the most important aspect of the C_p-curve is its general shape. Which in this project was obtained by using general parameters found in [11] and [12]. Especially the grid model should be reexamined in order to make it more accurate. At most the parameters of the control algorithms might need a re-tuning to work on a more accurate system.

[1] According to the Beaufort Scale, in which strong gale is force nine out of twelve.[21]

Islanding

The DC-control of the wind power units is currently working and manages to power up most loads. There are however some distinct improvements that could be made in the Plant Control in order to improve the control. The main reason to why the control fails is due to the fact that it makes the mistake of quickly turning on and off a unit. This could be avoided by for examples implementing some kind of hysteresis in the decision making algorithm. The Plant Control is currently always choosing to use the wind power units in the order they are stored in the array. This means that if unit 1 cannot operate the Plant Control will choose to activate unit 2, however when unit 1 becomes available again the Plant Control will choose to turn of unit 2 in order to turn on unit 1, provided that only one wind power unit is needed. This is unnecessary and might cause the system to fail. The fact that the wind power units are prioritized in a specific order also means that unit 1 will be used much more frequently than unit 3.

Reactive Power Control

In the project an algorithm for controlling the reactive power was implemented. In the normal case the reactive power is controlled to zero according to the Swedish grid codes [20]. During reactive power compensation the reactive power is controlling the direct voltage to its reference. There are two additional ways of controlling the reactive power output that was not considered. The first way is called Q-control, see section 4.2.1, which in essential means that the reactive power should be controlled by a reference value received from the grid owner. Another way of controlling the reactive power output is called power factor control, see section 4.2.1. During power factor control the reactive power should be controlled so that the reactive power output is proportional to the active power output. What differs the different control strategies are how the reactive power reference is set. This means that the implementation of these two additional control modes not would be difficult. As can be seen from the results of chapter 4 in section 4.6 the reactive power control is capable of tracking a reference value.

6.2 Real-time

In the project limitations it was stated that the possibility of running the models in real-time should be considered, if time permits. Some models for changing the wind speed in real-time were designed and implemented. Some initial tests of this were done while other parts of the project were stagnant. The results from these tests were somewhat positive. The current status is that the models cannot be run at real-time without delay. Currently the simulation runs at a third of the speed of the process. However in these tests no optimization what so ever for real-time simulations has been done, and it is believed that with some optimization it would be possible to run the

models in real-time as well. One major improvement would be to replace fast-sampling controllers with equivalent controls in continues-time. For example the PI/PID-controllers in Modelica's standard library. This would allow the solver used by Dymola to take larger step-sizes during its integration.

6.3 Future Work

There is a lot of work left that can be done in and around this project. Some suggestions are presented below.

Modeling

There is still a lot of work that can be done regarding modeling of the plant, especially when it comes to the grid models. A part of improving the models is to do model verification in order to be able to more accurately simulate the operation of a specific plant.

Switched Modeling and Control

In order to make the models more accurate in the short time scope the models and control should be more adapted to work on the switched model. More precisely electrical low pass filters needs to be introduced after the inverters in order to remove the main harmonics caused by the switching. Apart from this there are still a bit of work to be done regarding synchronizing with the grid, which is estimating the current system angle on the grid.

Real-Time

The work on running the models in real-time was begun and the possibility of the models to work in real-time is assessed to be high. However for this to actually work some of the models needs to be optimized for real-time. More importantly the fast sampling controls, sampling periods $<$1s, could be implemented in Dymola/Modelica as continues time models, this since these controls are fast and is basically simple PI-/PID-controllers.

Plant Management

There is a lot of work which is left to be done regarding the Plant Control, there are almost endless possibilities of how much intelligence and logics that can be implemented into the Plant Control. Some examples are listed below.

- Enable the Plant Control to automatically enter island mode.
- Enable the Plant Control to automatically disconnect from the grid in order to avoid unintended island operation.
- Enable the Plant Control to automatically compensate for errors in the loss estimation.

- Modify so that the Start and Stop of the wind power units, as well as disconnection form the grid, is done according to the grid codes.
- Improve the DC-control, all units does not need to be in either "Shut Down" or "DC-Control" some could be in for example "Max Power" or "Power Track". The Plant Control should also be made more aware of which wind power units that has been operating so that the control does not turn off one unit in order to turn on a different wind power unit. In the long time scope the Plant Control should also try to utilize the different wind power units an equal amount.
- More closely study the grid codes to see if any additional control modes are needed, or if any of the existing modes needs to be modified.

Scaling

Currently most of the system parts, such as turbine, generator, DC-grid, inverters and AC-grid, are scalable. However the only control algorithm that is automatically scaled is the current control of the generator. In order for the system to be fully scalable all of the controls should be parameterized, for instance the Speed Control is very dependent of the rotor's inertia.

Bibliography

[1] T. Dang and M. H. Rashid, "Introduction, history, and theory of wind power," North American Power Symposium (NAPS), 2009, pp. 1–6, 2009.

[2] T. Wizelius, Vindkraft i teori och praktik. Lund Studentlitteratur 2002, 2002.

[3] E. Sverige, "Vindstatistik 2010." Available at `http://www.energimyndigheten.se/Global/Press/Pressmeddelanden/Vindkraftsstatistik-2010-ny.pdf` 2011-09-22.

[4] D. Flan and R. Blom, "Great creations," Power Engineer (see also Power Engineering Journal) 2005, pp. 14–17, 2005.

[5] Falkenberg-energi, "Vertikalaxlad-vindkraft." Available at `http://www.falkenberg-energi.se/vindkraft/vertikalaxlad-vindkraft` 2011-09-22.

[6] D. photo archive, "Darrieus wind turbine." Available at `http://www.doedigitalarchive.doe.gov/ImageDetailView.cfm?ImageID=1003145&page=search&pageid=thumb` 2011-09-22.

[7] W. Hu, Y. Wang, X. Song, and Z. Wang, "Development of vertical-axis wind turbine with asynchronus generator interconnected to the electric network," Electrical Machines and Systems, 2008. ICEMS 2008. International Conference on, pp. 2289–2293, 2008.

[8] H. stuff work, "How a wind turbine works." Available at `http://science.howstuffworks.com/environmental/green-science/wind-power1.htm` 2011-09-22.

[9] J. F. Manwell, J. G. McGowan, and A. L. Rogers, Wind Energy Explained - Theory, Design and Application second edition. DJohn Wiley & Sons Ltd, 2009.

[10] Åke Larsson, "Vindkraft i lokala och regionala nät - elektriska egenskaper och elkvalitet," Elforsk, rapport 98:20, 1998.

[11] I. Catana, C.-A. Safta, and V. Panduru, "Power optimisation control system od wind turbines by changing the pitch angle," U.P.B. Sci. Bull., Series D, Vol. 72, Iss. 1, pp. 142–146, 2010.

[12] A. Pintea, D.Popescu, and I. Pisica, "Robust model based control method for wind energy production," MCPL'2010: 5th Conference on Management and Control of Production, Coimbra : Portugal (2010), 2010.

[13] A. Rolán, Álvaro Luna, G. Vásquez, D. Aguilar, and G. Azevedo, "Modeling of a variable speed wind turbine with a permanent magnet synchronous generator," IEEE International Symposium on Industrial Electronics (ISIE 2009), pp. 734–739, 2009.

[14] N. Urasaki, T. Senjyu, and K. Uezato, "Influence of all losses on permanent magnet synchronus motor drives," Industrial Electronics Society, 2000. IECON 2000. 26th Annual Conference of the IEEE, pp. 1371–1376 vol.2, 2000.

[15] M. Alaküla and P. Karlsson, Power Electronics, Devices, Converters, Control and Applications. Department of Industrial Electrical Enginering and Automation, Lund Institute of Technology, 2010.

[16] M. Otter, "Modelica overview." Available at `https://www.modelica.org/education/educational-material/lecture-material/english/ModelicaOverview.pdf` 2011-08-22.

[17] J. Thongham, P. Bouchard, H. Ezzaidi, and M. Ouhrouche, "Wind speed sensorless maximum power point tracking control of variable spweed wind energy conversion systems," Electric Machines and Drives Conference, 2009. IEMDC '09. IEEE International, pp. 1832–1837, 2009.

[18] Modelica-Association, "Modelica®- a unified object-oriented language for physical systems modeling." Available at `https://www.modelica.org/documents/ModelicaSpec32.pdf` 2011-08-23.

[19] Energinet.dk, "Technical regulation 3.2.5 for wind power plants with a power output greater than 11 kw." Available at `http://www.energinet.dk/SiteCollectionDocuments/Engelske20dokumenter/El/Grid20Code203.2.420Power20Unit20above201120kW20and20up20to201,520MW.pdf` 2011-10-24.

[20] Svenska-Kraftnät, "Technical regulation 3.2.5 for wind power plants with a power output greater than 11 kw." Available at `http://www.svk.se/Global/07_Tekniska_krav/Pdf/Foreskrifter/SvKFS2005_2.pdf` 2011-10-24.

[21] Nationalencyklopedin, "Beaufortskalan." Available at `http://www.ne.se.ludwig.lub.lu.se/enkel/beaufortskalan` 2011-11-08.

Appendix A

Equations

A.1 Derivation abc⇒ $\alpha\beta$ ⇒dq

Derivation of a perfectly symmetrical three phase voltage from the abc-reference frame to the dq-reference frame.

$$u_a = \hat{U}\cos(\omega_{el}t)$$

$$u_b = \hat{U}\cos(\omega_{el}t - \frac{2\pi}{3})$$

$$u_c = \hat{U}\cos(\omega_{el}t - \frac{4\pi}{3}) = \hat{U}\cos(\omega_{el}t + \frac{2\pi}{3})$$

$$U_{\alpha\beta} = \frac{\sqrt{2}}{3}(u_a + e^{j\frac{2\pi}{3}}u_b + e^{-j\frac{2\pi}{3}}u_c) =$$

$$= \frac{\sqrt{2}}{3}\hat{U}\left(\cos(\omega_{el}t) + e^{j\frac{2\pi}{3}}\cos(\omega_{el}t - \frac{2\pi}{3}) + e^{-j\frac{2\pi}{3}}\cos(\omega_{el}t + \frac{2\pi}{3})\right) =$$

$$= \left[\cos(\omega_{el}t) = \frac{e^{j\cdot\omega_{el}t} + e^{-j\cdot\omega_{el}t}}{2}\right] =$$

$$= \frac{1}{3\sqrt{2}}\hat{U}\left(e^{j\cdot\omega_{el}t} + e^{-j\cdot\omega_{el}t} + e^{j\frac{2\pi}{3}}(e^{j\cdot\omega_{el}t}e^{-j\frac{2\pi}{3}} + e^{-j\cdot\omega_{el}t}e^{j\frac{2\pi}{3}}) + \right.$$
$$\left. + e^{-j\frac{2\pi}{3}}(e^{j\cdot\omega_{el}t}e^{j\frac{2\pi}{3}} + e^{-j\cdot\omega_{el}t}e^{-j\frac{2\pi}{3}})\right) =$$

$$= \frac{\hat{U}}{3\sqrt{2}} \left(e^{j\cdot\omega_{el}t} + e^{-j\cdot\omega_{el}t} + e^{j\cdot\omega_{el}t} + e^{-j\cdot\omega_{el}t} e^{-j\frac{2\pi}{3}} + e^{j\cdot\omega_{el}t} + e^{-j\cdot\omega_{el}t} e^{j\frac{2\pi}{3}} \right) =$$

$$= \frac{\hat{U}}{3\sqrt{2}} \left(e^{j\cdot\omega_{el}t} (1+1+1) + e^{-j\cdot\omega_{el}t} \left(1 + e^{-j\frac{2\pi}{3}} + e^{j\frac{2\pi}{3}} \right) \right) =$$

$$= \left[\begin{array}{l} e^{j\frac{2\pi}{3}} = -\frac{1}{2} + j\frac{\sqrt{3}}{2} \\ e^{-j\frac{2\pi}{3}} = -\frac{1}{2} - j\frac{\sqrt{3}}{2} \end{array} \right] =$$

$$= \frac{\hat{U}}{3\sqrt{2}} \left(3e^{j\cdot\omega_{el}t} + e^{-j\cdot\omega_{el}t} \left(1 - \frac{1}{2} - j\frac{\sqrt{3}}{2} - \frac{1}{2} + j\frac{\sqrt{3}}{2} \right) \right) =$$

$$= \frac{\hat{U}}{3\sqrt{2}} \cdot 3e^{j\cdot\omega_{el}t} = \frac{1}{\sqrt{2}} \hat{U} e^{j\cdot\omega_{el}t}$$

$$U_{dq} = \underline{\underline{V_{\alpha\beta} e^{-j\cdot\omega_{el}t} = \frac{1}{\sqrt{2}} \hat{U}}}$$

Appendix B

Source code

B.1 Turbine Control

Listing B.1: TurbineControl.h

```
1  #ifndef TURBINECONTROL_H
2  #define TURBINECONTROL_H
3
4
5  #include "meanFilter.h"
6
7  class TurbineControl {
8  public:
9
10         void init(double lWind, double hWind, double hysteresis,
               double lRotor, int windFilterLength, int
               rotorFilterLength, double rho, double area, double
               cp_max, double nomEff);
11
12         void update(int cntrlMode, double rotorSpeed, double wind,
                double setEff);
13         int getCtrlMode();
14         double getBrake();
15         double getMaxEff();
16         double getDesEff();
17
18
19         //Brake constant
20         enum brake_modes { BRAKE_ON = 1, BRAKE_OFF = 0 };
21
22         //control mode constants
23         enum control_mode {SHT_DWN = 0, NOMEFF = 1, DCCONTR = 2,
               POWTRAC = 3, DELTA = 4};
24
25 private:
26         double lWind, hWind;
27         double hysteresis;
28
29         double lRotor;
30
31         int mode;
```

```
32          int intState;
33          double brake;
34
35          MeanFilter windFilter;
36          MeanFilter rotorFilter;
37
38          double rho, area, cp_max;
39          double nomEff;
40          double desEff;
41
42          //Internal state constants
43          enum int_state {ON = 1, OFF = 0, LWIND = -1};
44  };
45
46  #endif
```

Listing B.2: TurbineControl.cpp

```
1   #include "TurbineControl.h"
2   #include "PowerController.h"
3   #include <cmath>
4
5   void TurbineControl::init(double lWind, double hWind, double
        hysteresis, double lRotor, int windFilterLength, int
        rotorFilterLength, double rho, double area, double cp_max,
        double nomEff) {
6
7           this->lWind = lWind;
8           this->hWind = hWind;
9           this->lRotor = lRotor;
10          this->hysteresis = hysteresis;
11          this->rho = rho;
12          this->area = area;
13          this->cp_max = cp_max;
14          this->nomEff = nomEff;
15
16          mode = PowerController::SHT_DWN;
17          brake = BRAKE_ON;
18          intState = OFF;
19          desEff = 0;
20
21          windFilter.init(windFilterLength);
22          rotorFilter.init(rotorFilterLength);
23
24  }
25
26  void TurbineControl::update(int cntrlMode, double rotorSpeed,
        double wind, double setEff) {
27          rotorSpeed = rotorFilter.update(rotorSpeed);
28          wind = windFilter.update(wind);
29
30          if (cntrlMode == SHT_DWN) {
31                  intState = OFF;
32          }
33
```

```
34          if (intState == OFF || intState == LWIND) { // system
                should go to standstill, or is at standstill.
35                  mode = PowerController::SHT_DWN;
36                  if (rotorSpeed <= lRotor) { // brake should be
                        activated.
37                          brake = BRAKE_ON;
38                  } else if ((wind >= (hWind - hysteresis) ||
                        cntrlMode == SHT_DWN) && rotorSpeed <= 2*lRotor
                        ) {
39                          brake = (double)BRAKE_ON/5;
40                  }
41
42                  if (cntrlMode != SHT_DWN && wind >= (lWind +
                        hysteresis) && wind <= (hWind-hysteresis)) { //
                         Check if system should start.
43                          intState = ON;
44                          mode = PowerController::NOMEFF;
45                  }
46
47                  // Check if system is in SHT_DWN and why.
48                  if (wind >= (lWind + hysteresis) && wind <= (hWind
                        -hysteresis) && cntrlMode == SHT_DWN) {
49                          intState = OFF;
50                  } else if (cntrlMode == SHT_DWN) {
51                          intState = LWIND;
52                  }
53          }
54          if (intState == ON) {
55                  brake = BRAKE_OFF;
56                  if (wind <= (lWind) || wind >= (hWind)) { // Turn
                        of system
57                          intState = LWIND;
58                          mode = PowerController::SHT_DWN;
59                  }else if (cntrlMode == NOMEFF) {
60                          mode = PowerController::NOMEFF;
61                  } else if (cntrlMode == DCCONTR) {
62                          mode = PowerController::DCCONTR;
63                          desEff = setEff;
64                  } else if (cntrlMode == POWTRAC) {
65                          mode = PowerController::POWTRAC;
66                          desEff = setEff;
67                  } else if (cntrlMode == DELTA) {
68                          mode = PowerController::POWTRAC;
69                          desEff = 0.8 * getMaxEff();
70                  }
71          }
72  }
73
74
75  int TurbineControl::getCtrlMode() {
76          return mode;
77  }
78
79
80  double TurbineControl::getBrake() {
81          return brake;
```

```
82 }
83
84 double TurbineControl::getMaxEff() {
85         if (intState != LWIND) { // Estimate Maximum available
               power.
86                 double tmp = 0.5 * rho * area * pow(windFilter.get
                       (),3) * cp_max;
87                 if (tmp <= nomEff) {
88                         return tmp;
89                 } else {
90                         return nomEff;
91                 }
92         } else {
93                 return 0;
94         }
95 }
96
97 double TurbineControl::getDesEff() {
98         return desEff;
99 }
```

B.2 Generator Side Control

Listing B.3: GeneratorSideControl.h

```
1  #ifndef GENERATORSIDECONTROLLER_WITHOUT_PLANTCONTROLLER_H
2  #define GENERATORSIDECONTROLLER_WITHOUT_PLANTCONTROLLER_H
3
4  #include "PowerController.h"
5  #include "pid.h"
6  #include "currentContr.h"
7  #include "meanFilter.h"
8
9
10 class GeneratorSideController2 {
11
12         public:
13                 //initialize effectRegul
14                 void init_effectRegul(double c_init, double k_init
                       , double Ti_init, double w_ref, double
                       samplePeriod, double dp_lim, double dw_lim,
                       double ref_lim, double max_step, double min,
                       double max, double nomEff, double nomDC, double
                        k_pid, double Ti_pid, double Td_pid, double
                       Ni_pid,double yMax, double yMin, double iStart,
                        double yStart, int filterLength);
15
16                 //initialize speedController
17                 void init_speedController(double p, double Ti,
                       double Kd, double Ni, double yMax, double yMin,
                        double samplePeriod, double iStart, double
                       yStart, int errFilterLength, int inFilterLength
                       );
18
19                 //initialize currentController
20                 void init_currentController(double Ld, double Lq,
                       double Rs, double psim, double pp, double Ni,
                       double yMax, double yMin, double yStart,double
                       iStart, double samplePeriod, int filterLength);
21
22                 void update(int cntrlMode, double rotorSpeed,
                       double dcLevel, double power, double id_meas,
                       double iq_meas, double theta, double desEff);
23
24                 double getPRef();
25                 double getWRef();
26                 double getTauRef();
27                 double getUd();
28                 double getUq();
29
30                 int getState();
31
32         private:
33
34                 PowerController effectContr;
35                 PID speedContr;
36                 CurrentContr currentContr;
```

```
37                  MeanFilter filter;
38
39                  double w_ref, tau_ref, theta_old, samplePeriod;
40                  int countLimitEffect, countLimitSpeed;
41                  double tmp;
42                  int counter_effect, counter_speed;
43
44
45 };
46
47 #endif
```

Listing B.4: GeneratorSideController.cpp

```
1  #include "generatorSideController_without_plantController.h"
2
3  void GeneratorSideController2::init_effectRegul(double c_init,
       double k_init, double Ti_init, double w_ref, double
       samplePeriod, double dp_lim, double dw_lim, double ref_lim,
       double max_step, double min, double max, double nomEff, double
       nomDC, double k_pid, double Ti_pid, double Td_pid, double
       Ni_pid,double yMax, double yMin, double iStart, double yStart,
       int filterLength) {
4          //initialize effectRegul
5          effectContr.init(c_init, k_init, Ti_init, w_ref,
               samplePeriod, dp_lim, dw_lim, ref_lim, max_step, min,
               max, nomEff, nomDC, k_pid, Ti_pid, Td_pid, Ni_pid, yMax
               , yMin, iStart, yStart, filterLength);
6
7          this->countLimitEffect = int(samplePeriod/0.001);
8          this->counter_effect = countLimitEffect;
9          this->w_ref = 0;
10 }
11
12 void GeneratorSideController2::init_speedController(double p,
       double Ti, double Kd, double Ni, double yMax, double yMin,
       double samplePeriod, double iStart, double yStart, int
       errFilterLength, int inFilterLength) {
13         //initialize speedController
14         speedContr.init(p, Ti, Kd, Ni, yMax, yMin, samplePeriod,
               iStart, yStart, errFilterLength);
15
16         filter.init(inFilterLength);
17
18         for (int i = 0; i < inFilterLength; i++) {
19                 filter.update(0);
20         }
21
22         this->countLimitSpeed = int(samplePeriod/0.001);
23         this->counter_speed = countLimitSpeed;
24         this->tau_ref = yMin;
25 }
26
27 void GeneratorSideController2::init_currentController(double Ld,
       double Lq, double Rs, double psim, double pp, double Ni, double
        yMax, double yMin, double yStart,double iStart, double
```

```
    samplePeriod, int filterLength) {
28          //initialize currentController
29          currentContr.init(Ld, Lq, Rs, psim, pp, Ni, yMax, yMin,
                yStart, iStart, samplePeriod, filterLength);
30          this->samplePeriod = samplePeriod;
31          this->theta_old = 0;
32
33  }
34
35
36  void GeneratorSideController2::update(int ctrlMode, double
        rotorSpeed, double dcLevel, double power, double id_meas,
        double iq_meas, double theta, double desEff) {
37
38  /*
39  * uttermost loop, effect loop, get w_ref
40  */
41          //double omega = (theta - theta_old)/samplePeriod;
42
43          if(counter_effect == countLimitEffect) { // second
                uttermost loop, effect loop
44                  effectContr.update(ctrlMode, power, rotorSpeed,
                        dcLevel, desEff);
45                  w_ref = effectContr.get();
46                  counter_effect = 0;
47          }
48  /*
49  * second uttermost loop, speed loop, get tau_ref
50  */
51          if(counter_speed == countLimitSpeed) { //outer lopp, speed
                loop
52                  tmp = filter.update(w_ref);
53                  speedContr.update(tmp - rotorSpeed);
54                  tau_ref = speedContr.get();
55                  counter_speed = 0;
56          }
57
58
59  /*
60  * inner loop, current loop, get ud_ref and uq_ref
61  */
62          currentContr.update(tau_ref, id_meas, iq_meas, rotorSpeed)
                ; // inner loop, current loop
63
64          theta_old = theta;
65
66          // increment counters
67          ++counter_effect;
68          ++counter_speed;
69  }
70
71  double GeneratorSideController2::getUd() {
72          return currentContr.getId();
73  }
74
75  double GeneratorSideController2::getUq() {
```

```
76          return currentContr.getIq();
77  }
78
79  int GeneratorSideController2::getState() {
80          return effectContr.getState();
81  }
82
83  double GeneratorSideController2::getPRef() {
84          return effectContr.getPRef();
85  }
86
87  double GeneratorSideController2::getWRef() {
88          return tmp;
89  }
90
91  double GeneratorSideController2::getTauRef() {
92          return tau_ref;
93  }
```

B.3 Power Control

Listing B.5: EffRegul.h

```
#ifndef POWERCONTROLLER_H
#define POWERCONTROLLER_H
#include "pid.h"
#include "pi.h"

class PowerController {
        public:

        void init(double c_init, double k_init, double Ti_init,
            double w_ref, double samplePeriod, double dp_lim,
            double dw_lim, double ref_lim, double max_step, double
            min, double max, double nomEff, double nomDC, double
            k_pid, double Ti_pid, double Td_pid, double Ni_pid,
            double yMax, double yMin, double iStart, double yStart,
             int filterLength);
        void update(int ctrlMode, double p, double w, double
            dcLevel, double desEff);
        double get();
        int getState();
        double getPRef();

        enum control_mode {SHT_DWN = 0, NOMEFF = 1, DCCONTR = 2,
            POWIRAC = 3};

private:
        void leftSlope(double e);
        void rightSlope(double dp, double w, double e);
        void updateOutput(double p, double w, double dp, double dw
            , double p_ref, double e_ref, double e, int cntrlMode);
        double k, Ti;   // PI control variables
        double I;               // Integral
        double c;               // Control variable
        double min, max; // Min max values
        double dw_lim, dp_lim; // minimum change limits
        double w_old, p_old;    //old measurements
        double ref_lim;
        double p_ref_old;
        double y, yStart;
        double max_step;
        int last_state;
        bool p_ref_changed;
        int stop_pre, stop;
        double nomEff, nomDC, p_ref;
        double DC_gain;

        Pi contr;

        PID dcContr;
};

#endif
```

Listing B.6: EffRegul.cpp

```
1  #include "PowerController.h"
2  #include <math.h>
3
4  void PowerController::init(double c_init, double k_init, double
       Ti_init, double w_ref, double samplePeriod, double dp_lim,
       double dw_lim, double ref_lim, double max_step, double min,
       double max,double nomEff, double nomDC, double k_pid, double
       Ti_pid, double Td_pid, double Ni_pid,double yMax_pid, double
       yMin_pid, double iStart_pid, double yStart_pid, int
       filterLength) {
5
6
7          dcContr.init(k_pid, Ti_pid, Td_pid, Ni_pid, yMax_pid,
               yMin_pid, samplePeriod, iStart_pid, yStart_pid,
               filterLength);
8          this->c = c_init;
9          this->dp_lim = dp_lim;
10         this->dw_lim = dw_lim;
11         this->min = min;
12         this->max = max;
13         this->ref_lim = ref_lim;
14         this->max_step = max_step;
15         last_state = 0;
16         p_old = 0;
17         w_old = 0;
18         this->yStart = w_ref;
19         y = 0;
20         p_ref_old = 0;
21         I = 0;
22         Ti = Ti_init;
23         k = k_init;
24         p_ref_changed = false;
25         stop_pre = 1;
26         stop = 1;
27         this->nomEff = nomEff;
28         this->nomDC = nomDC;
29         p_ref = 0;
30         DC_gain = k_pid;
31 }
32
33 void PowerController::update(int ctrlMode, double p, double w,
       double dcLevel, double desEff) {
34         double dp = p-p_old;
35         double dw = w-w_old;
36         double e = 0;
37         double e_ref = y - w;
38
39         double error;
40         switch (ctrlMode) {
41         case SHT_DWN:
42                 stop = 1;
43                 p_ref = 0;
44                 break;
45         case NOMEFF:
```

```
46                  p_ref = nomEff;
47                  stop = 0;
48                  break;
49          case DCCONTR:
50                  stop = 0;
51
52                  error = nomDC - dcLevel;
53
54
55                  if (error >= 200) {
56                          p_ref = nomEff;
57                  } else if (error >= 100) {
58                          p_ref = desEff + 400;
59                  } else if (error <= -350) {
60                          p_ref = 0;
61                  } else if (error <= -100) {
62                          p_ref = desEff - 400;
63                  } else {
64                          p_ref = desEff + DC_gain*error;
65                  }
66                  break;
67          case POWTRAC:
68                  p_ref = desEff;
69                  stop = 0;
70                  break;
71          default:
72                  // Do nothing
73                  break;
74          }
75
76
77          if (p_ref < 0) {
78                  p_ref = 0;
79          } else if (p_ref > nomEff) {
80                  p_ref = nomEff;
81          }
82
83          e = p_ref - p;
84
85          updateOutput(p, w, dp, dw, p_ref, e_ref, e, ctrlMode);
86  }
87
88  double PowerController::get() {
89          return y;
90  }
91
92  void PowerController::leftSlope(double e) {
93
94          I += e/Ti;
95          double foo = k*e + I;
96
97          double temp = foo - y;
98          if (temp < -max_step) {
99                  temp = -max_step;
100         } else if (temp > max_step) {
101                 temp = max_step;
```

```
102         }
103         foo = y + temp;
104
105         if (foo > max) {
106                 y = max;
107         } else if (foo < min) {
108                 y = min;
109         } else {
110                 y = foo;
111         }
112
113         I = y - e*k;
114
115
116         last_state = 1;
117 }
118
119 void PowerController::rightSlope(double dp, double w, double e) {
120
121         double foo = w - abs(dp*c);
122         double temp = foo - y;
123         if (temp < -max_step) {
124                 temp = -max_step;
125         } else if (temp > max_step) {
126                 temp = max_step;
127         }
128         y += temp;
129
130         I = y - e*k; // Output leftSlope = output rightSlope
131         last_state = 2;
132 }
133
134 int PowerController::getState() {
135         return last_state;
136 }
137
138 double PowerController::getPRef() {
139         return p_ref;
140 }
141
142 void PowerController::updateOutput(double p, double w, double dp,
    double dw, double p_ref, double e_ref, double e, int cntrlMode)
     {
143         if (stop == 1) { // The Turbine Control has ordered
              standstill.
144                 if ( -ref_lim < e_ref) {
145                         //Decrease y to standstill
146                         y -= 30;
147                         if (y < 0) {
148                                 y = 0;
149                         }
150                         p_old = p;
151                         w_old = w;
152                         p_ref_old = p_ref;
153                 }
154                 stop_pre = stop;
```

```
155                     return;
156             } else if (stop_pre == 1) { // The plant controller has
                    decided to start/restart the plant
157                     y = yStart;
158                     stop_pre = stop;
159                     p_old = p;
160                     w_old = w;
161                     p_ref_old = p_ref;
162                     return;
163             }
164             if (p_ref_old != p_ref) {
165                     p_ref_changed = true;
166             }
167             if ((-ref_lim < e_ref && e_ref < ref_lim)) {// w has
                    changed
168                     if (p_ref_changed) {
169                             if (last_state <= 1) {
170                                     leftSlope(e);
171                             } else {
172                                     rightSlope(dp, w, e);
173                             }
174                             p_ref_changed = false;
175
176                     } else if (e < -dp_lim){
177                             leftSlope(e);
178                     } else if (dp < -dp_lim) {
179                             if ( dw > dw_lim) {
180                                     rightSlope(dp, w, e);
181                             } else if (dw < -dw_lim) {
182                                     leftSlope(e);
183                             } else {
184                                     if (dp < -10*dp_lim && last_state
                                            <= 1) {
185                                             leftSlope(e);
186                                     } else if (dp < -10*dp_lim){
187                                             rightSlope(dp, w, e);
188                                     }
189                                     w = w_old;
190                                     p = p_old;
191                             }
192                     } else if (dp > dp_lim) {
193                             if (dw < -dw_lim) {
194                                     rightSlope(dp, w, e);
195                             } else if (dw > dw_lim) {
196                                     leftSlope(e);
197                             } else {
198                                     if (dp > 10*dp_lim && last_state
                                            <= 1){
199                                             leftSlope(e);
200                                     } else if (dp > 10*dp_lim) {
201                                             rightSlope(dp, w, e);
202                                     }
203                                     w = w_old;
204                                     p = p_old;
205                             }
206                     } else{
```

```
207                             if (last_state <= 1) {
208
209                             } else {
210                                     }
211                                     // Do not update p_old/w_old
212                             w = w_old;
213                             p = p_old;
214                     }
215
216             } else {
217                     double foo = y;
218                     leftSlope(e);
219                     y=foo;
220
221                     // Do not update p_old/w_old
222                     w = w_old;
223                     p = p_old;
224             }
225
226
227             // limit output to min/max
228             if (y > max) {
229                     y = max;
230             } else if (y < min) {
231                     y = min;
232             }
233
234             p_old = p;
235             w_old = w;
236             stop = stop_pre;
237
238             p_ref_old = p_ref;
239     }
```

B.4 Mean Filter

Listing B.7: MeanFilter.h

```
#ifndef MEANFILTER_H
#define MEANFILTER_H

class MeanFilter {
public:

        void init(int length);
        double update(double u);
        double get();
        void reset();

private:
        static const int max = 10000;
        double vec[max];
        int startCounter;
        int filterLength;
        double y;
};
#endif
```

Listing B.8: MeanFilter.cpp

```
#include "meanFilter.h"


void MeanFilter::init(int length) {
        startCounter = 0;
        y = 0;

        if (length > max) {
                filterLength = max;
        } else if (length < 0) {
                filterLength = 0;
        } else
                filterLength = length;
}

double MeanFilter::update(double u) {
        if (startCounter < filterLength) {
                // Average output over the values obtained so far.
                vec[filterLength - ++startCounter] = u;
                if (startCounter > 1) {
                        u = 0;
                        for (int i = 1; i <= startCounter; i++) {
                                u += vec[filterLength - i];
                        }
                }
                return u/startCounter;
        }


        double tmp1 = 0;
```

```
31          double tmp2 = 0;
32          double sum = u;
33
34          // Update array.
35          for (int i = 0; i < filterLength; i++) {
36
37                  sum += vec[i];
38
39                  if (i == 0) {
40                          tmp1 = vec[i];
41                          vec[i] = u;
42                  } else {
43                          tmp2 = vec[i];
44                          vec[i] = tmp1;
45                          tmp1 = tmp2;
46                  }
47          }
48
49          y = sum/(filterLength + 1);
50          return y;
51  }
52
53  double MeanFilter::get() {
54          return y;
55  }
56
57  void MeanFilter::reset() {
58          startCounter = 0;
59          y = 0;
60  }
```

B.5 PI-Controller

Listing B.9: PI.h

```
#ifndef PI_H
#define PI_H
#include "piParam.h"

class Pi {
public:

        void init(double k,double Ti, double Ni, double yMax,
            double yMin, double samplePeriod,double iStart, double
            yStart);
        void update(double error);
        double get();
        void setI(double desired_y, double error);
        void setParam(PiParam param);
        PiParam getParam();
private:
        double I; // integrator-part
    double yMax, yMin; // output limits of the controller
        double k; // controller-gain
        double Ti; // timeconstant for the integrator
        double samplePeriod;
        double Ni; //Ni*Ti is the time constant of antiwindup
            compensation
        double iStart, yStart;
        double y; //output
};
#endif
```

Listing B.10: PI.cpp

```
#include "pi.h"



void Pi::init(double k, double Ti, double Ni, double yMax, double
    yMin, double samplePeriod, double iStart, double yStart) {
        this->k = k;
        this->Ti = Ti;
        this->yMax = yMax;
        this->yMin = yMin;
        this->samplePeriod = samplePeriod;
        this->Ni = Ni;
        this->iStart = iStart;
        this->yStart = yStart;
        I = iStart;
        y = yStart;
}

void Pi::update(double error) {
        double temp;

        double antiwindup;
```

```
22
23          temp = k*(error + (samplePeriod/Ti)*I);
24
25
26          if(temp >= yMax) {
27                  y = yMax;
28          } else if(temp <= yMin) {
29                  y = yMin;
30          } else {
31                  y = temp;
32          }
33
34          antiwindup = (y - temp)/(k*Ni);
35
36          I = I + error + antiwindup;
37
38
39  }
40
41  double Pi::get() {
42          return y;
43  }
44
45  void Pi::setI(double desired_y, double error){
46          I = (desired_y/k - error)*(Ti/samplePeriod);
47  }
48
49  void Pi::setParam(PiParam param) {
50          this->k = param.k;
51          this->Ti = param.Ti;
52          this->yMax = param.yMax;
53          this->yMin = param.yMin;
54          this->samplePeriod = param.samplePeriod;
55          this->Ni = param.Ni;
56          I = param.iStart;
57          y = param.yStart;
58  }
59
60  PiParam Pi::getParam() {
61          return PiParam().setParam(k, Ti, Ni, yMax, yMin,
                samplePeriod, iStart, yStart);
62  }
```

Listing B.11: PiParam.h

```
1  #ifndef PIPARAM_H
2  #define PIPARAM_H
3
4  class PiParam {
5  public:
6          PiParam setParam(double k,double Ti, double Ni, double
               yMax,double yMin, double samplePeriod,double iStart,
               double yStart);
7          PiParam getParam();
8
```

```
9           double k, Ti, Ni, yMax, yMin, samplePeriod, iStart, yStart
                ;
10  private:
11
12  };
13
14  #endif
```

Listing B.12: PiParam.cpp

```
1   #include "piParam.h"
2
3   PiParam PiParam::setParam(double k, double Ti, double Ni, double
        yMax, double yMin, double samplePeriod, double iStart, double
        yStart) {
4           this->k = k;
5           this->Ti = Ti;
6           this->Ni = Ni;
7           this->yMax = yMax;
8           this->yMin = yMin;
9           this->samplePeriod = samplePeriod;
10          this->iStart = iStart;
11          this->yStart = yStart;
12          return *this;
13  }
14
15  PiParam PiParam::getParam() {
16          return *this;
17  }
```

B.6 PID-Controller

Listing B.13: PID.h

```
#ifndef PID_H
#define PID_H

#include "meanFilter.h"

class PID {

public:
        void init(double p, double Ti, double Kd, double Ni,
           double yMax, double yMin, double samplePeriod, double
           iStart, double yStart, int filterLength);
        void update(double error);
        void setOutput(double out, double error);
        double get();

private:
        double p, Ti, Kd;
        double Ni;
        double yMax, yMin;
        double samplePeriod;
        double iStart, yStart;
        double y, I, e_old;

        MeanFilter filter;
};
#endif
```

Listing B.14: PID.cpp

```
#include "pid.h"

void PID::init(double p, double Ti, double Kd, double Ni, double
    yMax, double yMin, double samplePeriod, double iStart, double
    yStart, int filterLength) {
        this->p= p;
        this->Ti = Ti;
        this->Kd = Kd;
        this->Ni = Ni;
        this->yMax = yMax;
        this->yMin = yMin;
        this->samplePeriod = samplePeriod;

        y = yStart;
        I = iStart;
        e_old = 0;

        filter.init(filterLength);
}

void PID::update(double e) {

        e = filter.update(e);
```

```
22
23          double der = (e - e_old)/samplePeriod;
24          double temp = p*(e + (samplePeriod/Ti)*I + Kd*der);
25
26          if (temp >= yMax) {
27                  y = yMax;
28          } else if (temp <= yMin) {
29                  y = yMin;
30          } else {
31                  y = temp;
32          }
33
34          double antiWindup = (y - temp)/(p*Ni);
35
36          I = I + e + antiWindup;
37          e_old = e;
38  }
39
40  double PID::get() {
41          return y;
42  }
43
44  void PID::setOutput(double out, double error) {
45          y = out;
46          e_old = error;
47          I = (y/p - error) * (Ti/samplePeriod);
48          filter.reset();
49  }
```

B.7 Current Controller

Listing B.15: CurrentControl.h

```
#ifndef CURRENTCONTR_H
#define CURRENTCONTR_H

#include "id.h"
#include "iq.h"
#include "meanFilter.h"



class CurrentContr {
public:

        void init(double Ld, double Lq, double Rs, double psim,
                double pp, double Ni, double yMax, double yMin,
                    double yStart,
                double iStart, double samplePeriod, int
                    filterLength);

        void update(double tauRef, double id_meas, double iq_meas,
            double omega);
        double getId();
        double getIq();


private:
        double psim, pp; // magnetization and number of pole pairs
             in the generator
        Id id;
        Iq iq;

        MeanFilter filter;
};

#endif
```

Listing B.16: CurrentControl.cpp

```
#include "currentContr.h"

void CurrentContr::init(double Ld, double Lq, double Rs, double
    psim,
        double pp, double Ni, double yMax, double yMin, double
            yStart, double iStart,
        double samplePeriod, int filterLength) {

        // Create the current controllers
        id = Id();
        iq = Iq();

        // Initialize the current controllers
        id.init(yMax, yMin, Ld, Lq, Rs, Ni, samplePeriod, yStart,
            iStart);
```

```
13          iq.init(yMax, yMin, Ld, Lq, Rs, psim, Ni, samplePeriod,
                yStart, iStart);
14
15          // Initialize magnitization and number of pole pairs.
16          this->psim = psim;
17          this->pp = pp;
18
19          filter.init(filterLength);
20
21  }
22
23  void CurrentContr::update(double tauRef, double id_meas, double
        iq_meas, double omega) { // omega should be rotational speed of
         the rotor, not the electrical rotational speed
24
25          tauRef = filter.update(tauRef);
26
27          // Calculate id and iq references
28          double id_ref = 0;
29          double iq_ref = tauRef/(pp*psim);
30
31          double err_id = id_ref - id_meas;
32          double err_iq = iq_ref - iq_meas;
33
34          /* update the current controller
35           * since the current controller uses the electrical
36           * rotational speed the physical speed needs to be
37           * multiplied with the
38           * number of pole pairs in the generator. */
39          id.update(err_id, iq_meas, pp*omega);
40          iq.update(err_iq, id_meas, pp*omega);
41
42  }
43
44  double CurrentContr::getId() {
45          return id.get();
46  }
47
48
49  double CurrentContr::getIq() {
50          return iq.get();
51  }
```

B.8 Id/Iq-Controller

Listing B.17: Id.h

```
1  #ifndef ID_H
2  #define ID_H
3
4  class Id {
5  public:
6
7          void init(double yMax, double yMin, double Ld, double Lq,
               double Rs, double Ni,double samplePeriod,double yStart,
                double iStart);
8          void update(double error, double iqMeas, double omega);
9          double get();
10
11
12 private:
13         int id;
14         double yMax, yMin; // output limits of the controller
15         double k; // controller-gain
16         double Ti; // timeconstant for the integrator
17         double I; // I-part
18         double samplePeriod;
19         double Lq; //
20         double Ni; //Ni*Ti is the time constant of antiwindup
               compensation
21         double y; //output
22 };
23
24 #endif
```

Listing B.18: Id.cpp

```
1  #include "id.h"
2
3  void Id::init(double yMax, double yMin, double Ld, double Lq,
       double Rs, double Ni,double samplePeriod,double yStart, double
       iStart) {
4          this->yMax = yMax;
5          this->yMin = yMin;
6          this->samplePeriod = samplePeriod;
7          this->Lq = Lq;
8          this->Ni = Ni;
9          I = iStart;
10         Ti = (Ld/Rs + samplePeriod/2)*2;
11         k = (Ld/samplePeriod+Rs/2)/2;
12         y = yStart;
13 }
14
15 void Id::update(double error, double iqMeas, double omega) {
16         double temp;
17         double antiwindup;
18         temp = k*(error + samplePeriod/Ti*I) - omega*Lq*iqMeas;
19
20 // check limits
```

```
21          if (temp > yMax) {
22                  y = yMax;
23          } else if (temp < yMin) {
24                  y = yMin;
25          } else{
26                  y=temp;
27          }
28          antiwindup = (y - temp)/(k*Ni);
29          I = I + error + antiwindup; // update integrator part
30
31  }
32
33  double Id::get() {
34          return y;
35  }
```

Listing B.19: Iq.h

```
1   #ifndef IQ_H
2   #define IQ_H
3
4   class Iq {
5   public:
6
7
8           void init(double yMax, double yMin,double Ld, double Lq,
                double Rs,double psim, double Ni, double samplePeriod,
                double yStart, double iStart);
9           void update(double error, double id_meas, double omega);
10          double get();
11
12  private:
13          int id;
14          double yMax, yMin; // output limits of the controller
15          double k; // controller-gain
16          double Ti; // timeconstant for the integrator
17          double I; // I-part
18          double samplePeriod;
19          double Ld; //
20          double Ni; //Ni*Ti is the time constant of antiwindup
                compensation
21          double psim;
22          double y; //output
23  };
24  #endif
```

Listing B.20: Iq.cpp

```
1   #include "iq.h"
2
3
4   void Iq::init(double yMax, double yMin,double Ld, double Lq,
        double Rs,double psim, double Ni, double samplePeriod,double
        yStart, double iStart)
5   {
6           this->yMax = yMax;
```

```
7           this->yMin = yMin;
8           this->Ti = (Lq/Rs + samplePeriod/2)*2;
9           this->k = (Lq/samplePeriod + Rs/2)/2;
10          this->samplePeriod = samplePeriod;
11
12          this->Ld = Ld;
13          this->Ni = Ni;
14          this->psim = psim;
15          I = iStart;
16          y = yStart;
17  }
18
19  void Iq::update(double error, double idMeas, double omega) {
20          double temp;
21          double antiwindup;
22
23          temp = k*(error + samplePeriod/Ti*I) + omega*(psim + Ld*
                idMeas);
24
25          if (temp > yMax) {
26                  y = yMax;
27          } else if (temp < yMin) {
28                          y = yMin;
29          } else {
30                  y = temp;
31          }
32
33          antiwindup = (y - temp) / (k*Ni);
34          I = I + error + antiwindup;
35  }
36
37  double Iq::get() {
38          return y;
39  }
```

B.9 GridSideControllerOuter

Listing B.21: GridSideControllerOuter.h

```
1  #ifndef GRIDSIDECONTROLLEROUTER_H
2  #define GRIDSIDECONTROLLEROUTER_H
3
4  #include "gridCurrentController.h"
5  #include "gridVoltageController.h"
6  #include "pi.h"
7  #include "meanFilter.h"
8
9
10 class GridSideControllerOuter {
11 public:
12
13         void init(double Vdc_start, double Vdc_stop, double
               nomGridVd, double nomGridVq,
14                 double k_ud, double Ti_ud, double Ni_ud, double
                       yMax_ud, double yMin_ud, double iStart_ud,
                       double yStart_ud,
15                 double k_uq, double Ti_uq, double Ni_uq, double
                       yMax_uq, double yMin_uq, double iStart_uq,
                       double yStart_uq,
16                 double k_dc, double Ti_dc, double Td_dc, double
                       Ni_dc, double yMax_dc, double yMin_dc, double
                       iStart_dc, double yStart_dc, int filterLength,
17                 double k_ac, double Ti_ac, double Ni_ac, double
                       yMax_ac, double yMin_ac, double iStart_ac,
                       double yStart_ac,
18                 double k_id, double Ti_id, double Ni_id, double
                       yMax_id, double yMin_id, double iStart_id,
                       double yStart_id,
19                 double k_iq, double Ti_iq, double Ni_iq, double
                       yMax_iq, double yMin_iq, double iStart_iq,
                       double yStart_iq,
20                 double k_rctv, double Ti_rctv, double Ni_rctv,
                       double yMax_rctv, double yMin_rctv, double
                       iStart_rctv, double yStart_rctv,
21                 int powfilt, double sampleperiod_rctv, double
                       samplePeriod_outer, double samplePeriod_inner);
22         void update(int contrMode, double Vdc, double ud, double
               uq, double Vdc_ref, double ud_ref, double uq_ref,
               double id, double iq, double gridEff);
23         double getUdRef();
24         double getUqRef();
25         double getIdRef();
26         double getIqRef();
27         double getP();
28         double getQRef();
29         int getState();
30
31         // Control mode constants
32         enum control_mode {INACTIVE = 0, DCLEVEL = 1, ISLAND = 2,
               RCTVCOMP = 3};
33
```

```
34 private:
35
36         GridCurrentController CurrContr;
37         GridVoltageControl gridContr;
38         Pi rctvComp;
39
40         MeanFilter powerFilter;
41
42         int counter;
43         int cntr_rctv;
44
45         int cntr_rctv_limit;
46         int      counter_limit;
47
48         int preMode;
49         double Vdc_start, Vdc_stop;
50         double nomGridVd, nomGridVq;
51         double id_ref, iq_ref;
52         double Q_ref;
53         int int_state_inner;
54 };
55 #endif
```

Listing B.22: GridSideControllerOuter.cpp

```
1  #include "gridSideControllerOuter.h"
2
3  void GridSideControllerOuter::init(double Vdc_start, double
       Vdc_stop, double nomGridVd, double nomGridVq,
4                  double k_ud, double Ti_ud, double Ni_ud, double
                       yMax_ud, double yMin_ud, double iStart_ud,
                       double yStart_ud,
5                  double k_uq, double Ti_uq, double Ni_uq, double
                       yMax_uq, double yMin_uq, double iStart_uq,
                       double yStart_uq,
6                  double k_dc, double Ti_dc, double Td_dc, double
                       Ni_dc, double yMax_dc, double yMin_dc, double
                       iStart_dc, double yStart_dc, int filterLength,
7                  double k_ac, double Ti_ac, double Ni_ac, double
                       yMax_ac, double yMin_ac, double iStart_ac,
                       double yStart_ac,
8                  double k_id, double Ti_id, double Ni_id, double
                       yMax_id, double yMin_id, double iStart_id,
                       double yStart_id,
9                  double k_iq, double Ti_iq, double Ni_iq, double
                       yMax_iq, double yMin_iq, double iStart_iq,
                       double yStart_iq,
10                 double k_rctv, double Ti_rctv, double Ni_rctv,
                       double yMax_rctv, double yMin_rctv, double
                       iStart_rctv, double yStart_rctv,
11                 int powFilt, double sampleperiod_rctv, double
                       samplePeriod_outer, double samplePeriod_inner)
                       {
12
13
14                         /*
```

```
15                              *Initialize the current controller.
16                              */
17                              CurrContr.init(nomGridVd, nomGridVq, k_id,
                                     Ti_id, Ni_id, yMax_id, yMin_id,
                                    iStart_id, yStart_id,
18                                       k_iq, Ti_iq, Ni_iq, yMax_iq,
                                            yMin_iq, iStart_iq, yStart_iq,
                                            samplePeriod_inner);
19
20                              /*
21                              *Initialize the voltage controller.
22                              */
23                              gridContr.init(Vdc_start, Vdc_stop, k_ud,
                                    Ti_ud, Ni_ud, yMax_ud, yMin_ud,
                                    iStart_ud, yStart_ud,
24                                       k_uq, Ti_uq, Ni_uq, yMax_uq,
                                            yMin_uq, iStart_uq, yStart_uq,
25                                       k_dc, Ti_dc, Td_dc, Ni_dc, yMax_dc
                                            , yMin_dc, iStart_dc, yStart_dc
                                            , filterLength,
26                                       k_ac, Ti_ac, Ni_ac, yMax_ac,
                                            yMin_ac, iStart_ac, yStart_ac,
                                            samplePeriod_outer, CurrContr);
27
28                              /*
29                              *Initialize the controller for reactive
                                    compensation.
30                              */
31                              rctvComp.init(k_rctv, Ti_rctv, Ni_rctv,
                                    yMax_rctv, yMin_rctv, sampleperiod_rctv
                                    , iStart_rctv, yStart_rctv);
32
33                              //Initialize the counters and counters
                                    limits.
34                              cntr_rctv_limit = int (sampleperiod_rctv/
                                    samplePeriod_inner);
35                              counter_limit = int (samplePeriod_outer/
                                    samplePeriod_inner);
36                              counter = cntr_rctv_limit;
37                              cntr_rctv = counter_limit;
38
39                              this->Vdc_start = Vdc_start;
40                              this->Vdc_stop = Vdc_stop;
41                              this->int_state_inner = DCLEVEL;
42
43                              this->iq_ref = 0;
44                              this->id_ref = 0;
45                              this->nomGridVd = nomGridVd;
46                              this->nomGridVq = nomGridVq;
47                              preMode = INACTIVE;
48                              powerFilter.init(powFilt);
49                              Q_ref = 0;
50  }
51
52  void GridSideControllerOuter::update(int contrMode, double Vdc,
        double ud, double uq, double Vdc_ref, double ud_ref, double
```

```
    uq_ref, double id, double iq, double gridEff) {
53          bool setIdRef = false;
54          if ((contrMode == ISLAND) && (preMode != ISLAND)) {
55                  CurrContr.setIsland();
56          } else if (contrMode == RCTVCOMP && preMode !=RCTVCOMP) {
57                  CurrContr.setDcLev();
58                  setIdRef = true;
59                  rctvComp.setI(uq*id - ud*iq, 0);
60          } else if (contrMode != ISLAND && preMode != DCLEVEL) {
61                  CurrContr.setDcLev();
62          }
63          if (cntr_rctv >= cntr_rctv_limit) { // time to update the
                reactive power compensation loop
64                  if (contrMode == RCTVCOMP) {
65                          rctvComp.update(nomGridVd - ud);
66                          Q_ref = rctvComp.get();
67                  } else {
68                          Q_ref = 0;
69                  }
70                  cntr_rctv = 0;
71          }
72
73          if (counter >= counter_limit) { // time to update voltage
                loop
74                  double p = id*ud + iq*uq;
75                  if (Vdc > Vdc_stop) {
76                          powerFilter.update(p);
77                  }
78                  gridContr.update(contrMode, Vdc, ud, uq, Vdc_ref,
                        ud_ref, Q_ref, id, iq, gridEff); //*1.05
79                  iq_ref = gridContr.getIqRef();
80                  if (setIdRef) {
81                          id_ref = (p - iq_ref*uq)/ud;
82                          gridContr.setIdI(id_ref, Vdc_ref - Vdc);
83                          setIdRef = false;
84                  } else {
85                          id_ref = gridContr.getIdRef();
86                  }
87
88                  int_state_inner = gridContr.getState();
89                  counter = 0;
90          }
91
92          CurrContr.update(int_state_inner, id, iq, id_ref, iq_ref);
93          if (int_state_inner == INACTIVE) {
94                  rctvComp.setI(0,0);
95                  Q_ref=0;
96          }
97          ++counter;
98          ++cntr_rctv;
99          preMode = contrMode;
100
101 }
102
103 double GridSideControllerOuter::getUdRef() {
104         return CurrContr.getUdRef();
```

```
105 }
106
107 double GridSideControllerOuter::getUqRef() {
108         return CurrContr.getUqRef();
109 }
110
111
112 double GridSideControllerOuter::getIdRef() {
113         return id_ref;
114 }
115
116 double GridSideControllerOuter::getIqRef() {
117         return iq_ref;
118 }
119
120
121 int GridSideControllerOuter::getState() {
122         return int_state_inner;
123 }
124
125 double GridSideControllerOuter::getP() {
126         return powerFilter.get();
127 }
128
129 double GridSideControllerOuter::getQRef() {
130         return Q_ref;
131 }
```

B.10 GridVoltageController

Listing B.23: GridVoltageController.h

```
1  #ifndef GRIDVOLTAGECONTROLLER_H
2  #define GRIDVOLTAGECONTROLLER_H
3
4  #include "pi.h"
5  #include "pid.h"
6  #include "gridCurrentController.h"
7
8  class GridVoltageControl {
9  public:
10
11         void init(double Vdc_start, double Vdc_stop,
12                 double k_ud, double Ti_ud, double Ni_ud, double
                       yMax_ud, double yMin_ud, double iStart_ud,
                       double yStart_ud,
13                 double k_uq, double Ti_uq, double Ni_uq, double
                       yMax_uq, double yMin_uq, double iStart_uq,
                       double yStart_uq,
14                 double k_dc, double Ti_dc, double Td_dc, double
                       Ni_dc, double yMax_dc, double yMin_dc, double
                       iStart_dc, double yStart_dc, int filterLength,
15                 double k_ac, double Ti_ac, double Ni_ac, double
                       yMax_ac, double yMin_ac, double iStart_ac,
                       double yStart_ac,
16                 double samplePeriod, GridCurrentController
                       currContr);
17         void update(int contrMode, double Vdc, double ud, double
               uq, double Vdc_ref, double ud_ref, double uq_ref,
               double id, double iq, double eff);
18         double getIdRef();
19         double getIqRef();
20         int getState();
21         void setIdI(double desired_y, double error);
22
23         // Control mode constants
24         enum control_mode {INACTIVE = 0, DCLEVEL = 1, ISLAND = 2,
               RCTVCOMP = 3};
25
26 private:
27         Pi ud_contr;
28         Pi uq_contr;
29         PID Vdc_contr;
30         Pi Vac_contr;
31         GridCurrentController currContr;
32
33         double Vdc_start, Vdc_stop;
34         int int_state;
35 };
36 #endif
```

Listing B.24: GridVoltageController.cpp

```
1  #include "gridVoltageController.h"
```

```
2
3  void GridVoltageControl::init(double Vdc_start, double Vdc_stop,
4          double k_ud, double Ti_ud, double Ni_ud, double yMax_ud,
               double yMin_ud, double iStart_ud, double yStart_ud,
5          double k_uq, double Ti_uq, double Ni_uq, double yMax_uq,
               double yMin_uq, double iStart_uq, double yStart_uq,
6          double k_dc, double Ti_dc, double Td_dc, double Ni_dc,
               double yMax_dc, double yMin_dc, double iStart_dc,
               double yStart_dc, int filterLength,
7          double k_ac, double Ti_ac, double Ni_ac, double yMax_ac,
               double yMin_ac, double iStart_ac, double yStart_ac,
8          double samplePeriod, GridCurrentController currContr) {
9
10                 // Initialize the voltage controllers.
11                 ud_contr.init(k_ud, Ti_ud, Ni_ud, yMax_ud, yMin_ud
                       , samplePeriod, iStart_ud, yStart_ud);
12                 uq_contr.init(k_uq, Ti_uq, Ni_uq, yMax_uq, yMin_uq
                       , samplePeriod, iStart_uq, yStart_uq);
13
14                 // Initialize the V_dc and V_ac controller.
15                 Vdc_contr.init(k_dc, Ti_dc, Td_dc, Ni_dc, yMax_dc,
                        yMin_dc, samplePeriod, iStart_dc, yStart_dc,
                       filterLength);
16                 Vac_contr.init(k_ac, Ti_ac, Ni_ac, yMax_ac,
                       yMin_ac, samplePeriod, iStart_ac, yStart_ac);
17
18                 this->Vdc_start = Vdc_start;
19                 this->Vdc_stop = Vdc_stop;
20                 this->int_state = INACTIVE;
21                 this->currContr = currContr;
22 }
23
24 void GridVoltageControl::update(int cntrlMode, double Vdc, double
       ud, double uq, double Vdc_ref, double ud_ref, double uq_ref,
       double id, double iq, double eff) {
25         bool update = true;
26         if (eff != -1) {
27                 update = ud * id + uq * iq < eff;
28         }
29         if (cntrlMode == INACTIVE) {
30                 int_state = INACTIVE;
31         }else if (cntrlMode == DCLEVEL) { // Controller should
               control the DC-voltage level
32                 double q = uq*id - ud*iq;
33                 if (int_state == ISLAND) { // Set integrator to
                       avoid unnecesarry bumps in the control signal.
34                         Vdc_contr.setOutput(ud_contr.get(),Vdc_ref
                               - Vdc);
35                         Vac_contr.setI(uq_contr.get(), uq_ref - q)
                               ;
36                 }
37
38                 if (int_state != DCLEVEL) {
39                         int_state = INACTIVE;
40                         if (Vdc >=Vdc_start) {
41                                 int_state = DCLEVEL;
```

```
42                          }
43                  }
44
45                  if (int_state == DCLEVEL) {
46
47                          if (Vdc <= Vdc_stop) {
48                                  int_state = INACTIVE;
49                          } else {
50                                  double err = Vdc_ref - Vdc;
51                                  double errAC = uq_ref - q;
52                                  Vac_contr.update(errAC);
53                                  if (err >= 0 || update) {
54                                          Vdc_contr.update(err);
55                                  } else {
56                                          Vdc_contr.setOutput((eff -
                                                uq*iq)/ud, err);
57                                  }
58
59                                  int_state = DCLEVEL;
60                          }
61                  }
62
63
64          } else if(cntrlMode == ISLAND) { // Controller should
                operate in ISLAND mode.
65
66                  if (int_state != ISLAND) { // Check if the
                        controller is in ISLAND mode or not
67                          if (int_state == DCLEVEL || int_state ==
                               RCTVCOMP) {
68                                  ud_contr.setI(Vdc_contr.get(),
                                        ud_ref-ud);
69                                  uq_contr.setI(Vac_contr.get(),
                                        uq_ref-uq);
70                          }
71
72                          int_state = INACTIVE;
73                          if (Vdc >= Vdc_start) { // Check if DC-
                                voltage level is sufficient to enter
                                ISLAND mode.
74                                  int_state = ISLAND;
75                          }
76
77                  }
78                  if (int_state == ISLAND) { // Controller is
                        operating in ISLAND mode.
79                          if (Vdc <= Vdc_stop) { // Controller in
                                ISLAND mode, check if DC-voltage level
                                is sufficient to continue running in
                                ISLAND mode.
80                                  int_state = INACTIVE;
81                          } else {
82                                  double errd = ud_ref - ud;
83                                  double errq = uq_ref - uq;
84
85                                  ud_contr.update(errd);
```

```
86                                      uq_contr.update(errq);
87                              }
88                      }
89              } else if(cntrlMode == RCTVCOMP) {
90                      bool ind = false;
91                      if (int_state != RCTVCOMP) { // Check if the
                            controller is in ISLAND mode or not
92                              if (int_state == ISLAND) {
93                                      Vdc_contr.setOutput(ud_contr.get()
                                            , Vdc_ref - Vdc);
94                                      Vac_contr.setI(uq_contr.get(),
                                            uq_ref - uq);
95                              } else if (int_state == DCLEVEL) {
96                                      ind = true;
97                              }
98
99                              int_state = INACTIVE;
100                             if (Vdc >= Vdc_start) { // Check if DC-
                                    voltage level is sufficient to enter
                                    ACLEVEL control mode.
101                                     int_state = RCTVCOMP;
102                             }
103
104                     }
105
106                     if (int_state == RCTVCOMP) {
107                             double q = uq*id - ud*iq;
108                             if (Vdc <= Vdc_stop) { // Controller does
                                    not have enough DC-voltage to continue
                                    in ACLEVEL mode.
109                                     int_state = INACTIVE;
110                             } else {
111                                     double errDC = Vdc_ref - Vdc;
112                                     double errAC = uq_ref - q;
113                                     Vac_contr.update(errAC);
114                                     if (ind) {
115                                             double id_ref = (ud*id +
                                                    uq*iq - uq*getIqRef())/
                                                    ud;
116                                             Vdc_contr.setOutput(id_ref
                                                    , errDC);
117                                     }else if (errDC >= 0 || update) {
118                                             Vdc_contr.update(errDC);
119                                     } else {
120                                             Vdc_contr.setOutput((eff -
                                                    uq*iq)/ud, errDC);
121                                     }
122
123                             }
124                     }
125             }
126
127 }
128
129 double GridVoltageControl::getIdRef() {
130             if (int_state == DCLEVEL || int_state == RCTVCOMP) {
```

```
131                 return Vdc_contr.get();
132         } else if (int_state == ISLAND) {
133                 return ud_contr.get();
134         } else {
135                 return 0;
136         }
137 }
138
139 double GridVoltageControl::getIqRef() {
140         if (int_state == RCTVCOMP || int_state == DCLEVEL) {
141                 return 2*Vac_contr.get()/(3*currContr.getUdRef());
142         } else if (int_state == ISLAND) {
143                 return uq_contr.get();
144         } else {
145                 return 0;
146         }
147 }
148
149 int GridVoltageControl::getState() {
150         return int_state;
151 }
152
153 void GridVoltageControl::setIdI(double desired_y, double error) {
154         Vdc_contr.setOutput(desired_y, error);
155 }
```

B.11 GridCurrentController

Listing B.25: GridCurrentController.h

```
1  #ifndef GRIDCURRENTCONTROLLER_H
2  #define GRIDCURRENTCONTROLLER_H
3
4  #include "pi.h"
5
6  class GridCurrentController {
7  public:
8
9          void init(double nomGridVd, double nomGridVq, double k_d,
               double Ti_d, double Ni_d, double yMax_d, double yMin_d,
                double iStart_d, double yStart_d,
10                  double k_q, double Ti_q, double Ni_q, double
                       yMax_q, double yMin_q, double iStart_q, double
                       yStart_q, double samplePeriod);
11         void update(int cntrlMode, double id, double iq, double
               id_ref, double iq_ref);
12         double getUdRef();
13         double getUqRef();
14         void setDcLev();
15         void setIsland();
16
17         enum control_mode {INACTIVE = 0, DCLEVEL = 1, ISLAND = 2,
              RCTVCOMP = 3};
18 private:
19         Pi id_contr;
20         Pi iq_contr;
21
22         PiParam param;
23         int int_state;
24         double nomGridVd, nomGridVq;
25 };
26
27 #endif
```

Listing B.26: GridCurrentController.cpp

```
1  #include "gridCurrentController.h"
2
3  void GridCurrentController::init(double nomGridVd, double
       nomGridVq, double k_d, double Ti_d, double Ni_d, double yMax_d,
        double yMin_d, double iStart_d, double yStart_d,
4          double k_q, double Ti_q, double Ni_q, double yMax_q,
               double yMin_q, double iStart_q, double yStart_q, double
                samplePeriod) {
5
6          id_contr = Pi();
7          iq_contr = Pi();
8
9          // Initialize current controllers.
10         id_contr.init(k_d, Ti_d, Ni_d, yMax_d, yMin_d,
               samplePeriod, iStart_d, yStart_d);
```

```
11          iq_contr.init(k_q, Ti_q, Ni_q, yMax_q, yMin_q,
                samplePeriod, iStart_q, yStart_q);
12          param = iq_contr.getParam();
13
14          this->nomGridVd = nomGridVd;
15          this->nomGridVq = nomGridVq;
16
17          int_state = DCLEVEL;
18  }
19
20  void GridCurrentController::update(int cntrlMode, double id,
        double iq, double id_ref, double iq_ref) {
21          int_state = cntrlMode;
22
23          if (int_state == INACTIVE) {
24                  id_contr.setI(0,0);
25                  iq_contr.setI(0,0);
26          } else {
27                  double errd = id_ref - id;
28                  double errq = iq_ref - iq;
29
30                  id_contr.update(errd);
31                  iq_contr.update(errq);
32          }
33  }
34
35  double GridCurrentController::getUdRef() {
36          if (int_state == INACTIVE) {
37                  return 0;
38          } else {
39                  return id_contr.get() + nomGridVd;
40          }
41  }
42
43  double GridCurrentController::getUqRef() {
44          if (int_state == INACTIVE) {
45                  return 0;
46          } else {
47                  return iq_contr.get() + nomGridVq;
48          }
49  }
50
51  void GridCurrentController::setDcLev() {
52          iq_contr.setParam(param);
53  }
54
55  void GridCurrentController::setIsland() {
56          iq_contr.setParam(id_contr.getParam());
57  }
```

B.12 Plant Control

Listing B.27: PlantControl.h

```
1  #ifndef PLANTCONTROL_H
2  #define PLANTCONTROL_H
3
4
5  class PlantControl {
6
7  public:
8          void init(int nmbrOfUnits, double losses, double
               samplePeriod, double coolDownTime);
9
10         //Set the right mode for each powerplant, { dclevel,
               effect}
11         void update(int mode, double desEff, double power, bool
               rctvComp, const double maxEff[]);
12         int getGridMode();
13         int* getPlantMode();
14         double* getDesEff();
15         double getGridEff();
16
17
18         enum mode { SHTDWN = 0, MAXEFF = 1, ISLAND = 2, DESEFF =
               3, DELTA = 4 };
19
20 private:
21
22         const static int maxCapacity = 100;
23         int unitMode[maxCapacity];
24         double desEff[maxCapacity];
25         int nmbrOfUnits;
26         double samplePeriod;
27
28         double power;
29         double losses;
30
31         double cooldownTime, count;
32         int controlModeGrid; // mode to gridside.
33
34         double gridEff;
35
36         void setAllUnitMode(int mode, const double maxEff[]);
37         void splitDesEff(double desEff, const double maxEff[]);
38         void splitIsland(const double maxEff[]);
39         void setIslandInit(const double maxEff[]);
40 };
41
42 #endif
```

Listing B.28: PlantControl.cpp

```
1  #include "PlantControl.h"
2  #include "gridVoltageController.h"
3  #include "TurbineControl.h"
```

```
4
5  void PlantControl::init(int nmbrOfUnits, double losses, double
       samplePeriod, double coolDownTime){
6          if (maxCapacity >= nmbrOfUnits){
7                  this->nmbrOfUnits = nmbrOfUnits;
8                  for (int i = 0; i < nmbrOfUnits;  i++) {
9                          unitMode[i] = TurbineControl::SHT_DWN;
10                         desEff[i] = 0;
11                 }
12         } else {
13                 this->nmbrOfUnits = -1;
14         }
15
16         this->losses = losses;
17         this->cooldownTime = coolDownTime;
18         this->samplePeriod = samplePeriod;
19         controlModeGrid = GridVoltageControl::INACTIVE;
20         count  = 0;
21         gridEff = -1;
22 }
23
24 void PlantControl::update(int mode, double desEff, double power,
       bool rcvtComp, const double maxEff[]) {
25         this->power = power;
26         gridEff = -1;
27         switch (mode) {
28         case SHTDWN:
29                 controlModeGrid = GridVoltageControl::INACTIVE;
30                 setAllUnitMode(TurbineControl::SHT_DWN, maxEff);
31                 break;
32         case MAXEFF:
33                 if (rcvtComp) {
34                         controlModeGrid = GridVoltageControl::
                            RCTVCOMP;
35                 } else {
36                         controlModeGrid = GridVoltageControl::
                            DCLEVEL;
37                 }
38                 setAllUnitMode(TurbineControl::NOMEFF, maxEff);
39                 break;
40         case ISLAND:
41                 controlModeGrid = GridVoltageControl::ISLAND;
42                 if (power > 100) {
43                         if (count >= cooldownTime) {
44                                 splitIsland(maxEff);
45                         } else {
46                                 count += samplePeriod;
47                                 setIslandInit(maxEff);
48                         }
49                 } else {
50                         count = 0;
51                         setIslandInit(maxEff);
52                 }
53                 controlModeGrid = GridVoltageControl::ISLAND;
54                 break;
55         case DESEFF:
```

```
56                  gridEff = desEff;
57                  if (rcvtComp) {
58                          controlModeGrid = GridVoltageControl::
                               RCTVCOMP;
59                  } else {
60                          controlModeGrid = GridVoltageControl::
                               DCLEVEL;
61                  }
62                  splitDesEff(desEff, maxEff);
63                  break;
64          case DELTA:
65                  if (rcvtComp) {
66                          controlModeGrid = GridVoltageControl::
                               RCTVCOMP;
67                  } else {
68                          controlModeGrid = GridVoltageControl::
                               DCLEVEL;
69                  }
70                  setAllUnitMode(TurbineControl::DELTA, maxEff);
71                  break;
72          default:
73                  //Do nothing.
74                  break;
75          }
76  }
77
78  int PlantControl::getGridMode() {
79          return controlModeGrid;
80  }
81
82  int* PlantControl::getPlantMode() {
83          return unitMode;
84  }
85
86  double* PlantControl::getDesEff() {
87          return desEff;
88  }
89
90  void PlantControl::setAllUnitMode(int mode, const double maxEff[])
        {
91          for (int i = 0; i < nmbrOfUnits; i++) {
92                  unitMode[i] = mode;
93                  switch (mode) {
94                  case TurbineControl::NOMEFF:
95                          if (maxEff[i] > 0) {
96                                  desEff[i] = maxEff[i];
97                          } else {
98                                  desEff[i] = 0;
99                                  unitMode[i] = TurbineControl::
                                       SHT_DWN;
100                         }
101                         break;
102                 case TurbineControl::SHT_DWN:
103                         desEff[i] = 0;
104                         break;
105                 case TurbineControl::DELTA:
```

```
106                         if (maxEff[i] > 0) {
107                                 desEff[i] = 0.8 * maxEff[i]; //
                                        For good measure.
108                 } else {
109                                 desEff[i] = 0;
110                                 unitMode[i] = TurbineControl::
                                        SHT_DWN;
111                         }
112                         break;
113                 default:
114                         //do nothing.
115                         break;
116                 }
117         }
118 }
119
120 void PlantControl::splitDesEff(double desEff, const double maxEff
        []) {
121         int activeUnits = 0;
122         double potEff = 0;
123         desEff = desEff/(1 - losses);
124         while (potEff < desEff*1.2 && activeUnits < nmbrOfUnits) {
125                 potEff += maxEff[activeUnits++];
126         }
127
128         for (int i = 0 ; i < activeUnits ; i++) {
129                 if (maxEff[i] > 0) {
130                         unitMode[i] = TurbineControl::POWTRAC;
131                         this->desEff[i] = (maxEff[i]/potEff) *
                                desEff;
132                 } else {
133                         unitMode[i] = TurbineControl::SHT_DWN;
134                         this->desEff[i] = 0;
135                 }
136         }
137
138         for (int i = activeUnits ; i < nmbrOfUnits ; i++) {
139                 unitMode[i] = TurbineControl::SHT_DWN;
140                 this->desEff[i] = 0;
141         }
142 }
143
144 double PlantControl::getGridEff() {
145         return gridEff;
146 }
147
148 void PlantControl::splitIsland(const double maxEff[]) {
149         double minEff = power*1.6;  // Minimum 60% (-losses) more
                than consumed power for control margin.
150
151         int activeUnits = 0;
152         double potEff = 0;
153         while (potEff < minEff && activeUnits < nmbrOfUnits) {
154                 potEff += maxEff[activeUnits];
155                 if (maxEff[activeUnits++] > 0) {
156
```

```
157                     }
158             }
159
160             for (int i = 0 ; i < activeUnits ; i++) {
161                     if (maxEff[i] > 0) {
162                             unitMode[i] = TurbineControl::DCCONTR;
163                             this->desEff[i] = (maxEff[i]/potEff) *
                                    power/(1-losses);
164                     } else {
165                             unitMode[i] = TurbineControl::SHT_DWN;
166                             desEff[i] = 0;
167                     }
168             }
169
170             for (int i = activeUnits ; i < nmbrOfUnits ; i++) {
171                     unitMode[i] = TurbineControl::SHT_DWN;
172                     this->desEff[i] = 0;
173             }
174     }
175
176     void PlantControl::setIslandInit(const double maxEff[]) {
177             for (int i = 0; i < nmbrOfUnits; i++) {
178                     unitMode[i] = TurbineControl::POWTRAC;
179                     if (maxEff[i] > 1000) {
180                             desEff[i] = 1000;
181                     } else if (maxEff[i] > 0) {
182                             desEff[i] = maxEff[i];
183                     } else {
184                             desEff[i] = 0;
185                             unitMode[i] = TurbineControl::
                                    SHT_DWN;
186                     }
187             }
188     }
```