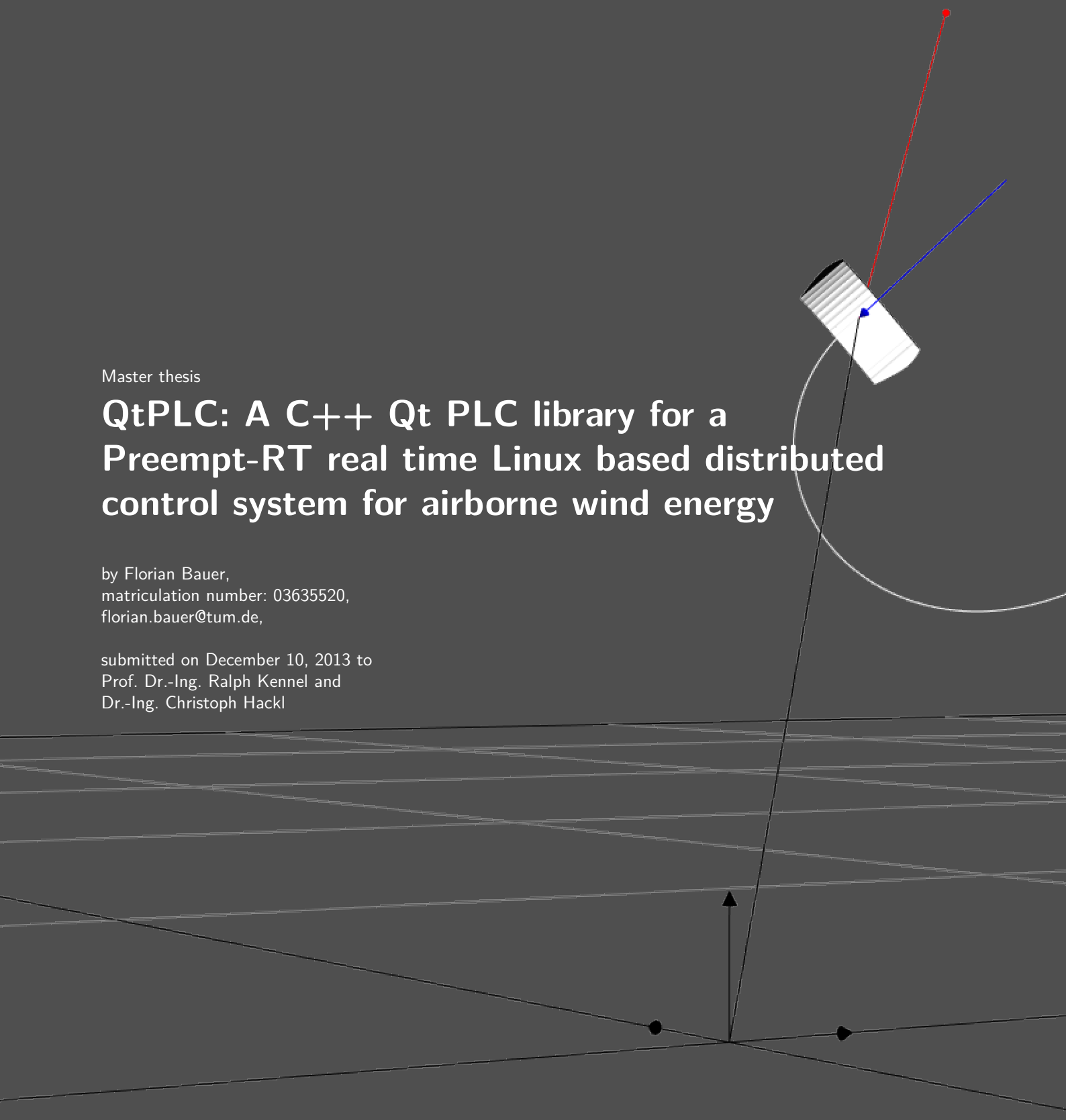
Institute for Electrical Drive Systems and Power Electronics,
Department of Electrical Engineering and Information Technology,
Technische Universität München



Master thesis

QtPLC: A C++ Qt PLC library for a Preempt-RT real time Linux based distributed control system for airborne wind energy

by Florian Bauer,
matriculation number: 03635520,
florian.bauer@tum.de,

submitted on December 10, 2013 to
Prof. Dr.-Ing. Ralph Kennel and
Dr.-Ing. Christoph Hackl

Keywords: C++, Qt, PLC, library, Linux, Preempt-RT, real time, distributed control system, airborne wind energy, renewable energy, embedded computer, Raspberry Pi, BeagleBone Black, open source.

Abstract

Airborne wind energy (AWE) uses a tethered airborne to harvest the wind energy. The highest power densities are achieved if the airborne is a rigid or flexible wing and if it is flown in crosswind motions like circles or figure eights. Energy is produced either in a pumping process with a ground based generator or continuously with onboard turbines. Similar output powers compared to classical wind energy are achievable already today. In order to build an AWE prototype for future university research, a distributed programmable logic controller (PLC) is needed, because controllers, sensors and actuators may be on ground and on the airborne. For this, the following PLC approach is proposed: Only inexpensive ordinary computers and embedded computers like the BeagleBone Black or the Raspberry Pi are used. All computers involved run a Linux with the Preempt-RT real time patch. With that it is possible to start any application with real time priority only with a special command, i.e. no special library is needed. Therefore the cross platform and open source C++ Qt framework in combination with the Qt Creator integrated development environment (IDE) is used. The nodes of the distributed PLC communicate via standard Internet Protocol (IP) links, so that Qt as underlying framework alone is sufficient. These communication links facilitate that it is arbitrary which and how many nodes run on any computer. For the same reason, the whole PLC software can be developed and run software-in-the-loop on a single computer. To ease the development and to encapsulate repeating tasks of the distributed PLC, a library based on Qt was developed and is thus named "QtPLC". Special focus was laid on a clean and intuitive design of the QtPLC application programming interface (API). The library also provides an extensible visualizer API: the QtPLC Control Center. The presented PLC approach is compared to the actual requirements of a PLC for this application and evaluated to fulfill all specifications. It is also compared to the solutions of other AWE researchers and companies. The approach was tested successfully with an AWE model with ground based generator which was exported from Matlab/Simulink to C++. It was executed on a Raspberry Pi with a Preempt-RT patched Raspbian in hard real time. The QtPLC Control Center with keyboard control was executed on a MacBook and communicated with the Raspberry Pi via ethernet. Consequently, the presented PLC approach with the developed QtPLC library seems to be suitable alternative of a PLC for a research prototype and for simulations.

Statement

This master thesis was realized at the Institute for Electrical Drive Systems and Power Electronics at the Technische Universität München (TUM) under the supervision of Dr.-Ing. Christoph Hackl. Absolutely no sections of this thesis were submitted or used to gain another academic degree. Hereby I state, that this thesis was written alone by myself, except the explicitly marked passages of the text.

Munich, December 9, 2013
Florian Bauer

Acknowledgments

I want thank Prof. Dr.-Ing. Ralph Kennel and Dr.-Ing. Christoph Hackl very much for mentoring and supervising my master thesis and for giving many helpful remarks! I really appreciate their openness for the new and unknown topic of airborne wind energy and the acceptance of my maybe non-standard approaches of finding solutions.

I also want to thank the airborne wind energy community. When I visited the *Airborne Wind Energy Conference 2013* in Berlin, I learned a lot from the presentations. I also got the privilege to meet and talk to the very friendly and open minded key experts of the field. I want to thank in particular Uwe Fechner, M.Sc., for his openness and willingness to share even unpublished information about his work on the kite control system of the TU Delft. With that I not only gained confidence in my PLC idea but I also got new input for improvements.

Not least, I want to thank my girlfriend Nadine Stappenbeck, M.Sc., for her love, her support and her proofreading!

Munich, December 9, 2013
Florian Bauer

Contents

Nomenclature

Symbol	Meaning
$\mathbb{N}$	$:= \{0, 1, 2, 3, \dots\}$, set of natural numbers
$\mathbb{N}_{>0}$	$:= \{1, 2, 3, \dots\}$, set of positive natural numbers
$\mathbb{R}$	$:= (-\infty, \infty)$, set of real numbers
$\mathbb{R}_{>0}$	$:= (0, \infty)$, set of positive real numbers
$\mathbb{R}_{\geq 0}$	$:= [0, \infty)$, set of positive real numbers including 0
In the following let $n, m \in \mathbb{N}_{>0}$.	
$\boldsymbol{x}$	$:= (x_1, x_2, \dots, x_n)^\top \in \mathbb{R}^n$ (column) vector with $x_i \in \mathbb{R}\ \forall i \in \{1, 2, \dots, n\}$, **all vectors are bold**
$\lvert\boldsymbol{x}\rvert$	$:= \sqrt{\boldsymbol{x}^\top \boldsymbol{x}} =: x$, the Euclidean norm (or 2-norm) of $\boldsymbol{x} \in \mathbb{R}^n$
x	$\in \mathbb{R}$, scalar, **all scalars are non-bold**
$\boldsymbol{X}$	$:= \begin{pmatrix} x_{11} & \dots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{nm} \end{pmatrix} \in \mathbb{R}^{n \times m}$, matrix with coefficients $x_{ij}\ \forall i \in \{1, 2, \dots, n\}\ \forall j \in \{1, 2, \dots, m\}$, **all matrices are upper case and bold**
x	$\in \mathbb{R}$, constant scalar, **all constants are non-italic**
$\operatorname{dir} \boldsymbol{x}$	$:= \frac{\boldsymbol{x}}{\lvert\boldsymbol{x}\rvert} \in \mathbb{R}^3 \cap \big([-1, 1], [-1, 1], [-1, 1]\big)^\top$, the direction or unit vector of $\boldsymbol{x} \in \mathbb{R}^n$
$\dot{x}$	$:= \frac{\mathrm{d}x}{\mathrm{d}t} \in \mathbb{R}$, first time derivative of $x \in \mathbb{R}$
$\ddot{x}$	$:= \frac{\mathrm{d}^2 x}{\mathrm{d}t^2} \in \mathbb{R}$, second time derivative of $x \in \mathbb{R}$
$\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$	$\in \mathbb{R}^3 \left[(1, 1, 1)^\top\right]$ axis unit vectors of the x, y and z axis of a cartesian coordinate system
α, β, γ	$\in \mathbb{R}\,[^\circ]$ Euler angles for the rotations around the x, y and z axis of a cartesian coordinate system, i.e. roll angle, pitch or elevation angle, yaw or azimuth angle
$\boldsymbol{x}^y$	$\in \mathbb{R}^3$, 3D vector with cartesian coordinates in the y fixed cartesian coordinate system where $y \in [\mathrm{b} = \text{body}, \mathrm{e} = \text{earth}, \mathrm{t} = \text{tether}]$

List of symbols

Symbol	Meaning
Latin symbols.	
A	$\in \mathbb{R}_{>0}\,[\mathrm{m}^2]$, projected wing area
$\boldsymbol{a}_\mathrm{b}$	$= \dot{\boldsymbol{v}}_\mathrm{b} = \ddot{\boldsymbol{r}}_\mathrm{b} \in \mathbb{R}^3\,[(\mathrm{m/s}^2, \mathrm{m/s}^2, \mathrm{m/s}^2)^\top]$, acceleration of the wing or body
c_ae	$:= \sqrt{(c_\mathrm{d,i} + c_\mathrm{d,t})^2 + c_\mathrm{l}^2} \in \mathbb{R}_{\geq 0}\,[1]$, combined aerodynamic coefficient
c_AWE	$:= \frac{c_\mathrm{ae}^3}{c_\mathrm{d,i}^2} \in \mathbb{R}_{\geq 0}\,[1]$, AWE coefficient
c_d	$:= c_\mathrm{d,i} + c_\mathrm{d,t} \in \mathbb{R}\,[1]$, drag coefficient
c_d^*	$\in \mathbb{R}\,[1]$, drag coefficient at optimal angle of attack α^*
$c_\mathrm{d,i}$	$\in \mathbb{R}_{>0}\,[1]$, intrinsic drag coefficient
$c_\mathrm{d,t}$	$\in \mathbb{R}\,[1]$, turbine drag coefficient
c_l	$\in \mathbb{R}\,[1]$, (intrinsic) lift coefficient
c_l^*	$\in \mathbb{R}\,[1]$, lift coefficient at optimal angle of attack α^*
c_t	$\in \mathbb{R}_{>0}\,[\mathrm{N/m}]$, spring constant of the tether
g	$= 9.81\,\mathrm{m/s}^2$, gravitational acceleration
$\boldsymbol{F}_\mathrm{a}$	$\in \mathbb{R}^3\,[(\mathrm{N}, \mathrm{N}, \mathrm{N})^\top]$, acceleration force vector
F_ae	$:= \lvert\boldsymbol{F}_\mathrm{ae}\rvert \in \mathbb{R}\,[\mathrm{N}]$, magnitude of the total aerodynamic force
$\boldsymbol{F}_\mathrm{ae}$	$\in \mathbb{R}^3\,[(\mathrm{N}, \mathrm{N}, \mathrm{N})^\top]$, total aerodynamic force vector
$\boldsymbol{F}_\mathrm{ae,d}$	$\in \mathbb{R}^3\,[(\mathrm{N}, \mathrm{N}, \mathrm{N})^\top]$, drag force vector
$F_\mathrm{ae,d,i}$	$\in \mathbb{R}\,[\mathrm{N}]$, magnitude of the intrinsic drag force
$\boldsymbol{F}_\mathrm{ae,l}$	$\in \mathbb{R}^3\,[(\mathrm{N}, \mathrm{N}, \mathrm{N})^\top]$, lift force vector
$\boldsymbol{F}_\mathrm{g}$	$\in \mathbb{R}^3\,[(\mathrm{N}, \mathrm{N}, \mathrm{N})^\top]$, gravitational force vector
F_t	$:= \lvert\boldsymbol{F}_\mathrm{t}\rvert \in \mathbb{R}\,[\mathrm{N}]$, magnitude of the tether force
$\boldsymbol{F}_\mathrm{t}$	$\in \mathbb{R}^3\,[(\mathrm{N}, \mathrm{N}, \mathrm{N})^\top]$, tether force vector
$I_\mathrm{m,ref}$	$\in \mathbb{R}\,[\mathrm{A}]$, reference current for the main electrical machine
l_t	$\in \mathbb{R}_{\geq 0}\,[\mathrm{m}]$, tether length
m_b	$\in \mathbb{R}_{>0}\,[\mathrm{kg}]$, mass of the wing or body
P	$\in \mathbb{R}\,[\mathrm{W}]$, power
P_l	$\in \mathbb{R}\,[\mathrm{W}]$, power losses
$\overline{P}_\mathrm{ref}$	$\in \mathbb{R}\,[\mathrm{W}]$, average reference power
P_w	$\in \mathbb{R}\,[\mathrm{W}]$, wind power
$P_\mathrm{w,e}$	$\in \mathbb{R}\,[\mathrm{W}]$, extracted wind power
$\boldsymbol{r}_\mathrm{b}$	$\in \mathbb{R}^3\,[(\mathrm{m}, \mathrm{m}, \mathrm{m})^\top]$, position vector of the wing or body
$\boldsymbol{r}_\mathrm{b,0}$	$\in \mathbb{R}^3\,[(\mathrm{m}, \mathrm{m}, \mathrm{m})^\top]$, initial position vector of the wing or body
$\boldsymbol{R}_{\boldsymbol{x}_\mathrm{e}}(\alpha)$	$:= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$, rotation matrix to rotate a vector with the roll angle α around the x axis of the earth fixed cartesian coordinate system, i.e. the $\boldsymbol{x}_\mathrm{e}$ unit vector

Symbol	Meaning
$\boldsymbol{R}_{\boldsymbol{y}_\mathrm{e}}(\beta)$	$:= \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}$, rotation matrix to rotate a vector with the pitch or elevation angle β around the y axis of the earth fixed cartesian coordinate system, i.e. the $\boldsymbol{y}_\mathrm{e}$ unit vector
$\boldsymbol{R}_{\boldsymbol{z}_\mathrm{e}}(\gamma)$	$:= \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$, rotation matrix to rotate a vector with the yaw or azimuth angle γ around the z axis of the earth fixed cartesian coordinate system, i.e. the $\boldsymbol{z}_\mathrm{e}$ unit vector
$\boldsymbol{R}_{\boldsymbol{z}_\mathrm{e}\boldsymbol{y}_\mathrm{e}\boldsymbol{x}_\mathrm{e}}(\gamma,\beta,\alpha)$	$:= \boldsymbol{R}_{\mathrm{z}_\mathrm{e}}(\gamma)\boldsymbol{R}_{\mathrm{y}_\mathrm{e}}(\beta)\boldsymbol{R}_{\mathrm{x}_\mathrm{e}}(\alpha)$, assembled x-y-z Euler angles transformation matrix
$\boldsymbol{s}$	$\in \mathbb{R}^n, n \in \mathbb{N}$, vector with shape parameters
t	$\in \mathbb{R}\,[\mathrm{s}]$, time
t_j	$\in \mathbb{R}\,[\mathrm{s}]$, jitter time
t_m	$\in \mathbb{R}_{\geq 0}\,[\mathrm{s}]$, message delay
t_i	$\in \mathbb{R}\,[\mathrm{s}]$, maximum jitter for the ith slave node with $i \in \mathbb{N}_{>0}$
$\boldsymbol{T}_{\mathrm{b}\to\mathrm{e}}$	$\in \mathbb{R}^{3\times3}$, transformation matrix to transform a force or velocity vector of the body fixed cartesian coordinate system to the earth fixed cartesian coordinate system
$\boldsymbol{T}_{\mathrm{e}\to\mathrm{b}}$	$= \boldsymbol{T}_{\mathrm{b}\to\mathrm{e}}^{-1} \in \mathbb{R}^{3\times3}$, transformation matrix to transform a force or velocity vector of the earth fixed cartesian coordinate system to the body fixed cartesian coordinate system
$\boldsymbol{T}_{\mathrm{t}\to\mathrm{e}}$	$\in \mathbb{R}^{3\times3}$, transformation matrix to transform a force or velocity vector of the tether fixed cartesian coordinate system to the earth fixed cartesian coordinate system
$\boldsymbol{v}_\mathrm{b}$	$= \dot{\boldsymbol{r}}_\mathrm{b} \in \mathbb{R}^3\,\left[(\mathrm{m/s},\mathrm{m/s},\mathrm{m/s})^\top\right]$, wing or body velocity
$\boldsymbol{v}_{\mathrm{b},0}$	$\in \mathbb{R}^3\,\left[(\mathrm{m/s},\mathrm{m/s},\mathrm{m/s})^\top\right]$, initial wing or body velocity
$\boldsymbol{v}_{\mathrm{b,r}}$	$\in \mathbb{R}^3\,\left[(\mathrm{m/s},\mathrm{m/s},\mathrm{m/s})^\top\right]$, radial portion of the wing or body velocity
$\boldsymbol{v}_{\mathrm{b,t}}$	$\in \mathbb{R}^3\,\left[(\mathrm{m/s},\mathrm{m/s},\mathrm{m/s})^\top\right]$, tangential portion of the wing or body velocity
v_r	$:= \lvert\boldsymbol{v}_\mathrm{r}\rvert \in \mathbb{R}\,[\mathrm{m/s}]$, apparent or relative wind speed
$\boldsymbol{v}_\mathrm{r}$	$\in \mathbb{R}^3\,\left[(\mathrm{m/s},\mathrm{m/s},\mathrm{m/s})^\top\right]$, apparent or relative wind velocity
v_t	$:= \lvert\boldsymbol{v}_\mathrm{t}\rvert \in \mathbb{R}\,[\mathrm{m/s}]$, tether speed
$\boldsymbol{v}_\mathrm{t}$	$\in \mathbb{R}^3\,\left[(\mathrm{m/s},\mathrm{m/s},\mathrm{m/s})^\top\right]$, tether velocity
$v_{\mathrm{t,out}}$	$\in \mathbb{R}\,[\mathrm{m/s}]$, tether reel out speed
v_w	$:= \lvert\boldsymbol{v}_\mathrm{w}\rvert$, wind speed
$\boldsymbol{v}_\mathrm{w}$	$\in \mathbb{R}^3\,\left[(\mathrm{m/s},\mathrm{m/s},\mathrm{m/s})^\top\right]$, wind velocity
$\boldsymbol{x}_\mathrm{b}, \boldsymbol{y}_\mathrm{b}, \boldsymbol{z}_\mathrm{b}$	$\in \mathbb{R}^3\,\left[(1,1,1)^\top\right]$, axis unit vectors of the x, y and z axis of the cartesian coordinate system of the wing or body fixed reference frame
$\boldsymbol{x}_\mathrm{e}, \boldsymbol{y}_\mathrm{e}, \boldsymbol{z}_\mathrm{e}$	$\in \mathbb{R}^3\,\left[(1,1,1)^\top\right]$, axis unit vectors of the x, y and z axis of the cartesian coordinate system of the earth fixed reference frame
$\boldsymbol{x}_\mathrm{w}, \boldsymbol{y}_\mathrm{w}, \boldsymbol{z}_\mathrm{w}$	$\in \mathbb{R}^3\,\left[(1,1,1)^\top\right]$, axis unit vectors of the x, y and z axis of the cartesian coordinate system of the wind fixed reference frame in which $\boldsymbol{x}_\mathrm{w}$ is parallel to the wind velocity vector $\boldsymbol{v}_\mathrm{w}$
Greek symbols.	
α	$\in \mathbb{R}\,[^\circ]$ angle of attack, roll angle
α^*	$\in \mathbb{R}\,[^\circ]$ optimal angle of attack
β_c	$\in \mathbb{R}\,[^\circ]$ pitch angle of the wing or body
$\beta_{\mathrm{c,ref}}$	$\in \mathbb{R}\,[^\circ]$ reference pitch angle of the wing or body

Symbol	Meaning
$\beta_\mathrm{e} = \beta_\mathrm{w}$	$\in \mathbb{R}\,[^\circ]$ elevation angle in the wind and the earth fixed coordinate system (they are equal)
$\beta_\mathrm{e,ref}$	$\in \mathbb{R}\,[^\circ]$ reference elevation angle in the earth fixed coordinate system
γ_c	$\in \mathbb{R}\,[^\circ]$ yaw angle of the wing or body
$\gamma_\mathrm{c,ref}$	$\in \mathbb{R}\,[^\circ]$ reference yaw angle of the wing or body
γ_e	$\in \mathbb{R}\,[^\circ]$ azimuth angle in the earth fixed coordinate system
$\gamma_\mathrm{e,ref}$	$\in \mathbb{R}\,[^\circ]$ reference azimuth angle in the earth fixed coordinate system
γ_w	$\in \mathbb{R}\,[^\circ]$ azimuth angle in the wind fixed coordinate system
Δl_t	$:= l_\mathrm{t} - \lvert\boldsymbol{r}_\mathrm{b}\rvert \in \mathbb{R}\,[\mathrm{m}]$ difference of the tether length and the norm of the wing's or body's position vector
ϵ	$\in \mathbb{R}\,[^\circ]$, angle between $-\boldsymbol{v}_\mathrm{r}$ and $\boldsymbol{F}_\mathrm{ae}$
ζ	$\in \mathbb{R}\,[1]$, power harvesting factor
λ	$:= \frac{v_\mathrm{r}}{v_\mathrm{w}} \in \mathbb{R}\,[1]$, wing speed ratio
λ^*	$\in \mathbb{R}\,[1]$, optimal wing speed ratio
ρ	$\in \mathbb{R}_{>0}\,\left[\mathrm{kg/m^3}\right]$, air density
$\boldsymbol{\varphi}$	$\in \mathbb{R}^3\,\left[(^\circ,^\circ,^\circ)^\top\right]$, orientation Euler angles
φ_m	$\in \mathbb{R}\,[^\circ]$, angular position of the main electrical machine
$\boldsymbol{\varphi}_\mathrm{s,ref}$	$\in \mathbb{R}^n\,[^\circ], n \in \mathbb{N}_{>0}$, vector of reference angular positions of the n steering electrical machines
$\boldsymbol{\chi}$	$\in \mathbb{R}^3$, velocity or force vector
ω_m	$\in \mathbb{R}\,[^\circ/\mathrm{s}]$, angular speed of the main electrical machine

List of abbreviations and terms

Abbreviation/term	Meaning
API	application programming interface
AWE	airborne wind energy
CAN	Controller Area Network, serial bus
Chap.	Chapter
DLL	dynamic linked library
DOF	degree of freedom
Fig.	Figure
GCC	GNU Compiler Collection
GDB	GNU Debugger, debugger software
GPIO	general purpose input output
GUI	graphical user interface
I2C or I^2C	Inter-Integrated Circuit, serial bus
IDE	integrated development environment
IMU	inertial measurement unit
IP	Internet Protocol
KISS principle	keep it simple and stupid, i.e simplicity is a key design goal and unnecessary complexity is avoided
Lst.	Listing
OSI model	Open Systems Interconnection model, network protocol layers model
p.	page
PLC	programmable logic controller
pp.	pages
Preempt-RT	real time patch for Linux
Qt	cross platform C++ framework
Qt Creator	IDE for Qt
QtPLC	name of the developed PLC library based on Qt
RS232	point to point serial bus
RTAI	Real Time Application Interface, real time patch for Linux
Sec.	Section
SPI	Serial Peripheral Interface, serial bus
SSH	Secure Shell, software to access a computer remotely over a network via a command line interface
TCP	Transmission Control Protocol, part of the Internet Protocol family
UDP	User Datagram Protocol, part of the Internet Protocol family
VNC	Virtual Network Computing, software to access a computer remotely over a network via a graphical user interface
WLAN	Wireless Local Area Network
Xenomai	real time patch for Linux

1 Motivation

Wind energy plants use the wings of a turbine to convert the energy of the wind into rotatory mechanical energy which is then converted into electrical energy by a generator. In fact, the wing tips harvest the majority of the wind energy [1, p. 5]. However, the material consuming parts, i.e. the foundation, in particular for offshore plants, the steel and concrete tower, the nacelle, where the heavy generator is placed in a height of up to 100 meters and above, are all necessary for that technology. The increased masses and volumes of the parts of those *classical* wind energy plants have not only a high footprint due to its production but also became a logistic challenge.

For about a decade, several research groups and small companies especially in Europe and the United States investigate the field of *airborne wind energy* (AWE) [1, p. xi], also known as *high altitude wind energy*. AWE uses a tethered airborne, i.e. a rigid or flexible wing, lighter than air parts like a helium filled balloon, zeppelin or similar, to harvest the wind energy in higher altitudes and with less material compared to classical wind energy. This thesis focuses on those concepts which use a tethered wing that is flown mainly perpendicular to the wind direction – i.e. crosswind – in circles or figure eights. Throughout this thesis, this is called crosswind AWE. With that the highest power densities are achieved [1, pp. 3]. These concepts may include one or more ground based or flying electrical machines, which can be operated in generator or motor mode by power electronics. Depending on the concept, besides the ground based electrical machine(s), only a tether and a wing, which can be a kite or a parachute, are needed. This was already enough to achieve a power output of 2 MW [2] by a commercial product.

It is envisaged to build a crosswind AWE prototype for research at the university. The control system, i.e. the programmable logic controller (PLC), is one key part of such a plant and must be distributed because sensors and actuators may be on ground and on the airborne. Since the fundings for university research prototypes are often limited, the costs for the PLC should be as low as possible. So in this thesis a PLC is proposed that consists only of inexpensive ordinary and embedded computers like the *BeagleBone Black* or the *Raspberry Pi*. All computers involved run a *Linux* with the *Preempt-RT* real time patch. With this specific patch it is possible to start any application with real time priority. Hence, it is possible to use the cross platform and open source C++ Qt framework, which has application programming interfaces (APIs) for all major tasks [3]. The nodes of the distributed PLC are proposed to communicate via standard Internet Protocol (IP) links, so that Qt as underlying framework alone is sufficient. With these communication links, it is arbitrary on which computer how many and which nodes run. For the same reason, the whole PLC software can be developed and run software-in-the-loop on a single computer. To ease the development and to encapsulate repeating tasks of the distributed PLC, a library based on Qt was developed and named "QtPLC".

In this thesis, first a theoretical background on crosswind AWE is given in Chap. 2. In Chap. 3, the idea of the proposed PLC is presented in detail and compared to the actual requirements of a PLC for a crosswind AWE prototype. This approach is also compared to solutions of AWE researchers and companies. In Chap. 4, a model of a crosswind AWE plant with ground based generator is derived and implemented in Matlab/Simulink. This model is then exported to C++ and used to implement QtPLC processes. Here, the QtPLC API is presented and background information of important QtPLC implementations are given. The performance of this PLC approach together with the QtPLC library is presented for two different test setups. Finally, conclusions and an outlook are given in Chaps. 5 and 6.

2 Theoretical background of crosswind airborne wind energy

This chapter gives a theoretical background about the physics of AWE with strong focus on crosswind AWE. The first book about AWE is "Airborne Wind Energy" by Uwe Ahrens, Moritz Diehl and Roland Schmehl (Eds.) from Springer and was recently published in late 2013. This book may be used as a wider introduction, but important facts will be found in this chapter: First a general overview of crosswind AWE is given, how such a system is operated and how electrical energy can be generated. This is followed by the derivation of the maximum power of AWE. Finally, some issues about the automation of a crosswind AWE plant are considered.

2.1 Overview

2.1.1 Electrical energy generation with crosswind AWE

The crosswind AWE concepts use a tethered wing, i.e. a body which is shaped so that it creates a high lift force. The magnitude of the lift force is proportional to the square of the air speed. To gain a high lift force, the wing is flown in a crosswind motion like circles or figure eights, for which the relative air speed is much higher than the wind speed alone and thus the lift force is much higher.

There are two possibilities to generate electrical energy which may also be combined: The first possibility is shown in Fig. 2.1. Here, the wing is tethered and the tether is attached to a winch which is driven by an electrical machine. Energy is generated in a pumping process: In the first phase the wing is flown in crosswind with a high lift force. The wing pulls the tether which is reeled out slowly and thus energy is produced. In the second phase the wing is flown in a low force position like the zenith or pitched down (no crosswind motion) and reeled back in, little energy is consumed. This approach is called lift mode [4], or throughout this thesis lift AWE, since the lift force of the wing is utilized directly.

In the second possibility the wing is tethered to the ground with constant tether length and has embedded electrical machines with turbines. The turbines break the wing like the generator breaks the wings of a classical wind energy plant. However, the turbines are exposed to the much higher relative wind speed through crosswind flight instead of the wind speed alone. This is called drag mode [4], or throughout this thesis drag AWE, since energy is generated by adding an additional drag to the wing.

With drag AWE energy can be generated continuously, but the whole energy is to be transmitted via the tether to the ground. With lift AWE the disadvantage of pumping energy production is compensated by the fact that the main electrical machine is on the ground and actually no sensor and no actuator is needed intrinsically on the wing. Thus, ordinary surf kites or parachutes with ordinary tethers can be used.

SkySails states that their ship propulsion system already achieved a (mechanical) power (equivalent) of 2 MW [2]. From this success it is assumed that at least the same magnitude of rated power is achievable for electrical crosswind AWE plants. So the power output is already today comparable to classical wind energy but, for lift AWE, by only using a generator on ground, a

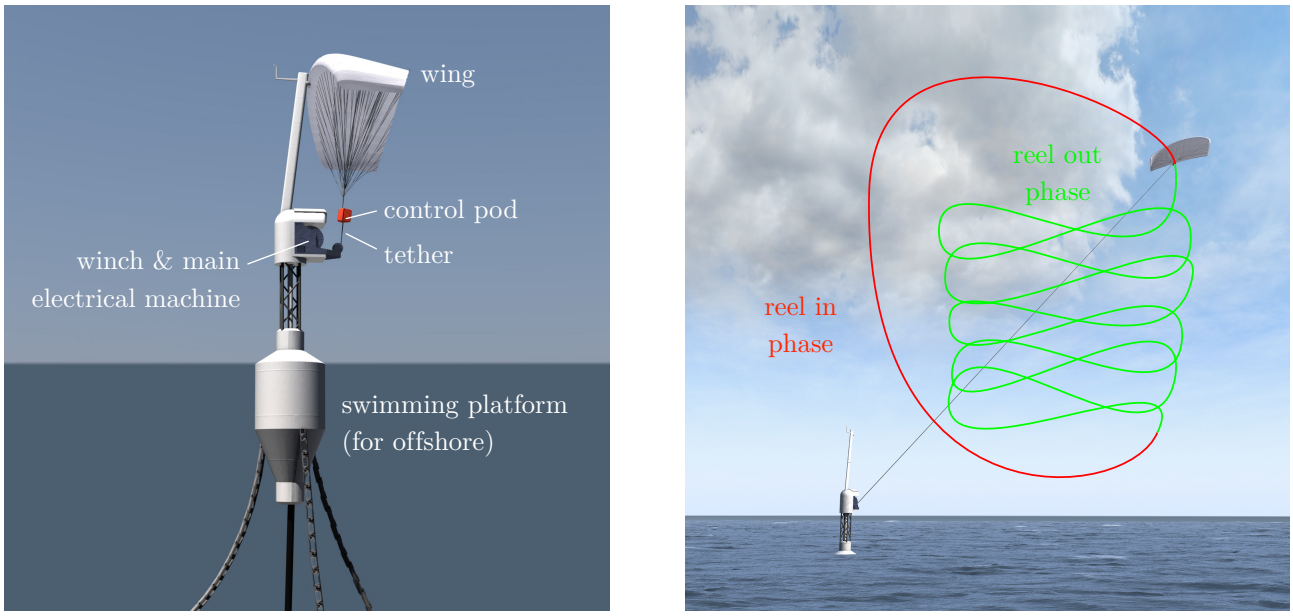


Figure 2.1: Example lift AWE plant visualizations from SkySails; system setup (left)[1] and pumping energy production process with reel out and reel in phase (right)[2].

tether and a wing – just enough to convert the wind energy in *proper* mechanical energy. Hence, such a system may be installed on a buoy-like swimming platform, as SkySails intents to, see Fig. 2.1 (left). By flying in higher altitudes where the winds are usually stronger the power output can be increased, which is impossible for classical wind energy and is another advantage of AWE.

2.1.2 Types of used wings

Possible wings for both crosswind AWE approaches are rigid airplanes, leading edge inflated kites and ram air inflated kites or parachutes, as shown in Fig. 2.2. The cross section profiles of all types are sketched below the photographs to emphasize that *all* these types – including the flexible ones – are wings and can produce a high lift force. So this property is the same as for modern classical wind energy plants using the lift principle instead of the drag principle. In order to control the flight direction of the wing, its geometry can be changed, i.e. a rigid airplane has controllable rudders and a kite or parachute has steering lines.

All types have their advantages and disadvantages. The differences are in the aerodynamic and mechanical properties. Different research groups and companies use different types. In this thesis the types are neither assessed nor was it necessary to chose a specific type in the simple presented models. However, the following list outlines some properties of the three mentioned types:

[1] Image source (edited): SkySails system rendering. SkySails GmbH, "Renderings SkySails Power", http://www.skysails.info/fileadmin/user_upload/Presselounge/power/sks_ps_print_07_20x30cm_300dpi.jpg, accessed: November 06, 2013.

[2] Image source (edited): SkySails phases rendering. SkySails GmbH, "Renderings SkySails Power", http://www.skysails.info/fileadmin/user_upload/Presselounge/power/sks_ps_print_03_20x30cm_300dpi.jpg, accessed: November 06, 2013.

Figure 2.2: Types of wings for crosswind AWE: rigid airplane (left)[3], leading edge inflated kite (middle)[4]and ram air inflated kite (right)[5]each with a sketched cross section profile.

1. *Rigid airplanes* always keep their shape, so only a single tether is needed. Rudders are used for steering, but need energy that is to be transmitted from the ground or generated onboard. Compared to flexible wings, the stiff shape can be better aerodynamically optimized. Rigid airplanes have the highest lift to drag ratios, which is a main factor of the maximum theoretical power. Electrical machine(s) with turbines can be embedded into the structure, which makes this type beneficial for drag AWE. However, there is no market for rigid airplanes for this application, so the whole rigid airplane has to be developed from scratch. Additionally, a single crash will lead to total loss of the airplane.

2. *Leading edge inflated kites* consist of a textile-like foil spanned on top from the leading edge main tube and smaller strut tubes to the rear. The tubes are inflated by pressurized air before the flight. Through the inflated tubes the kite mainly keeps its shape even after a crash and can be restarted. Because the kite has no inlets, this is even possible on water which is the reason why kite surfers use this type [5, p. 7]. In contrast to rigid airplanes, leading edge inflated kites have a lower lift to drag ratio [1, p. 18]. They may have one or two main tethers to transmit the high aerodynamic forces of the kite either to a control pod or directly to the ground. Additionally, they have two steering tethers to change slightly the shape with relatively low forces from the control pod or the ground. Electrical machine(s) for drag AWE cannot be embedded into the kite, but into a control pod. Even hard crashes on the ground are not likely to lead to serious damage to the kite.

3. *Ram air inflated kites* have air inlets at their leading edge and use the dynamic pressure of the streaming air to inflate their profile to the shape of a wing during the flight. So the disadvantage of ram air inflated kites over leading edge inflated kites is the need for air speed to gain and hold the shape and therefore the difficulty to start. It is almost impossible to restart a crashed ram air inflated kite especially on water when it gets filled

[3]Image source: Ampyx Power plane. Ampyx Power, http://www.ampyxpower.com/files/get/51, accessed: November 06, 2013.
[4]Image source: TU Delft kite. TU Delft, http://www.tudelft.nl/fileadmin/UD/MenC/Support/Internet/TU_Website/TU_Delft_portal/Onderzoek/Environment/Climate_City_Campus/img/kite4.jpg, accessed: Mai 17, 2013.
[5]Image source: SkySails kite. SkySails GmbH, http://www.skysails.info/fileadmin/user_upload/Presselounge/SkySails_Kite/11_Kite_300dpi_rgb.jpg, accessed: Mai 17, 2013.

with it. However, ram air inflated kites do not need to be inflated before the flight, are not that threatened by (small) leaks and never need to be reinflated manually. [5, p. 7] Ram air inflated kites have a slightly higher lift to drag ratio [1, pp. 289], but steering, drag AWE possibility and crash damage are similar to leading edge inflated kites.

Combinations of the wing concepts that try to use the beneficial properties of several types are possible as well. One example is a kite airplane, shown in Fig. 2.3.

Figure 2.3: Combination of kite and airplane – example photograph of a kite airplane [6, p. 21, Fig. 2.1].

2.1.3 Dynamics of a tethered wing: the small earth

A wing looses one degree of freedom (DOF)

- if it is tethered,
- if the tether is always tense, which can be achieved by a controller, and
- if the tether is regarded to be ideal, i.e. massless, volume-less and infinitely stiff.

Hence, flights are only possible on the *wind window quarter sphere* or *small earth quarter sphere* or short *wind window* or *small earth*, shown in Fig. 2.4. This is a dedicated *area* in the sky in the shape of a quarter sphere opened to the wind velocity vector $\boldsymbol{v}_\mathrm{w} \in \mathbb{R}^3 \left[(\mathrm{m/s}, \mathrm{m/s}, \mathrm{m/s})^\top\right]$. The sphere's radius equals the length of the tether. Consequently, the position of the wing (and also its velocity, acceleration and forces) can be described with respect to a spherical coordinate system with the elevation angle $\beta_\mathrm{w} \in \mathbb{R}\,[^\circ]$ and the azimuth angle $\gamma_\mathrm{w} \in \mathbb{R}\,[^\circ]$. The index "w" stands for "wind" because the coordinate system is aligned to $\boldsymbol{v}_\mathrm{w}$. This consideration is also valid for non constant tether lengths, i.e. for a variable small earth radius. To steer the wing on a designed flight path on the small earth sphere, mainly its yaw orientation is controlled through the wing's rudders or steering lines. Through the loss of one DOF and the consideration in the spherical coordinate system, the wing can be controlled like an ordinary airplane or car (with approximately constant speed) that is controlled on a designed path in a two dimensional cartesian coordinate system, i.e. β_w and γ_w can be treated like the components of such a cartesian coordinate system. The term "small earth" results from this similarity that the tethered wing is flown on the small earth like an airplane on the *real* earth [7].

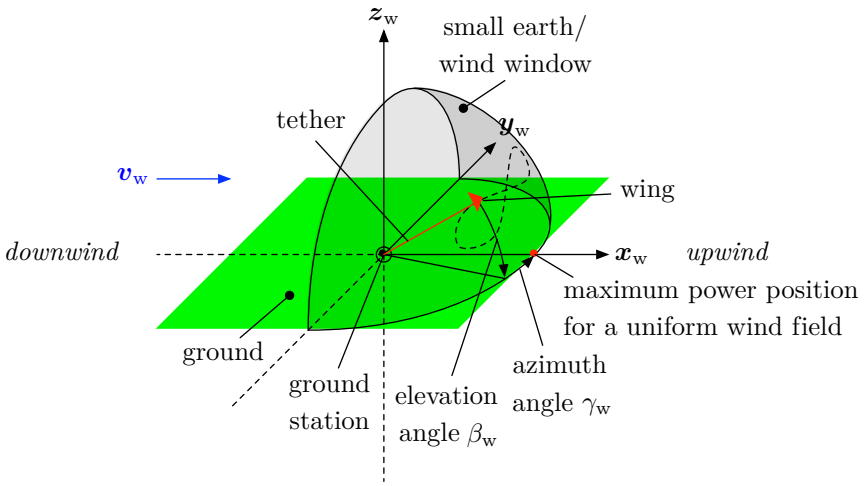


Figure 2.4: Small earth/wind window in the wind cartesian coordinate system with the axis unit vectors $\boldsymbol{x}_\mathrm{w}, \boldsymbol{y}_\mathrm{w}, \boldsymbol{z}_\mathrm{w} \in \mathbb{R}^3 \left[(1,1,1)^\top\right]$ in which $\boldsymbol{x}_\mathrm{w}$ is parallel to the wind velocity vector $\boldsymbol{v}_\mathrm{w} \in \mathbb{R}^3 \left[(\mathrm{m/s}, \mathrm{m/s}, \mathrm{m/s})^\top\right]$; the position of the wing can be described with respect to a spherical coordinate system with the elevation angle $\beta_\mathrm{w} \in \mathbb{R}\,[°]$ and the azimuth angle $\gamma_\mathrm{w} \in \mathbb{R}\,[°]$ [5, p. 6][6, p. 18, Fig. 2.7].

2.1.4 Basic aerodynamics

The total aerodynamic force $\boldsymbol{F}_\mathrm{ae} \in \mathbb{R}^3 \left[(\mathrm{N}, \mathrm{N}, \mathrm{N})^\top\right]$ acting on an arbitrary body can be split into two portions, drag force $\boldsymbol{F}_\mathrm{ae,d}$ and lift force $\boldsymbol{F}_\mathrm{ae,l}$, i.e.

$$\boldsymbol{F}_\mathrm{ae} = \boldsymbol{F}_\mathrm{ae,d} + \boldsymbol{F}_\mathrm{ae,l}.$$

The portions are defined by the flow of air with the apparent or relative wind velocity $\boldsymbol{v}_\mathrm{r} \in \mathbb{R}^3 \left[(\mathrm{m/s}, \mathrm{m/s}, \mathrm{m/s})^\top\right]$, i.e. the actual air velocity the body is exposed to: $\boldsymbol{F}_\mathrm{ae,d}$ is parallel to $\boldsymbol{v}_\mathrm{r}$ and $\boldsymbol{F}_\mathrm{ae,l}$ is perpendicular to $\boldsymbol{v}_\mathrm{r}$, see Fig. 2.5 (left). The common formulation for these forces is

$$\boldsymbol{F}_\mathrm{ae,d} = \frac{1}{2}\rho|\boldsymbol{v}_\mathrm{r}|^2 \, A \, c_\mathrm{d}(\alpha) \, \mathrm{dir}(\boldsymbol{v}_\mathrm{r}) \tag{2.1}$$

$$\boldsymbol{F}_\mathrm{ae,l} = \underbrace{\frac{1}{2}\rho|\boldsymbol{v}_\mathrm{r}|^2 \, A \, c_\mathrm{l}(\alpha)}_{\text{magnitude}} \underbrace{\mathrm{dir}\left(\boldsymbol{v}_\mathrm{r} \times \boldsymbol{y}_\mathrm{b}\right)}_{\text{direction}}, \tag{2.2}$$

where $\rho \in \mathbb{R}_{>0} \left[\mathrm{N/m^2}\right]$ is the air density, $A \in \mathbb{R}_{>0} \left[\mathrm{m^2}\right]$ is the projected area of the wing, $\alpha \in \mathbb{R}\,[°]$ is the angle of attack, $c_\mathrm{d}(\alpha) \in \mathbb{R}_{>0}\,[1]$ is the drag coefficient and $c_\mathrm{l}(\alpha) \in \mathbb{R}\,[1]$ is the lift coefficient. The last term of each equation describes the direction where $\boldsymbol{y}_\mathrm{b} \in \mathbb{R}^3 \left[(1,1,1)^\top\right]$ is the y coordinate axis unit vector of the body fixed cartesian coordinate system, see Fig. 2.5 (right), and

$$\mathrm{dir}(\boldsymbol{n}) := \frac{\boldsymbol{n}}{|\boldsymbol{n}|}$$

is the direction or unit vector of an arbitrary vector $\boldsymbol{n} \in \mathbb{R}^m, m \in \mathbb{N}$. The angle of attack may be expressed via the scalar product of $\boldsymbol{v}_\mathrm{r}$ and the z axis unit vector of the body fixed coordinate

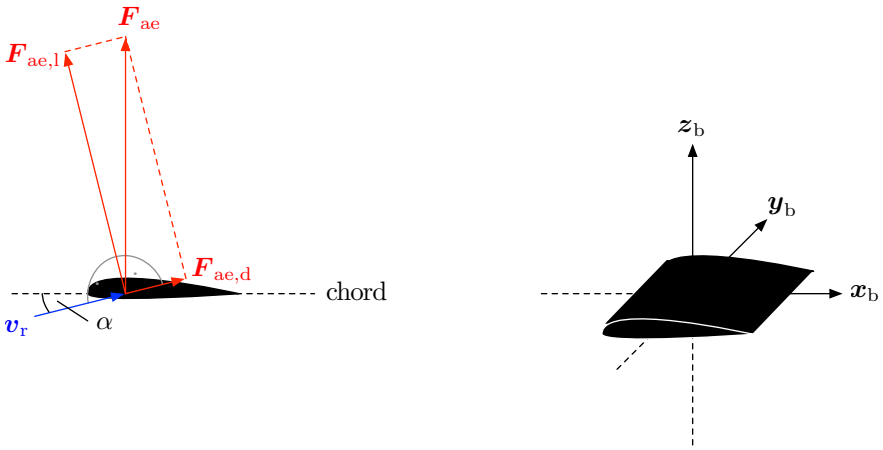


Figure 2.5: Relative wind velocity $\boldsymbol{v}_\mathrm{r} \in \mathbb{R}^3\left[(\mathrm{m/s}, \mathrm{m/s}, \mathrm{m/s})^\top\right]$ with angle of attack $\alpha \in \mathbb{R}\,[^\circ]$ produces the total aerodynamic force $\boldsymbol{F}_\mathrm{ae} \in \mathbb{R}^3\left[(\mathrm{N}, \mathrm{N}, \mathrm{N})^\top\right]$ which can be split into drag force $\boldsymbol{F}_\mathrm{ae,d}$ and lift force $\boldsymbol{F}_\mathrm{ae,l}$ with $\boldsymbol{F}_\mathrm{ae,d} \parallel \boldsymbol{v}_\mathrm{r}$ and $\boldsymbol{F}_\mathrm{ae,l} \perp \boldsymbol{v}_\mathrm{r}$ (left) and wing or body fixed cartesian coordinate system with perpendicular axis unit vectors $\boldsymbol{x}_\mathrm{b}, \boldsymbol{y}_\mathrm{b}, \boldsymbol{z}_\mathrm{b} \in \mathbb{R}^3\left[(1,1,1)^\top\right]$ where the wing spans over the $\boldsymbol{x}_\mathrm{b}\boldsymbol{y}_\mathrm{b}$ plane (right).

system, $\boldsymbol{z}_\mathrm{b}$,

$$\begin{aligned}
\boldsymbol{v}_\mathrm{r}^\top(-\boldsymbol{z}_\mathrm{b}) &= |\boldsymbol{v}_\mathrm{r}|\underbrace{|-\boldsymbol{z}_\mathrm{b}|}_{1}\cos(90^\circ-\alpha)\\
\Leftrightarrow \frac{\boldsymbol{v}_\mathrm{r}^\top}{|\boldsymbol{v}_\mathrm{r}|}(-\boldsymbol{z}_\mathrm{b}) &= \cos(90^\circ-\alpha)\\
\Leftrightarrow \arccos\left(\frac{\boldsymbol{v}_\mathrm{r}^\top}{|\boldsymbol{v}_\mathrm{r}|}(-\boldsymbol{z}_\mathrm{b})\right) &= 90^\circ-\alpha\\
\Leftrightarrow \alpha &= 90^\circ - \arccos\left(\frac{\boldsymbol{v}_\mathrm{r}^\top}{|\boldsymbol{v}_\mathrm{r}|}(-\boldsymbol{z}_\mathrm{b})\right). \text{ [1, 8, 9]}
\end{aligned}$$

2.2 Maximum power of AWE

Already in 1980 Miles Loyd derived the fundamental equations for both lift AWE and drag AWE over speed and force ratios [4]. Over the years the equations were refined. A very elegant derivation to calculate the maximum power of a tethered wing *in general*, is to use the maximum power which it extracts from the wind and subtract the inevitable losses [1, pp. 12]. This derivation is presented in the following.

2.2.1 Extracted power from the wind

First, the extracted power from the wind of a general (airborne) body is derived in this section. For this, consider a volume of air in rest in which a tethered (airborne) body is moved with the constant velocity $\boldsymbol{v}_\mathrm{b} \in \mathbb{R}^3\left[(\mathrm{m/s}, \mathrm{m/s}, \mathrm{m/s})^\top\right]$. Regardless of the body's shape, this will lead to a total aerodynamic force $\boldsymbol{F}_\mathrm{ae}$ acting on the body with a certain angle $\epsilon \in \mathbb{R}\,[^\circ]$ to $-\boldsymbol{v}_\mathrm{b}$. This situation is shown in Fig. 2.6 (left) where a tethered airborne is pulled horizontally to the left.

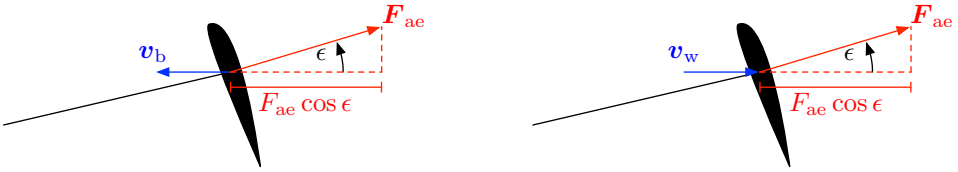


Figure 2.6: Thought experiment to derive the extracted power from the wind: A moving airborne body with the velocity $\boldsymbol{v}_{\mathrm{b}}$ in a still standing volume of air (left) and a still standing airborne body with a moving volume of air with the velocity $\boldsymbol{v}_{\mathrm{w}} = -\boldsymbol{v}_{\mathrm{b}}$ (right). In both cases a total aerodynamic force $\boldsymbol{F}_{\mathrm{ae}}$ with an angle ϵ acts on the body.

Considering that $\boldsymbol{F}_{\mathrm{ae}}$ compensates mass and that thus the airborne is in an equilibrium, the power $P \in \mathbb{R}\,[\mathrm{W}]$ needed to maintain $\boldsymbol{v}_{\mathrm{b}}$ is

$$P = v_{\mathrm{b}}\, F_{\mathrm{ae}} \cos(\epsilon).$$

Now, consider the situation in a coordinate system that stays at the airborne's position. So the volume of air streams by. This coordinate transformation is valid because both are inertial coordinate systems. This is the same situation when the airborne is tethered on the ground and the wind streams by with the constant wind velocity $\boldsymbol{v}_{\mathrm{w}} = -\boldsymbol{v}_{\mathrm{b}}$, as shown in Fig. 2.6 (right). So the power that the body extracts from the wind, $P_{\mathrm{w,e}} \in \mathbb{R}\,[\mathrm{W}]$, is

$$P_{\mathrm{w,e}} = v_{\mathrm{w}}\, F_{\mathrm{ae}} \cos(\epsilon).$$

The force that moves the air mass results in that case from pressure differences in the atmosphere. [1, pp. 12]

"Lemma: Power extraction formula
Regard a constant wind speed v_w. The total power [$P_{w,e}$] that a [body] extracts from this wind field is given by the product of v_w with the total aerodynamic force [F_{ae}] that the [body] experiences and the cosine of the angle [ϵ] between the direction of this force and the wind:

$$[P_{w,e} = v_w\, F_{ae} \cos(\epsilon)]\ ." \text{ [1, p. 12, Lemma 1.1]} \tag{2.3}$$

2.2.2 Power limit of airborne wind energy

The from the wind extracted power in Eq. (2.3) cannot be harnessed completely. To calculate the maximum usable power P, the inevitable losses P_{l} are subtracted from the extracted wind power $P_{\mathrm{w,e}}$, i.e.

$$\begin{aligned} P &= P_{\mathrm{w,e}} - P_{\mathrm{l}} \\ &= v_{\mathrm{w}}\, F_{\mathrm{ae}} \cos(\epsilon) - P_{\mathrm{l}}. \end{aligned}$$

The upper limit for the extracted wind power is reached for $\epsilon = 0$, i.e.

$$\begin{aligned} P &\leq v_{\mathrm{w}}\, F_{\mathrm{ae}} \cos(0) - P_{\mathrm{l}} \\ &\leq v_{\mathrm{w}}\, F_{\mathrm{ae}} - P_{\mathrm{l}}. \end{aligned} \tag{2.4}$$

The magnitude of the total aerodynamic force of a body results from the geometric sum of the magnitudes of the drag and lift forces from Eqs. (2.1) and (2.2),

$$\begin{aligned} F_{\mathrm{ae}} &= \sqrt{F_{\mathrm{ae,d}}^2 + F_{\mathrm{ae,l}}^2} \\ &= \sqrt{\left(\frac{1}{2}\rho v_{\mathrm{r}}^2 \, A \, c_{\mathrm{d}}\right)^2 + \left(\frac{1}{2}\rho v_{\mathrm{r}}^2 \, A \, c_{\mathrm{l}}\right)^2} \\ &= \frac{1}{2}\rho v_{\mathrm{r}}^2 \, A \sqrt{c_{\mathrm{d}}^2 + c_{\mathrm{l}}^2}, \end{aligned}$$

where the dependency of the aerodynamic coefficients from the angle of attack α is dropped for simplicity. For drag AWE, a turbine is added to the body which is ideally aligned with the relative wind velocity $\boldsymbol{v}_{\mathrm{r}}$. Therefore, the drag coefficient c_{d} may be split into an intrinsic portion $c_{\mathrm{d,i}}$, which is induced by the body alone, and into a portion induced by a possible turbine $c_{\mathrm{d,t}}$, which leads to

$$\begin{aligned} F_{\mathrm{ae}} &= \frac{1}{2}\rho v_{\mathrm{r}}^2 \, A \underbrace{\sqrt{(c_{\mathrm{d,i}} + c_{\mathrm{d,t}})^2 + c_{\mathrm{l}}^2}}_{=: c_{\mathrm{ae}}} \\ &= \frac{1}{2}\rho v_{\mathrm{r}}^2 \, A \, c_{\mathrm{ae}}. \end{aligned} \tag{2.5}$$

Inserting Eq. (2.5) into Eq. (2.4) gives

$$P \leq v_{\mathrm{w}} \, \frac{1}{2}\rho v_{\mathrm{r}}^2 \, A \, c_{\mathrm{ae}} - P_{\mathrm{l}}. \tag{2.6}$$

For a wing, the lower boundary for the power loss P_{l} is the intrinsic drag loss, i.e.

$$\begin{aligned} P_{\mathrm{l}} &\geq v_{\mathrm{r}} \, F_{\mathrm{ae,d,i}} \\ &\geq v_{\mathrm{r}} \, \frac{1}{2}\rho v_{\mathrm{r}}^2 \, A \, c_{\mathrm{d,i}} \\ &\geq \frac{1}{2}\rho v_{\mathrm{r}}^3 \, A \, c_{\mathrm{d,i}}. \end{aligned}$$

This lower boundary is inserted into Eq. (2.6),

$$\begin{aligned} P &\leq v_{\mathrm{w}} \, \frac{1}{2}\rho v_{\mathrm{r}}^2 \, A \, c_{\mathrm{ae}} - \frac{1}{2}\rho v_{\mathrm{r}}^3 \, A \, c_{\mathrm{d,i}} \\ &\leq \frac{1}{2}\rho v_{\mathrm{r}}^2 \, A \, \left(v_{\mathrm{w}} \, c_{\mathrm{ae}} - v_{\mathrm{r}} \, c_{\mathrm{d,i}}\right). \end{aligned} \tag{2.7}$$

By introducing the dimensionless wing speed ratio $\lambda \in \mathbb{R}$ [1],

$$\lambda := \frac{v_{\mathrm{r}}}{v_{\mathrm{w}}}, \tag{2.8}$$

and inserting $v_{\mathrm{r}} = \lambda v_{\mathrm{w}}$ into Eq. (2.7) gives

$$\begin{aligned} P &\leq \frac{1}{2}\rho \lambda^2 v_{\mathrm{w}}^2 \, A \, \left(v_{\mathrm{w}} \, c_{\mathrm{ae}} - \lambda v_{\mathrm{w}} \, c_{\mathrm{d,i}}\right) \\ &\leq \frac{1}{2}\rho v_{\mathrm{w}}^3 \, A \, \lambda^2 \left(c_{\mathrm{ae}} - \lambda \, c_{\mathrm{d,i}}\right). \end{aligned} \tag{2.9}$$

Obviously, the power becomes 0 if $\lambda = 0$ or if $\lambda = \frac{c_{\mathrm{ae}}}{c_{\mathrm{d,i}}}$, i.e. v_{r} ranges from 0 to $v_{\mathrm{r}} = \frac{c_{\mathrm{ae}}}{c_{\mathrm{d,i}}} v_{\mathrm{w}}$, if the wing is driven only by the wind v_{w}. Consequently, v_{r} is to be reduced to a certain value between 0 and $v_{\mathrm{r}} = \frac{c_{\mathrm{ae}}}{c_{\mathrm{d,i}}} v_{\mathrm{w}}$. For lift AWE this is achieved by releasing the tether, for drag AWE this is achieved by adding turbine drag. The maximum power is found by maximizing this equation over λ. For this, the right hand side of Eq. (2.9), is differentiated with respect to λ and set to zero,

$$\begin{aligned} 0 &= \frac{\mathrm{d}}{\mathrm{d}\lambda} \left(\frac{1}{2} \rho v_{\mathrm{w}}^3 \, A \, \lambda^2 (c_{\mathrm{ae}} - \lambda \, c_{\mathrm{d,i}}) \right) \\ &= \frac{\mathrm{d}}{\mathrm{d}\lambda} (\lambda^2 \, c_{\mathrm{ae}} - \lambda^3 \, c_{\mathrm{d,i}}) \\ &= 2\lambda \, c_{\mathrm{ae}} - 3\lambda^2 \, c_{\mathrm{d,i}} \\ &= \lambda (2 \, c_{\mathrm{ae}} - 3\lambda \, c_{\mathrm{d,i}}). \end{aligned}$$

As evaluated above the first result $\lambda = 0$ leads to $P = 0$ and can be dropped. The second result with $\lambda \neq 0$ is

$$\begin{aligned} 0 &= \lambda^* (2 \, c_{\mathrm{ae}} - 3\lambda^* \, c_{\mathrm{d,i}}) \qquad | \cdot \lambda^{*-1} \\ &= 2 \, c_{\mathrm{ae}} - 3\lambda^* \, c_{\mathrm{d,i}} \\ 3\lambda^* \, c_{\mathrm{d,i}} &= 2 \, c_{\mathrm{ae}} \\ \lambda^* &= \frac{2}{3} \, \frac{c_{\mathrm{ae}}}{c_{\mathrm{d,i}}}. \end{aligned} \tag{2.10}$$

Consequently, v_{r} is to be reduced to $\frac{2}{3}$ of the value it would have if it was driven from the wind alone. The result of Eq. (2.10) inserted into Eq. (2.9) leads to the maximum power of

$$\begin{aligned} P &\leq \frac{1}{2} \rho v_{\mathrm{w}}^3 \, A \, \lambda^{*2} \, (c_{\mathrm{ae}} - \lambda^* \, c_{\mathrm{d,i}}) \\ &\leq \frac{1}{2} \rho v_{\mathrm{w}}^3 \, A \, \left(\frac{2}{3} \, \frac{c_{\mathrm{ae}}}{c_{\mathrm{d,i}}} \right)^2 \left(c_{\mathrm{ae}} - \frac{2}{3} \, \frac{c_{\mathrm{ae}}}{c_{\mathrm{d,i}}} \, c_{\mathrm{d,i}} \right) \\ &\leq \frac{2}{27} \rho v_{\mathrm{w}}^3 \, A \, \frac{c_{\mathrm{ae}}^3}{c_{\mathrm{d,i}}^2}. \text{ [1, pp. 12]} \end{aligned}$$

"Theorem: Power limit of airborne wind energy
[...] Regard a wing with area A and aerodynamic [intrinsic drag and lift] coefficients [$c_{d,i}$] and [c_l] that is moved in the wind of [...] wind speed v_w and air density ρ. When the wing's motion [...] is not only influenced by its intrinsic lift and drag, but also by additional drag forces, such as an on-board turbine with corresponding drag coefficient [$c_{d,t}$] and by non-aerodynamic forces, such as a tether, then the total usable power P that can be harvest from the wind using these extra forces is limited by

$$\left[P \leq \frac{2}{27} \rho v_w^3 \, A \, \frac{c_{ae}^3}{c_{d,i}^2} \right] \tag{2.11}$$

with

$$\left[c_{ae} = \sqrt{(c_{d,i} + c_{d,t})^2 + c_l^2} \right]. \tag{2.12}$$

This limit can be achieved if the total aerodynamic force is in line with the wind direction, if the wing's drag is the only loss and if the [relative wind speed] of the wing is made equal to

$$\left[v_r = \frac{2}{3} \frac{c_{ae}}{c_{d,i}} v_w \right] \text{". [1, p. 17, Theorem 1.1]} \tag{2.13}$$

2.2.3 Optimal tether reel out speed in pure lift AWE and optimal turbine drag in pure drag AWE

For a uniform wind field, it can be shown, that the position where a tethered wing can achieve the fundamental power limit, is in the pure down wind position [1, pp. 33], marked in Fig. 2.4 on p. 17. The situation of the velocities and forces of a massless wing in an acceleration less state in this maximum power position is shown in Fig. 2.7. Of course, in reality the wind field is

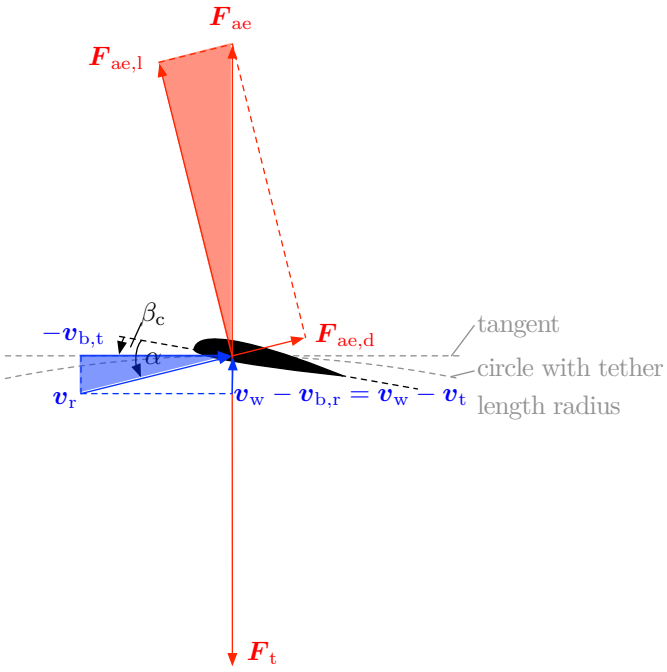


Figure 2.7: Velocities, forces, angle of attack α and pitch angle β_{c} for a massless and unaccelerated wing which flies exactly perpendicular to the wind. Here, $\boldsymbol{v}_{\mathrm{r}}$ is the relative wind velocity, $\boldsymbol{v}_{\mathrm{b,t}}$ is the tangential portion of the body velocity, $\boldsymbol{v}_{\mathrm{b,r}}$ is the radial portion of the body velocity which is equal to the tether velocity $\boldsymbol{v}_{\mathrm{t}}$, $\boldsymbol{F}_{\mathrm{ae}}$ is the total aerodynamic force, $\boldsymbol{F}_{\mathrm{ae,d}}$ is the drag force and $\boldsymbol{F}_{\mathrm{ae,l}}$ is the lift force.

not uniform but sheered and the wind speed of that position is 0, so the real maximum power position will be at a higher elevation angle. The highlighted triangles in Fig. 2.7 are similar [4]

and thus the ratio

$$\begin{aligned}\frac{v_\mathrm{r}}{v_\mathrm{w}-v_\mathrm{t}} &= \frac{F_\mathrm{ae}}{F_\mathrm{ae,d}} \\ &= \frac{\frac{1}{2}\rho v_\mathrm{r}^2\, A\, c_\mathrm{ae}}{\frac{1}{2}\rho v_\mathrm{r}^2\, A\, c_\mathrm{d}} \\ &= \frac{c_\mathrm{ae}}{c_\mathrm{d}}\end{aligned} \tag{2.14}$$

can be established.

For pure lift AWE, i.e. $c_\mathrm{d} = c_\mathrm{d,i}$, Eq. (2.14) can be inserted into Eq. (2.10) which leads, with the definition of λ in Eq. (2.8), to the optimal tether speed of

$$\begin{aligned}\lambda^* &= \frac{2}{3}\,\frac{c_\mathrm{ae}}{c_\mathrm{d,i}} \\ \frac{v_\mathrm{r}}{v_\mathrm{w}} &= \frac{2}{3}\,\frac{v_\mathrm{r}}{v_\mathrm{w}-v_\mathrm{t}} \\ v_\mathrm{w} &= \frac{3}{2}\,(v_\mathrm{w}-v_\mathrm{t}) = \frac{3}{2}v_\mathrm{w} - \frac{3}{2}v_\mathrm{t} \\ \frac{3}{2}v_\mathrm{t} &= \frac{3}{2}v_\mathrm{w} - v_\mathrm{w} = \frac{1}{2}v_\mathrm{w} \\ v_\mathrm{t} &= \frac{1}{3}v_\mathrm{w}.\end{aligned} \tag{2.15}$$

For pure drag AWE, the tether length stays constant, i.e. $v_\mathrm{t} = 0$, and thus the ratio in Eq. (2.14) becomes

$$\frac{c_\mathrm{ae}}{c_\mathrm{d}} = \frac{v_\mathrm{r}}{v_\mathrm{w}} \quad (=\lambda)$$

which is equal to λ. With the portions of the drag coefficient, $c_\mathrm{d} = c_\mathrm{d,i} + c_\mathrm{d,t}$, and the optimal λ^* as in Eq. (2.10), the optimal turbine drag coefficient is given by

$$\begin{aligned}\lambda^* &= \frac{2}{3}\,\frac{c_\mathrm{ae}}{c_\mathrm{d,i}} \\ \frac{c_\mathrm{ae}}{c_\mathrm{d}} &= \frac{2}{3}\,\frac{c_\mathrm{ae}}{c_\mathrm{d,i}} \\ \frac{1}{c_\mathrm{d,i}+c_\mathrm{d,t}} &= \frac{2}{3}\,\frac{1}{c_\mathrm{d,i}} \\ c_\mathrm{d,i}+c_\mathrm{d,t} &= \frac{3}{2}c_\mathrm{d,i} \\ c_\mathrm{d,t} &= \frac{1}{2}c_\mathrm{d,i}.\end{aligned} \tag{2.16}$$

2.2.4 Major losses

There are several losses that lower the actual power below that of Eq. (2.11). The major losses to be mentioned are listed below:

- For lift AWE, only in the reel-out phase power is produced. In the reel-in phase, power is consumed. This lowers the average cycle power below the derived limit.

- The tether adds additional not negligible drag [1, pp. 12]. This drag may be modeled by a raise of the body's drag coefficient and thus lowers the maximum (intrinsic) power. This is in particular critical for drag AWE, since the whole power has to be transmitted over the tether.
- The body can stay in the maximum power position only for one point in time. If the body is flown in circles around the maximum power position, than this position is never reached. Additionally, since the steering results from a geometric deformation, usually steering drag is added to the wing. So there are further losses because circles or figure eights need to be flown. [1]
- The exact wind direction and wind speed and thus the exact maximum power position may be unknown in the altitude of the wing. Additionally, the tether cannot be aligned exactly with the wind vector because the tether exit point is on the ground.
- Further losses occur in the electrical machines, gears, etc. Power to steer the electrical machines has to be provided.

2.2.5 The power harvesting factor (i.e. the Betz factor for AWE)

The *imagined* power of the wind P_w through an area that equals the projected wing area A is given by

$$P_\mathrm{w} = \frac{1}{2}\rho v_\mathrm{w}^3 \, A.$$

The power harvesting factor ζ is defined as the ratio of the maximum AWE power of Eq. (2.11) and that imagined wind power through the wing area A,

$$\zeta := \frac{P}{P_\mathrm{w}} = \frac{\frac{2}{27}\rho v_\mathrm{w}^3 \, A \, \frac{c_\mathrm{ae}^3}{c_\mathrm{d,i}^2}}{\frac{1}{2}\rho v_\mathrm{w}^3 \, A} = \frac{4}{27}\frac{c_\mathrm{ae}^3}{c_\mathrm{d,i}^2}. \tag{2.17}$$

This factor has the same meaning as the Betz factor for classical wind energy, since it is a definition for the maximum reachable efficiency of a wing. [1, p. 18]

2.2.6 Experimentally achieved power harvesting factors and power densities

For rigid airplanes realistic values for the lift and drag coefficients at the optimal angle of attack α^* are $c_\mathrm{l}^* = 1.0$ [1, p. 18] and $c_\mathrm{d}^* = 0.07$ [1, p. 18]. Inserted into Eq. (2.17), this results in a maximum theoretical power harvesting factor of $\hat{\zeta} \approx 30$. At a moderate wind speed of $v_\mathrm{w} = 10\,\mathrm{m/s}$ and an air density of $\rho = 1.2\,\mathrm{kg/m^3}$, this leads with Eq. (2.11) to a maximum reachable power of $\hat{P}/A \approx 18.2\,\mathrm{kW/m^2}$.

The company Makani reported the best experimentally realized power harvesting factor of $\zeta = 8$ [1, p. 18] so far, which is $\approx 0.25\hat{\zeta}$. The maximum reachable power at the mentioned wind conditions with this power harvesting factor is $P/A \approx 4.8\,\mathrm{kW/m^2}$.

For kites usually the drag coefficient is worse and is realistically about $c_\mathrm{d}^* = 0.2$ [1, p. 18] which leads to a maximum power harvesting factor of $\hat{\zeta} \approx 4$ [1, p. 18] and power density with the same wind conditions of $P/A \approx 2.4\,\mathrm{kW/m^2}$.

Considering

- the size of the currently hugest SkySails kite of $A = 320\,\mathrm{m^2}$ [10],

- the mentioned lift and drag coefficients, $c_\mathrm{l}^* = 1.0$ and $c_\mathrm{d}^* = 0.2$, and
- a higher wind speed of $v_\mathrm{w} = 15\,\mathrm{m/s}$, which is realistic in higher altitudes,

the maximum power of that kite with Eq. (2.11) is $P \approx 2.5\,\mathrm{MW}$. Considering losses the claimed maximum power equivalent of 2 MW [2] by SkySails seems feasible.

2.3 Automation of a crosswind AWE plant

2.3.1 General requirements

Crosswind AWE only has a chance in the electrical power generation industry, if it is automated completely, i.e. a crosswind AWE plant must perform the following tasks reliably and autonomously:

1. start the wing when the wind is strong enough,
2. produce power by flying circles or figure eights (and perform pumping cycles for lift AWE), and
3. land the wing when the wind speed is too low or the weather becomes too extreme.

All this has to be done while holding certain variables like the tether force or the distance between the wing and the ground within certain boundaries to *guarantee* a save operation.

2.3.2 Control Strategies

There are several different control strategies to fly the above mentioned maneuvers, see e.g. [5]. With the considerations of the small earth in Sec. 2.1.3 on pp. 16, maybe the most obvious control strategy is to use a nested controller structure [5, 11], as shown in Fig. 2.8:

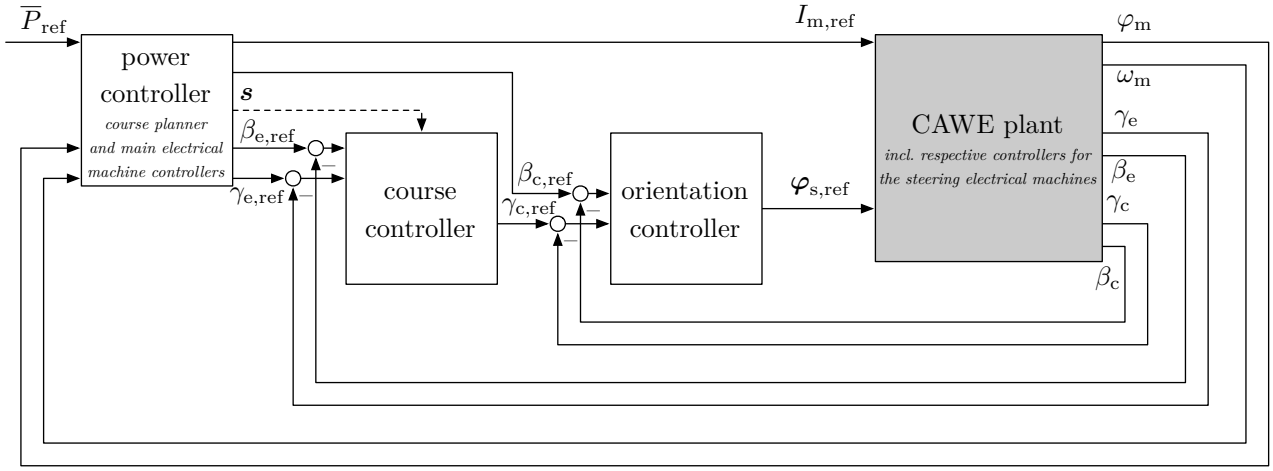


Figure 2.8: Control strategy with nested controllers.

1. The innermost controller is an orientation controller, i.e. it controls the wing's yaw angle $\gamma_\mathrm{c} \in \mathbb{R}\,[°]$, which is the angle of the wing around the tether. The index "c" stands for control. Additionally, this controller may also control the wing's pitch angle $\beta_\mathrm{c} \in \mathbb{R}\,[°]$.

The outputs in both cases are the reference positions for the steering electrical machines $\boldsymbol{\varphi}_{\mathrm{s,ref}} \in \mathbb{R}^n \left[(^\circ, ^\circ, \dots)^\top\right], n \in \mathbb{N}_{>0}$.

2. The course controller brings the wing on the desired course, that is e.g. a circle path, a figure eight path or the zenith position. The circle path may be defined by its center position and the figure eight path may be defined by its position of intersection. Thus, it would be sufficient if the controller receives a position in the two coordinates of the wind window sphere, i.e. the elevation angle $\beta_{\mathrm{w,ref}}$ and the azimuth angle $\gamma_{\mathrm{w,ref}}$, and one or more shape parameters, in Fig. 2.8 denoted as $\boldsymbol{s}$. In order to lose the dependency of the wind direction, the elevation and azimuth angles in the earth fixed reference frame, $\beta_{\mathrm{e,ref}}$ and $\gamma_{\mathrm{e,ref}}$, may be chosen. The controller's output, i.e. its control variable, is only the wings's yaw angle, γ_{ref}, which is set so, that the wing heads towards the desired path or position.

3. Finally, the outermost controller is the power controller which has two tasks: Firstly, it plans the course and thus the reel out and reel in phases for lift AWE, i.e. it outputs $\beta_{\mathrm{e,ref}}$ and $\gamma_{\mathrm{e,ref}}$ as well as $\boldsymbol{s}$ to the course controller. It may also set the pitch angle of the wing $\beta_{\mathrm{c,ref}}$ (powering/depowering). Secondly, it controls the main electrical machine with the reference current $I_{\mathrm{m,ref}} \in \mathbb{R}\,[\mathrm{A}]$ which is linked directly to its moment and thus to the tether force. The main electrical machine's angular speed $\omega_{\mathrm{m}} \in \mathbb{R}\,[^\circ/\mathrm{s}]$ is to be measured in order to control it. To detect when the phases from reel in to reel out should be changed in lift AWE, also the main electrical machine's position angle $\varphi_{\mathrm{m}} \in \mathbb{R}\,[^\circ]$ is to be measured which is linked directly to the tether length. The reference value of this controller is e.g. the average output power $\overline{P}_{\mathrm{ref}}$.

So, with the introduction of the small earth coordinates, standard approaches from aviation can be used [5, p. 28, p. 50]. The advantage from the small earth approach is, that the control problem is divided into several easier control problems.

Another approach, that is investigated thoroughly, is to use a nonlinear model predictive controller to control the entire system in all phases. [1]

The biggest automation challenges for both control engineering and mechanical engineering are the start and landing phases. Many researchers and companies may have a prototype with fully automated pumping cycle, but no automatic start or landing. SkySails performs the start and landing phases with a telescoping boom. Others investigate a rotatory start and landing to artificially generate enough relative wind speed on the ground. For drag AWE, in contrast to lift AWE, start and landing seems to be easier, since the flying electrical machines with the turbines can be used in motor mode and therefore the wing can start and land like a helicopter. [1]

2.3.3 Sensors and actuators

Fig. 2.8 shows the most important signals to be measured. To measure the position of the wing angle sensors at least for the main tether or an inertial measurement unit (IMU) are necessary. The most important sensors already come with the encoders and current sensors of the electrical machines. With an IMU, depending on the product, also the velocity, orientation and orientation velocity may be available. However, an IMU must be mounted in or at the wing, needs electricity and the data needs to be transmitted to the ground. This results in the need to produce some power onboard or to transmit some power over the tether to the wing. The latter would exclude the use of standard tethers as used for kite surfing or paragliding. However, in future there could be the need for some electricity on the wing for safety reasons anyway, e.g. to illuminate the wing or to power collision avoidance electronics to protect civil aircrafts.

The most important actuators are the main and steering electrical machines. Additionally, guides for the tethers and a mechanism that enables the wing to reach all azimuth angles $\gamma_{\mathrm{e}} \in$

$[0, 360\,°)$ are crucial. Depending on the specific concept, there may be some more actuators like breaks for the electrical machines or a mechanism that separates the wing from the tether.

3 Proposed PLC approach for a lift airborne wind energy research prototype

A key component of a lift AWE research prototype is the PLC of the automation system. In the first section of this chapter, the terminology of "PLC" is classified. The proposed PLC approach for the research prototype is presented in the second section. Regardless of that approach, the general requirements of a PLC for this application are then specified and compared directly with the proposed approach. The last section outlines the solutions of AWE research groups, companies and available PLCs, to which the proposed approach is also compared to.

3.1 Scope

A commonly used picture for automation is the automation pyramid shown in Fig. 3.1. The

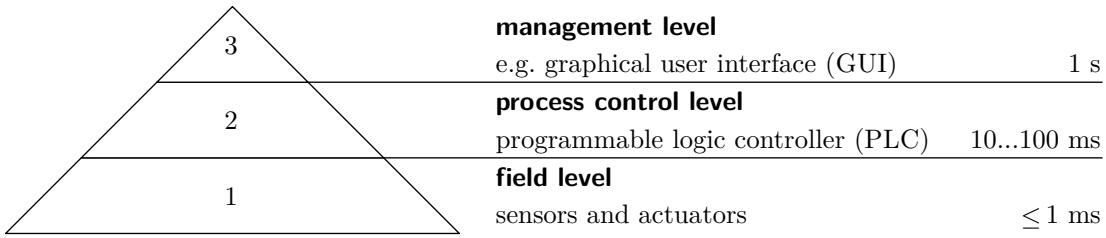


Figure 3.1: Automation pyramid with rough control cycle period times for a lift AWE plant. [12, pp. 202]

pyramid has three levels: The field level is the lowest level which contains the sensors and actuators. In many cases they have specialized micro controllers for low level data processing and control. One example for this is the power electronics controller of an electrical machine. Above that level is the process control level with a PLC. A PLC consists of one or more computers with digital, analog or bus interface possibilities e.g. by specialized interface cards to which the sensors and actuators are connected. On these computers the higher level controllers are executed, e.g. the three nested controllers from Fig. 2.8 on p. 25. Finally, the highest level is the management level which may be implemented with a GUI or a high level communication interface, e.g. Internet. Here, the states of the system are visualized and higher level control goals are set and transmitted to the PLC, e.g. the reference average power of the lift AWE plant. [12, pp. 202]

This thesis focuses on the upper two layers where with the management level a GUI is meant. Throughout this thesis the term PLC is used for control system software and hardware with input and output possibilities to interface sensors and actuators. For the software development, the term also includes an IDE and at least an oscilloscope GUI to display system states and signal trends for debugging purposes. Here, the term does not restrict the choice of programming language to the classical PLC programming languages from IEC 61131.

3.2 Outline of the proposed PLC approach

For the lift AWE prototype the in Fig. 3.2 visualized PLC approach is proposed. The whole

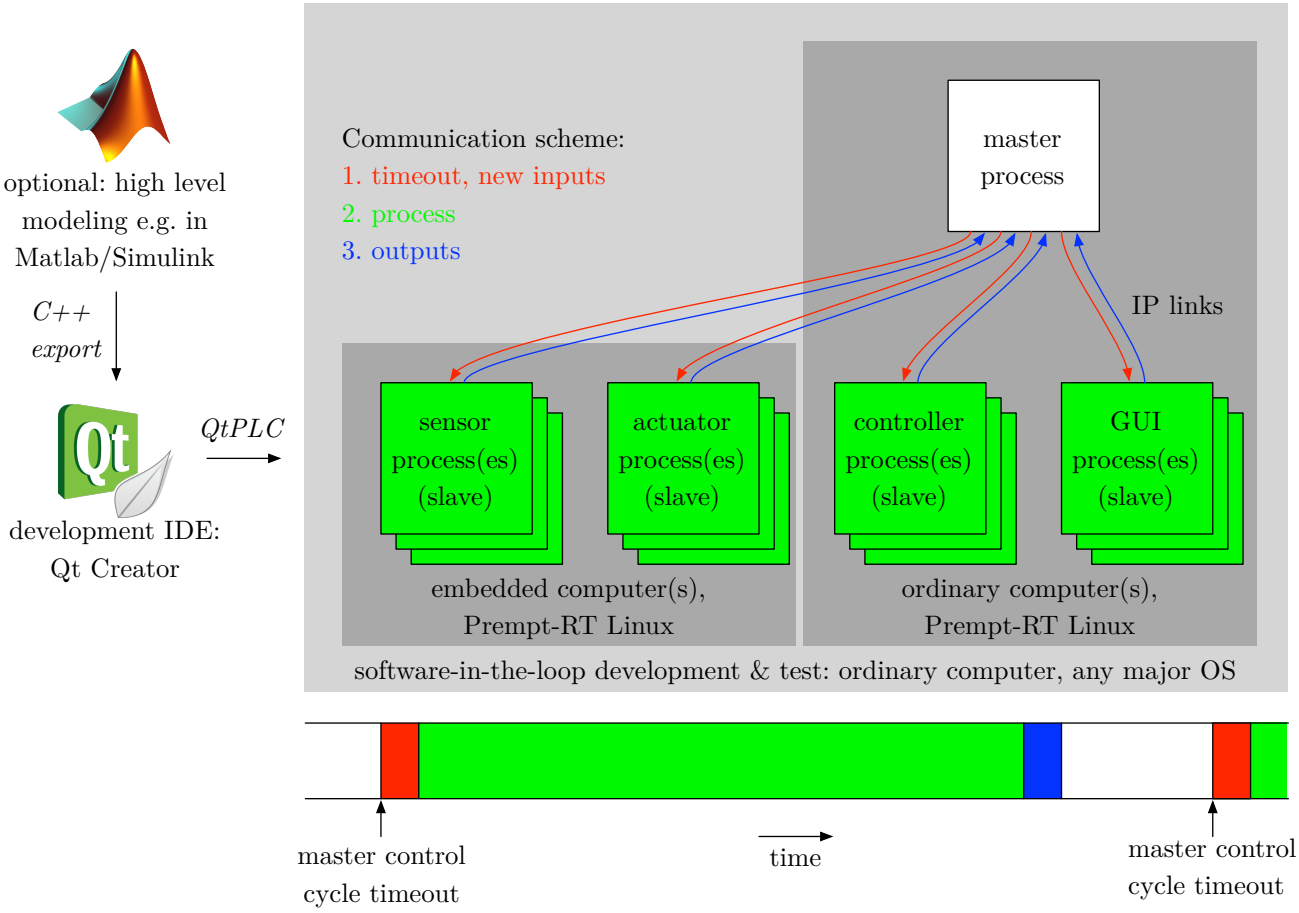


Figure 3.2: Overview visualization of the proposed PLC approach.

approach is designed with the KISS principle (keep it simple and stupid, i.e simplicity is a key design goal and unnecessary complexity is avoided) and focus is paid to low cost in general. A detailed explanation is listed below:

Hardware

For high computationally intensive tasks ordinary computers are used. Small embedded computers are used where the computational load is low, low level interfaces like CAN bus[6] are needed or the computer is needed in a remote place, e.g. inside the control pod. As embedded computers inexpensive and preferably open source ones like the BeagleBone Black or the Raspberry Pi, shown in Fig. 3.3, are selected. This holds the hardware costs at a minimum.

Operating system

For each computer, a Preempt-RT patched real time Linux is used to gain *hard* real time.

[6] Controller Area Network, serial bus.

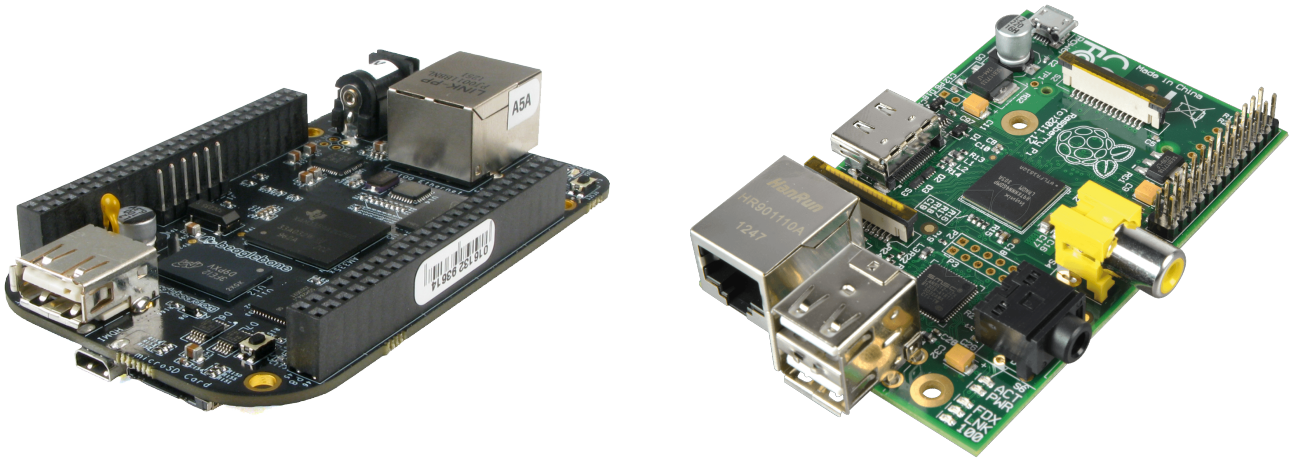


Figure 3.3: Photographs of the BeagleBone Black (left)[7]and the Raspberry Pi (right)[8].

With this patch, *any* application can be executed with real time priority only with a special command. Thus, any programming language and framework can be used. All advantages from a modern Linux operating system are available, e.g. graphical user interface (if needed), remote access via SSH[9] and VNC[10], compiler tool chains including remote debugging e.g. with GDB[11] server, frameworks etc. The PLC software can be directly developed on the target hardware or remotely. Since Linux runs on almost any computer/processor the above mentioned embedded computers can be used. (Only micro controllers usually do not run Linux due to their low computational power.)

Framework

The PLC software is developed in C++ using the open source and cross platform Qt framework which has APIs for all major tasks and comes with the cross platform Qt Creator IDE. To ease the PLC software development and to encapsulate repeating tasks, in particular the communication of the distributed control system, a C++ PLC library based on Qt is developed in this thesis and therefore named "QtPLC". The application specific control software may be developed directly with object oriented C++ or with major control design and simulation software packages, e.g. Matlab/Simulink, via their export functions to C/C++. Special focus was paid to an abstract, clean and intuitive API of QtPLC, to make the development of the PLC software as simple as possible. Since Qt is cross platform, the developed PLC software runs on almost *any* operating system and thus almost any computer (even on smart phones).

Communication

The computers communicate with each other via IP links, i.e. with the standard User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) provided by Qt classes via ethernet or Wireless Local Area Network (WLAN). WLAN is only used if necessary for

[7]Image source: https://www.logicsupply.com/blog/wp-content/uploads/2013/05/beaglebone-black-board-logicsupply_pic1.jpg, accessed: December 03, 2013.
[8]Image source: http://cdn-reichelt.de/bilder/web/xxl_ws/A300/RASPBERRY_PI_B_01.png, accessed: December 03, 2013.
[9]Secure Shell, software to access a computer remotely over a network via a command line interface.
[10]Virtual Network Computing, software to access a computer remotely over a network via a graphical user interface.
[11]GNU Debugger, debugger software.

speed and reliability reasons, e.g. for the communication from the ground station to the wing.

The distributed control system is build up modular with different processes for controllers, actuators, sensors, graphical user interfaces (GUI) etc. It is arbitrary on which computer how many and which processes run, because an IP link between processes can also be established on a single computer. This gives the advantage that the whole PLC software can be developed and tested fully software-in-the-loop on a single, ordinary computer that runs an operating system that is supported by Qt.

As a starting point, a very simplistic (KISS principle) communication scheme is used that does not require time synchronization: Each process is seen as a black box with input and output signals. A sensor may also have input signals e.g. for a sensor fusion process and an electrical machine actuator should provide sensor signals e.g. for current, angular speed and angular position. All signals are defined globally. The master collects all signal values and any process can access any signal as input and define any signal as output. For that, each process registers itself only to the master process and informs the master about its desired input and output signals. Only the master runs a control cycle timer. The input-process-output scheme is visualized in Fig. 3.2 in red-green-blue: On each timeout, the master sends to all the slave processes a timeout command. The required input signals from the last control cycle are attached to this command for the corresponding slave. The processes then calculate output values and immediately send them back to the master. So with this scheme, only two messages per control cycle are sent between each process and the master. New processes can join the distributed PLC by connecting to the master. However, the disadvantage is that for each signal link, e.g. from a sensor to a controller, a communication dead time of the master control cycle period time is introduced, but is negligible if the control cycle time is small enough. Additionally, for the slaves' timeout an offset of the needed time to send the timeout message from the master to the respective slave is introduced, but is negligible as well, if the sending of messages is fast enough. For that, the communication protocol was designed with minimum overhead.

User interaction

The QtPLC library provides the "QtPLC Control Center" graphical user interface API for both to visualize data and states of the system and to interact with the PLC e.g. by setting certain signal values. The API of the QtPLC Control Center is designed such that it is easily extensible e.g. with a 3D visualization which is very beneficial for AWE. It acts as an ordinary process for the PLC. Via an IP link over the Internet, the QtPLC Control Center may also be run on a remote place.

It should be pointed out that the use of a Preempt-RT Linux leads to the advantage that no specialized API for the real time ability is needed. The scheduler of Preempt-RT is designed such that (almost) any process, i.e. (almost) any code no matter if it is executed in kernel or user space, is preemptive [13]. Therefore, also any user space process can be set to the highest priority, e.g. with the *chrt* command [14]. Hence, Qt alone as underlying framework can be used for the PLC and all the mentioned advantages emerge. This is not possible with the other real time Linux patches like *RTAI*[12] or *Xenomai*[13].

Note that although it is recommended to use C/C++ on a Preempt-RT Linux [13] the PLC software processes are not limited to be developed with the QtPLC C++ library because any process can be started with real time priority in Preempt-RT. However, all parts of the commu-

[12] "RTAI – Real Time Application Interface for Linux". https://www.rtai.org, accessed: November 18, 2013.
[13] "Xenomai: Real-Time Framework for Linux". http://www.xenomai.org, accessed: November 18, 2013.

nication scheme are implemented in QtPLC and would need to be reimplemented in the used programming language or framework, respectively.

3.3 Requirements of a PLC for a lift AWE prototype

3.3.1 Development of the specifications and comparison to the proposed approach

Regardless of the idea of the proposed approach, in the following list the requirements of a PLC for a lift AWE research prototype are specified, weighted from "important" to "must" and explained. Below each specification, the general fulfillment of the proposed approach is discussed and visualized with a background in a traffic light color (red/yellow/green) or in gray if a further discussion is needed.

A) *Simplicity (must)*
The PLC must be simple to program and to configure, i.e. an IDE with important development features like auto completion and debugging is crucial. It must be able to import models from major controller simulation tools, e.g. Matlab/Simulink. Additionally, the PLC must be simple to operate, i.e. variables should be plotted in real time and interaction during runtime through a user interface must be possible. This should also be possible from a remote place e.g. over a network or Internet.

Fulfillment in the proposed approach: If Qt Creator as IDE, if the QtPLC library and if its QtPLC Control Center as GUI are used, this requirement is fulfilled.

B) *Real time ability (must)*
A PLC *must* be real time capable.

Fulfillment in the proposed approach: Since the use of a Preempt-RT patched Linux is proposed, the processes run with real time priority. However, with the use of Qt alone only TCP/IP and UDP/IP sockets are provided for which indeed no real time priority can be set, i.e. no real time ethernet can be used with Qt alone.

A deeper discussion on real time, how critical this point actually is and if this specification is fulfilled, is given in the next section.

C) *Software-in-the-loop ability (must)*
For rapid development, it must be possible to develop and test the PLC software completely with models, i.e. software-in-the-loop, before it is tested on a real system. Preferably, this should be possible on an ordinary computer.

Fulfillment in the proposed approach: The whole PLC software can be developed fully software-in-the-loop on a single ordinary computer that runs a Qt supported operating system. So this requirement is fulfilled.

D) *Interoperability (must)*
PLC software parts may later be needed for other specialized simulation software. An example from classical wind energy is the software *Bladed* in which the actual control software can be integrated via a dynamic linked library (DLL) [15]. To allow the use of the same code base as for the actual plant, the PLC software must be compilable to such a DLL and to other targets.

Fulfillment in the proposed approach: The PLC software is programmed with C++ and the Qt cross platform framework. So many different targets are possible and this requirement is fulfilled.

E) *Modularity and hardware-in-the-loop ability (must)*
The PLC must be modular to form a distributed control system. With that, the PLC is neither limited to one type of wing concept nor tether concept, i.e. parts of the PLC may be placed on the ground and on the wing to control actuators, read sensors and process data. Because all these sensors and actuators might be from different manufacturers and might be exchanged during the development, it must be simple to add, exchange and remove heterogenous parts. This modularity would also make it simple to replace actuators and sensors (individually) by a model, i.e. hardware-in-the-loop.

Fulfillment in the proposed approach: Any type of computer can be used that runs with Linux. The computers communicate via IP links over ethernet or WLAN. Through the hardware interfaces of the embedded computers, sensors and actuators can be connected or emulated by models on that computers. Consequently, this requirement is fulfilled.

F) *Inexpensive and open source hardware and software (must)*
In particular for research prototypes for universities, the fundings for hardware is often limited. So the PLC hardware and software licenses must be inexpensive. Preferably, everything is open source to not only be unbound to a specific manufacturer but also to have maximum freedom and deep insight in how the system works. Hence, the PLC software developers would have the possibility to change and extend the PLC to the given application requirements.

Fulfillment in the proposed approach: On the software side, this requirement is fulfilled since only open source software is used. On the hardware side, the mentioned BeagleBone Black and Raspberry Pi (Model B) are available for 45 $[14] and 35 $[15] and are (partly) open source. So, this requirement is fulfilled.

G) *Extensibility and interface ability (must)*
In order to communicate with sensors and actuators, which is a crucial PLC feature, the hardware components must have standard communication interfaces as CAN bus or RS232[16]. It must also be possible to extend the system with other communication protocols. Hence, the computers should have low level interfaces like SPI[17] or I2C[18]. To allow direct measurements of digital and analog voltages, digital and analog general purpose input outputs (GPIOs) should be provided.

Fulfillment in the proposed approach: Only the high level controllers, e.g. the three nested controllers from Fig. 2.8 on p. 25, run on the PLC, i.e. on the ordinary and the mentioned embedded computers with a Preempt-RT Linux. The embedded computers are used to communicate with the lower level controllers like the power electronics

[14] Price by Newark/element14: http://www.newark.com/circuitco/bb-bblk-000/dev-board-am3358-59-arm-mpu-beaglebone/dp/65W6016, accessed: December 07, 2013.
[15] Price by Newark/element14: http://www.newark.com/raspberry-pi/raspbrry-modb-512m/model-b-assembled-board-only/dp/43W5302, accessed: December 07, 2013.
[16] Point to point serial bus.
[17] Serial Peripheral Interface, serial bus.
[18] Inter-Integrated Circuit, serial bus.

controllers. These controllers are assumed to be connected to the PLC via a standard interface like CAN bus with which also measured data, e.g. current, angular speed and angular position, is transmitted to the PLC. Both mentioned embedded computers come with the low level interfaces RS232, SPI, I2C, GPIOs and the BeagleBone Black also with CAN bus and analog GPIOs [16, 17]. With the GPIOs, sensors may also be connected directly to the PLC. One promising possibility for further extensions is to use the open source and inexpensive GNUBLIN[19]boards. However, in order to operate these interfaces and additional boards, platform specific code and libraries are necessary. Compared to industrial PLCs, homogenous input and output cards for a great variety of interfaces are not available which limits the interface possibilities. However, the mentioned restrictions and open questions seem to be manageable for the research prototype and this requirement is seen as fulfilled.

H) *Small and light hardware parts (important)*
Since parts of the PLC may fly inside a control pod under the wing or attached to or inside the wing, the hardware parts should be small and light. This would not only increase the efficiency but would also lower the risk of damages due to crashes.

Fulfillment in the proposed approach: With the use of the mentioned credit card sized embedded computers, this requirement is fulfilled.

I) *Low overhead, fast execution (important)*
In order to run complex controllers and in particular detailed AWE models in the case of software- and hardware-in-the-loop, the applications must be executed fast and with low overhead. A control frequency of minimum 100 Hz (i.e. control period time of 10 ms) which is in the same magnitude as it is used for classical wind energy plants [18] should be realizable. For that, the communication between the parts of the distributed PLC should have a low overhead and must be fast as well.

Fulfillment in the proposed approach: The PLC software is proposed to be implemented in C++, which is slightly slower then C but much simpler to write through object orientation. So the overhead for the programming language is almost minimal. However, the QtPLC library and the final applications must be implemented well to hold that advantage. High computational intensive controllers and models are proposed to run on ordinary computers which are available with high computational powers. With the proposed communication protocol the signal values are exchanged in binary format so that almost only the pure necessary bytes are sent between the nodes (see next chapter). In the example applications described in the next chapter a period time of 10 ms was achieved easily. So low overhead and fast execution are fulfilled.

J) *Small dead times (important)*
A distributed PLC may suffer from high dead times through communication delays. For any controller the highest dead times within the control loop limit the control performance. Hence, dead times should be as low as possible.

Fulfillment in the proposed approach: With the chosen starting point of the communication scheme, a low dead time is only achieved if the control cycle period time is low. Since a control cycle time of 10 ms is easily achievable, i.e. a low dead time per

[19] embedded projects GmbH: "GNUBLIN". http://gnublin.embedded-projects.net/, accessed: November 24, 2013

communication of 10 ms, and since this scheme is only the starting point and future versions may implement other schemes, this requirement is seen to be fulfilled.

K) *Reliability (important)*
The system should be reliable and should have performance monitoring and checks, e.g. watchdogs.

Fulfillment in the proposed approach: It will be tough to gain the same reliability of an industrial PLC, at least in the medium run. The reason for this, is the usage of non-real time ethernet and the from scratch developed QtPLC library, which is untested on a real plant, yet. However, monitoring features like watchdogs are implemented in QtPLC to gain a minimum of reliability.

In conclusion, the most important features are fulfilled or achievable with the proposed approach. The last point, reliability, is not yet a big issue, since it is mainly intended for prototyping. However, if this approach wants to be used by companies or for a later spin-off without starting to develop the PLC with an industrial solution from scratch, this issue is to be kept in mind.

The more important issue is the real time requirement which is discussed in detail in the next section.

3.3.2 "Real time"

A qualitative definition of "real time" When dealing with PLCs "real time" means qualitatively that the computations hold time requirements that are to be specified. The term must not be mixed with "real time clock", i.e. a hardware clock a computer may have which runs on a few kilohertz with a small battery to count the clock time in absence of power, so that the operating system right after booting knows the correct system time. For a PLC, real time means

1. that the whole input-process-output sequence, illustrated in Fig. 3.2 on p. 29, must be executed with the given control frequency within given jitter constraints, i.e. the control frequency stays constant in given constraints, and

2. that the input-process-output has to be finished within the control period time, i.e. the workload time plus eventual overhead time must be lower than the control period time.

A constant control cycle time is needed, because then continuos time controllers can be designed and later converted to discrete ones with fixed cycle time [19, pp. 399]. Furthermore the PLC or control algorithms must output the correct signal values on time otherwise the calculated results become unusable.

Whether the input-process-output sequence is finished within the control period time, depends on the used hardware, the software implementation, the system's scheduling and the overall system load. In contrast, meeting the jitter constraints depends mainly on the system's scheduling and the availability of precise timers.

Quantitative definitions of "real time" "Real time" is often divided into "soft real time" and "hard real time". However, the exact definition of these terms vary. [20, 21, 22] It is also not simple to make a good distinction between a real time operating system and an ordinary operating system. Steven Rostedt, a Red Hat developer of the Preempt-RT real time patch for Linux, emphasizes this jokingly:

> "[...] all operating systems are real-time. That is, they all have some kind of deadline, even Windows. If you hit a key and the computer doesn't respond in say 5

> minutes, you are likely to throw the computer out the window. It failed to meet its deadline. When your deadlines are big enough, pretty much any operating system will do."
> —Steven Rostedt (Red Hat Preempt-RT developer) [23]

So it has to be defined how strong the real time requirements for the intended application are and then a system that meets that requirements must be chosen.

In this thesis, the following quantitative real time terms are defined:

1. No real time: There are no time constraints and thus there is no dedicated mechanism at all that forces to hold time constraints of the above described input-process-output sequence.

   An example is a complex fluid dynamics simulation for which the computation of one second takes a whole day. Obviously, there are no special requirements to the hard- and software of the used computer.

2. Soft real time: The mean of the jitter is approximately zero, i.e. the control cycle frequency is approximately constant.

   An example is a simulation whose simulation time equals the actual physical time (regardless an offset) and thus "seems" to run in real time. Another example is the playing of a video stream. Soft real time can be reached by any operating system, depending on the system load.

3. Hard real time: The mean and absolute value of the jitter stay within given constraints for the whole or for the majority of the time. This may be forced by a dedicated mechanism e.g. with real time task priorities of "real time operating systems". The exact maximum absolute value and the exact mean value of the jitter have to be specified. Additionally, the percentage of the time for which these constraints always have to be satisfied are specified and may be monitored. The compliance of the specified constraints have to be measured for a given system.

   For control cycle frequencies in the magnitude of 100 Hz and above, hard real time is only reached by real time operating systems, like the here proposed Preempt-RT patched Linux.

4. Provable real time: There is a highly deterministic mechanism, that forces the mean and absolute value of the jitter into given constraints (which are usually lower then for hard real time). Hard real time is already more deterministic than soft real time, but for provable real time, the compliance of the jitter constraints are provable mathematically. It is also possible to calculate a worst case execution time of the system. I.e. neglecting hardware and software faults, the input-process-output sequence is *always* executed with the control frequency plus/minus the small mathematically/statistically calculated jitter and is *always* finished within the mathematically calculated worst case execution time.

   Provable real time is usually only reached with micro controllers with very simple scheduling e.g. through interrupts. This type of real time is only needed where a real time violation is likely to lead to a not functioning system, high damage or human deaths. An example is the airbag controller of a car or the power electronics controller of the electrical machines.

It has to be pointed out, that these cases are defined for this thesis. In particular the terms "soft" and "hard" are sometimes defined slightly different or no distinction is made about the provability, see e.g. [20, 21, 22, 24].

Real time in the proposed approach and applied to a crosswind AWE prototype "No real time" is not an option and "provable real time" for the higher level controllers, like the power or flight controllers, that shall run on the PLC, is not needed. "Soft real time" is fine for simulations (software-in-the-loop) on ordinary computers with ordinary operating systems. The decision is clearly "hard real time" with the use of a real time operating system because much more reliability is gained in particular for control period times in the magnitude of 10 ms and below.

In [20] the Preempt-RT patch was tested and compared with other real time Linux variants on a *Beagle Board* with the result that even under heavy load the magnitude of the jitter

- is always below 158 µs [20, p. 11, tab. 6],
- in 95 % of the time even below 47 µs [20, p. 11, tab. 6]
- with a median of −1 µs [20, p. 11, tab. 6].

The highest possible frequency for which the jitter never exceeds 50 % of the control cycle period time was approximated to 3.16 kHz [20, p. 11, tab. 7] and for which the jitter does not exceed that limit for 95 % of the time to 10.64 kHz [20, p. 11, tab. 7]. So with these results, Preempt-RT is suitable for specifications B) and I).

It has to be mentioned, that in [20] also an unpatched Linux kernel was tested, where the *CONFIG_PREEMPT* Linux kernel build flag was set and the real time process was run as *chrt* user. The result again under heavy load was a jitter below 69 µs for 95 % [20, p. 11, tab. 6] of the time. However, for the complete measurement time of only two hours the maximum jitter magnitude was more than 1 ms [20, p. 11, tab. 6] and is likely to be even higher for longer testing times due to kernel or other low level routines [20]. Thus, using this solution for the application of a crosswind AWE PLC may not be excluded but should only be used, if the control cycle period time is way above 10 ms.

There are also investigations to determine the real time behavior of standard IP communications: In [25] among others the timing of sending UDP packets between two single board computers running an ordinary Linux was measured. The average transmission time including jitter for a 256 Byte message was reported to be $\approx 35 \ldots 50$ µs [25, Fig. 4]. The transmission time increases linearly with the size of the message (with an offset due to overhead) so that for a message of 1024 Byte an average delay time of ≈ 75 µs [25, Fig. 1] is denoted. Surprisingly, the real time operating system *VxWorks* performed much worse. Only the RTnet [26] real time ethernet extensions for RTAI and Xenomai and a pure data link layer communication, i.e. the layer right above the physical layer in the Open Systems Interconnection (OSI) model, performed slightly better.

Truly, the communication delay is not guarantied because ordinary IP sockets instead of real time ethernet is used. To make it more likely that packets are sent without high delay from sender to receiver, the computers of the actual PLC should not run unnecessary processes which use the network like automatic updaters. Additionally, the network topology can further minimize delays induced by congestions and similar, e.g. by connecting all computers in a subnet by an ethernet switch [27]. Besides that, the packets are sent by the high prioritized real time processes. With the high reachable transmission rates of ethernet of up to 1 GBit/s and above, the delay that is caused by the lack of prioritization should be limited by the time needed to send the remaining packets that are already buffered on the network card. This delay time is already included in the above cited results.

In conclusion,

- regarding the proposed Preempt-RT with a maximum jitter of $t_\mathrm{j} = 158$ µs [20, p. 11, tab. 6],

- regarding the communication scheme where the master informs the slaves about a timeout via a message,
- regarding that each timeout message with the attached input signals does not exceed a size of 256 Byte, which equals 64 pure floating point values, and thus the message delay is always below $t_\mathrm{m} = 50\,\mu\mathrm{s}$ [25, Fig. 4],
- regarding that 10 slave processes are registered to the master and that all timeout messages are sent consecutively (for UDP this can be optimized through a broadcast message), and
- assuming that the above presented measured values hold for the specifically proposed system,

then the maximum jitter per timeout the 10th slave would have is

$$\begin{aligned} t_{10} &= t_\mathrm{j} + 10\,t_\mathrm{m} \\ &= 158\,\mu\mathrm{s} + 10 \cdot 50\,\mu\mathrm{s} \\ &= 658\,\mu\mathrm{s}. \end{aligned} \tag{3.1}$$

This value is far below 10 % of a control cycle period time of 10 ms and is thus evaluated as suitable. It is further evaluated that the determinism, which would be gained from a real time ethernet like RTnet in conjunction with RTAI or Xenomai, is not worth to be implemented. This would exclude the use of Preempt-RT and several mentioned advantages would be lost. However, if this determinism is needed in a future application, it should be not that hard to add it in QtPLC by using other libraries in addition to Qt.

With the starting point of the communication scheme, TCP instead of UDP is used. In the next version of QtPLC the possibility to use UDP will be implemented to gain a lower overhead. The magnitude of the overall jitter in Eq. (3.1) is below 1 ms. In conclusion, although the UDP communication needs to implemented and the absolute jitter has to measured in the final system, the real time requirement for the proposed PLC approach is seen as fulfilled.

3.4 PLCs used by AWE researchers and companies

In this section the PLC solutions of research groups and companies of the AWE community are outlined. The section raises no claim to completeness, but the solutions of important researchers and companies, including SkySails and the TU Delft, are discussed. These two solutions are also compared to the proposed approach of this thesis. In general, the PLC solutions are quite different and particularly self-developed which emphasizes that there is no ideal solution.

3.4.1 SkySails

The company SkySails, located in Hamburg, Germany, presents theirs approach in [1, pp. 599]. SkySails offers commercially a kite system for supporting ship propulsion with automatic start and landing. The kite is steered via a control pod under the kite. The pod is powered over a conducting connection of a non-standard tether. With its sensors, actuators and an emergency battery, the pod can steer the kite autonomously in the case of an incident. Otherwise, pod and ship communicate with each other over the tether or via a wireless link. Further sensors and actuators are on the ship. [1, pp. 599]

Many parts of the distributed control system are self-designed to meet the requirements. Inside the pod is a servo controller and an embedded flight computer each on a self-designed board. They communicate via CAN bus. On the ship is the main computer with a (not further specified)

real time Linux which communicates to the flight controller and to an industrial PLC. The latter controls the hydraulic winches and the telescope boom for start and landing and is chosen because an industrial PLC meets "the functional safety requirements of ISO 13849" [1, p. 605]. Via ethernet/TCP, the industrial PLC, the main computer and the GUI on the ship's bridge or optionally other GUI computers can interact with each other. It is stated, that the GUI on the ship's bridge was very important for visualization and control. SkySails chose a control frequency of 10 Hz [1, p. 600]. For each cycle approximately 420 [1, p. 600] sensor, auxiliary and debug values are recorded and stored for 10 days [1, p. 600] on the main computer. For the system design, special focus was paid to reliability, because it is an industrial product. [1, pp. 599]

The timing of the computers in the control pod are synchronized with a CAN message. They encapsulated the C++ classes so, that these parts of the software can be developed and tested on an ordinary computer. For the ethernet/TCP connection, a self designed code generation tool is used that takes an XML definition file for the different exchanged signals. [1, pp. 599]

Many design considerations are similar to the proposed approach of this thesis: A distributed control system with heterogenous hardware is used where ordinary computers run high computational intensive tasks and embedded computers are located in remote places or to interface with sensors and actuators. Interestingly, even the simple time synchronization scheme with a timeout message is partly congruent. QtPLC includes the QtPLC Control Center GUI with an extensible API. The communication between the nodes of the distributed control system seems to be less complex with QtPLC: Instead of using a code generation tool, the exchangeable signals are defined in an INI file that is read from the processes shortly after the start (see next chapter). Additionally, not only parts of the code but the complete PLC software can be developed software-in-the-loop with the QtPLC library.

3.4.2 TU Delft

The TU Delft, namely the team of Dr.-Ing. Roland Schmehl, investigates airborne wind energy for more then ten years. They are specialized in a wide field of lift AWE including the investigation of different wing types, modeling, aerodynamic optimization, control, etc. They currently have a 20 kW [1, p. 404] kite power demonstrator with kites of up to 50 m^2 [1, p. 404] area. They use a battery powered control pod and one (ordinary) tether to the ground station. Several sensors are involved including an IMU attached to the kite. The control pod, IMU and ground station communicate via wireless links. [5, pp. 5]

Recently, the control system of the TU Delft was redesigned. Many design considerations are presented in [11]: The distributed control system is modular. The main computer is a desktop computer running a standard Linux, configured for embedded applications. It is stated, that this is a powerful solution because new controllers could be developed easily. Additionally, more complex algorithms like nonlinear model predictive control could be executed on such a powerful computer. Two more different computers are on the ground and two on the airborne. Several communication protocols like ethernet, CAN bus and WLAN are used. The control cycle frequency is 20 Hz [11]. The nodes' clocks are synchronized to a precision of at least 1 ms using a network time protocol server. Several applications including control processes, GUI processes, a data logger process or a clock daemon were implemented. [11]

For the less time critical software components a high level software framework based on ZeroMQ was developed. ZeroMQ provides a socket abstraction with different communication schemes. It supports several programming languages. For the slower wireless links, UDP is chosen. For the data serialization, Google protocol buffers is used with which messages can be defined in a specialized file. This file is then fed to the Google protocol buffers code generator which can output all major programming languages. For the winch control computer, a Xenomai real time Linux and the software framework *OROCOS* are chosen. [11]

Before that redesign, an XBee wireless link was used for the communication between ground and control pod [11]. With that [5, p. 61] reported several crashes due to connection losses with which the kite flow uncontrolled. In the new design a WLAN 802.11n link with a WLAN-n router is used in the following manner: For the control pod two antennas and for the ground station a directed antenna is used and the 5.5 GHz frequency band is chosen. With that interferences are avoided and maximum reliability is gained. With this fast wireless link it is even uncritical if packets have to be sent two or three times due to losses. During a test flight, including start, landing and power production with tether lengths of up to 1000 m, not a single packet loss was recorded. For the about 150 byte packets a mean round trip time of 4.7 ms was measured. [11]

There are again several similarities between proposed approach of this thesis and the TU Delft's solution, namely the use of heterogenous hardware, the use of several software applications and the use of ethernet as main communication channel. A beneficial approach is to use ZeroMQ and Google protocol buffers to gain independency of the programming language. Not all sources of the developed frameworks and applications are published, yet. But Uwe Fechner told at the Airborne Wind Energy Conference 2013 that he intents to publish everything under an open source license sometime in 2014. It would be an interesting task then to compare QtPLC with the TU Delft's software and carry out optimizations.

OROCOS (Open Robot Control Software), that is only used for the winch control computer of the TU Delft, is a cross-platform and open source C++ framework intended for machine and robot control. It has an operating system abstraction for the creation of threads with a thread-save communication. It is ported to the major operating systems including the real time Linux patches RTAI/LXRT and Xenomai. With the CORBA Library it is also possible to form a distributed control system over ethernet. [28] So OROCOS could be a suitable alternative on the software side of the proposed PLC, i.e. to the QtPLC library. A comparison of both software solutions is out of scope of this thesis but should be performed in a future work.

3.4.3 Other AWE researchers and companies

The research group of the KU Leuven, Belgium, under Prof. Dr. Moritz Diehl, are specialized in nonlinear model predictive control in lift AWE. They built a small prototype with a model airplane for rotatory start and landing. Besides micro controllers for the low level control, a Xenomai Linux on a high end desktop computer is used to run the complex nonlinear model predictive controllers. As underlying framework OROCOS is used. [1, pp. 465]

KITEnergy from Italy uses LabVIEW and the PXI real time platform of National Instruments. The communication with the sensors and actuators are established via Profibus, RS232 and other. Via ethernet a PC running a GUI is connected. As control cycle time 100 ms is chosen. [1, pp. 379]

EnerKite, a startup based in Berlin, built a 60 kW [1, pp. 427] demonstrator on a truck. They use a ram air inflated kite with three tethers to three winches on the ground. [1, pp. 427] From a dialogue with the EnerKite engineers on the Airborne Wind Energy Conference 2013, the information was fetched that they once used an embedded development board for the control. However, they quite early switched to an industrial PLC from Bachmann, which is also used in classical wind energy.

3.4.4 Other solutions

There is a range of industrial PLC offers from companies like Beckhoff, Bachmann, Siemens or dSpace which should meet many of the above specified requirements for a lift AWE prototype. However, such a solution was excluded because the costs of such a PLC are too high for the usually very limited fundings for university research prototypes.

There are also a couple of open source projects that deal with PLCs. Most of them utilize a micro controller to build a general purpose PLC device. Examples are "Brick Open Source PLC"[20,21], "open-plc"[22] or "ClassicLadder"[23]. A micro controller, however, has not enough computational power to run complex controllers and in particular complex models for lift AWE. There are also a couple of open source supervisory control and data acquisition (SCADA) tools and libraries like "OpenAPC"[24] or "pvbrowser"[25]. SCADAs are basically GUIs instead of PLCs and are thus only on the management level in the automation pyramid. There are a few other open source PLC projects, but non of them seems to be promising for a lift AWE prototype.

Several open source and inexpensive boards for embedded development are available, namely the "Arduino"[26] utilizing a micro controller or the above mentioned BeagleBone Black and Raspberry Pi. Very interesting for interfacing with sensors and actuators are also the above mentioned GNUBLIN boards.

The actual goal of the proposed PLC approach with the QtPLC library is to use such open source embedded boards (which can run a Linux) for a distributed PLC. But also ordinary computers, be it a portable laptop or a high end desktop computer, are utilizable while making the development as simple as possible. Instead of using one of the PLC programming languages from IEC 61131 like structured text or even ladder diagram, the use of the object oriented C++ language and higher level tools like Matlab/Simulink is forced. Other programming languages may also be used in the future. Additionally, the QtPLC Control Center GUI application may be utilized as SCADA.

[20] Smith, W.: "Small Brick Open Source PLC". http://startingelectronics.com/projects/small-open-source-PLC, accessed: November 26, 2013.

[21] Smith, W.: "Large Brick Open Source PLC". http://startingelectronics.com/projects/large-open-source-PLC, accessed: November 26, 2013.

[22] "open-plc". http://code.google.com/p/open-plc/wiki/Main?tm=6, accessed: November 26, 2013.

[23] Le Douarain, Marc: "ClassicLadder". https://sites.google.com/site/classicladder, accessed: November 26, 2013.

[24] "OpenAPC – Open Advanced Process Control". http://www.openapc.com, accessed: November 26, 2013.

[25] Lehrig, S.: "pvbrowser". http://pvbrowser.de, accessed: November 26, 2013.

[26] "Arduino". http://arduino.cc, accessed: 26.11.2013.

4 Introduction to the QtPLC API using the example of a lift airborne wind energy plant model

In this chapter, first a lift AWE plant is modeled mathematically, implemented in Matlab/Simulink and exported to C++. Instead of explaining abstractly the QtPLC API alone, the most important parts and concepts are explained in detail by implementing the example lift AWE model with the QtPLC library and the Qt Creator IDE. This may also be seen as a QtPLC tutorial. Exemplary results of the execution performance of the QtPLC processes are presented for two test system setups. The chapter closes with the interpretation of some simulation results of the lift AWE plant.

4.1 Lift AWE plant modeling in Simulink and export to C++

The simplified diagram in Fig. 4.1 visualizes the modeling approach of the dynamic lift AWE plant and shows the most important quantities. This model considers

- the wing's dynamics as a mass point with mass $m_\mathrm{b} \in \mathbb{R}_{>0}$ [kg],
- the wing's aerodynamics with lift and drag coefficients for different angles of attack, $c_\mathrm{d}(\alpha)$ and $c_\mathrm{l}(\alpha)$, and
- the tether as a massless, volume less and lossless spring with spring constant $c_\mathrm{t} \in \mathbb{R}_{>0}$ [N/m].

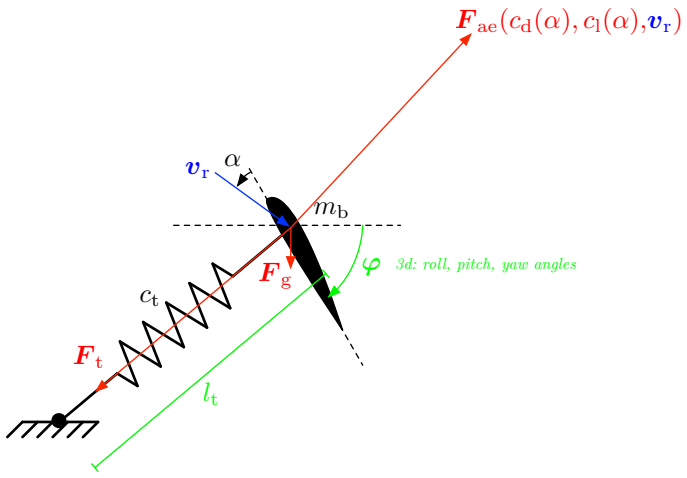


Figure 4.1: Modeling approach of the lift AWE plant with important quantities and properties.

The complete orientation of the wing in three Euler angles $\boldsymbol{\varphi} \in \mathbb{R}\left[(^\circ, ^\circ, ^\circ)^\top\right]$ and the tether length $l_\mathrm{t} \in \mathbb{R}_{>0}\,[\mathrm{m}]$ can be set directly to desired values.

In this section the mathematical model is derived in detail and implemented in Matlab/Simulink. Several source codes are presented. However, not every single line is explained in detail. It is assumed that the reader has good knowledge in Matlab/Simulink.

4.1.1 Dynamics model

A mass point has only 3 DOF. The orientation has no dynamics and is not covered by that model and can be chosen freely. Newton's second axiom states

$$m_\mathrm{b}\boldsymbol{a}_\mathrm{b} = \sum_{i=1}^{n} \boldsymbol{F}_i, \quad n \in \mathbb{N},$$

with the acceleration $\boldsymbol{a}_\mathrm{b} \in \mathbb{R}^3\left[(\mathrm{m/s^2}, \mathrm{m/s^2}, \mathrm{m/s^2})^\top\right]$. The acceleration and the forces are actually time variant, but "(t)" with $t \in \mathbb{R}\,[\mathrm{s}]$ is dropped for simplicity. The forces acting on the mass point are the aerodynamic force $\boldsymbol{F}_\mathrm{ae}$, the tether force $\boldsymbol{F}_\mathrm{t}$ and the gravitational force $\boldsymbol{F}_\mathrm{g}$, i.e.

$$m_\mathrm{b}\boldsymbol{a}_\mathrm{b} = \boldsymbol{F}_\mathrm{ae} + \boldsymbol{F}_\mathrm{t} + \boldsymbol{F}_\mathrm{g}.$$

Newton's second axiom is only valid in inertial, i.e. unaccelerated, reference frames. Here, the earth itself is used as inertial reference frame in cartesian coordinates, denoted by a superscript $^\mathrm{e}$. Although the earth rotates around its own axis and moves around the sun and the sun moves in the galaxy and so on, this is a suitable assumption since the accelerations these effects are comparably low. The wing's acceleration is the derivative of the wing's velocity $\boldsymbol{v}_\mathrm{b}$ and the velocity is the derivative of the wing's position $\boldsymbol{r}_\mathrm{b} \in \mathbb{R}^3\left[(\mathrm{m}, \mathrm{m}, \mathrm{m})^\top\right]$, i.e. $\boldsymbol{a}_\mathrm{b} = \dot{\boldsymbol{v}}_\mathrm{b} = \ddot{\boldsymbol{r}}_\mathrm{b}$. The gravitational force in the earth fixed cartesian coordinate system has only a z component which is the mass multiplied by the gravitational acceleration $\mathrm{g} \approx 9.81\,\mathrm{m/s^2}$. Consequently, the differential equation

$$m_\mathrm{b}\ddot{\boldsymbol{r}}^\mathrm{e}_\mathrm{b} = \boldsymbol{F}^\mathrm{e}_\mathrm{ae} + \boldsymbol{F}^\mathrm{e}_\mathrm{t} + \begin{pmatrix} 0 \\ 0 \\ -m_\mathrm{b}\mathrm{g} \end{pmatrix}, \quad \boldsymbol{r}^\mathrm{e}_\mathrm{b}(0) = \boldsymbol{r}^\mathrm{e}_{\mathrm{b},0}, \quad \dot{\boldsymbol{r}}^\mathrm{e}_\mathrm{b}(0) = \boldsymbol{v}^\mathrm{e}_\mathrm{b}(0) = \boldsymbol{v}^\mathrm{e}_{\mathrm{b},0} \tag{4.1}$$

with initial position $\boldsymbol{r}^\mathrm{e}_{\mathrm{b},0} \in \mathbb{R}^3\left[(\mathrm{m}, \mathrm{m}, \mathrm{m})^\top\right]$ and initial velocity $\boldsymbol{v}^\mathrm{e}_{\mathrm{b},0} \in \mathbb{R}^3\left[(\mathrm{m/s}, \mathrm{m/s}, \mathrm{m/s})^\top\right]$ fully describes the mass point's motion in the earth fixed cartesian coordinate system. Equations for the two missing forces, $\boldsymbol{F}^\mathrm{e}_\mathrm{ae}$ and $\boldsymbol{F}^\mathrm{e}_\mathrm{t}$, are derived in the next two sections.

The gravitational force is calculated in an extra block in the Simulink implementation. Fig. 4.2 shows a screenshot of the "gravitation" subsystem and Fig. 4.3 shows a screenshot of the "mass point dynamics" subsystem.

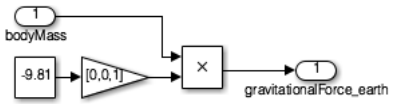


Figure 4.2: Screenshot of the "gravitation" subsystem in Simulink.

4.1.2 Aerodynamics model

The in Sec. 2.1.4 on pp. 17 introduced basic aerodynamics equations are used directly as aerodynamics model. As reference frame the wing or body fixed cartesian coordinate system is

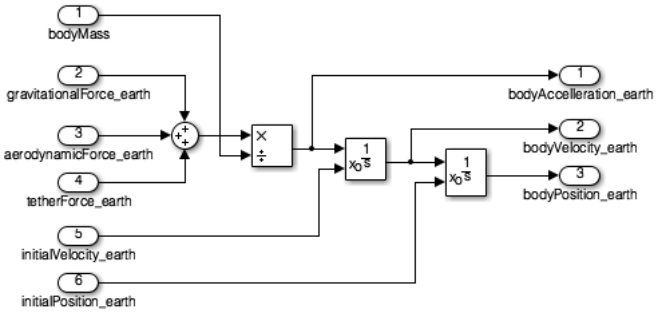


Figure 4.3: Screenshot of the "mass point dynamics" subsystem in Simulink.

used, shown in Fig. 2.5 (right) on p. 18. Quantities in this coordinate system are denoted by a superscript $^{\mathrm{b}}$, i.e.

$$\begin{aligned}
\boldsymbol{F}^{\mathrm{b}}_{\mathrm{ae,d}} &= \frac{1}{2}\rho|\boldsymbol{v}^{\mathrm{b}}_{\mathrm{r}}|^2\, A\, c_{\mathrm{d}}(\alpha)\ \mathrm{dir}\,\boldsymbol{v}^{\mathrm{b}}_{\mathrm{r}} \\
\boldsymbol{F}^{\mathrm{b}}_{\mathrm{ae,l}} &= \frac{1}{2}\rho|\boldsymbol{v}^{\mathrm{b}}_{\mathrm{r}}|^2\, A\, c_{\mathrm{l}}(\alpha)\ \mathrm{dir}\left(\boldsymbol{v}^{\mathrm{b}}_{\mathrm{r}} \times \boldsymbol{y}^{\mathrm{b}}_{\mathrm{b}}\right) \\
\boldsymbol{F}^{\mathrm{b}}_{\mathrm{ae}} &= \boldsymbol{F}^{\mathrm{b}}_{\mathrm{ae,l}} + \boldsymbol{F}^{\mathrm{b}}_{\mathrm{ae,d}} \\
\alpha &= 90\,^\circ - \arccos\left(\frac{\boldsymbol{v}^{\mathrm{b}}_{\mathrm{r}}}{|\boldsymbol{v}^{\mathrm{b}}_{\mathrm{r}}|} - \boldsymbol{z}^{\mathrm{b}}_{\mathrm{b}}\right).
\end{aligned}$$

Here, $c_{\mathrm{d}}(\alpha) = c_{\mathrm{d,i}}(\alpha)$ since a pure lift AWE plant is modeled. As lift and drag coefficients with dependency of α the graphs from [9, p. 7] are used as reference for a cubic spline interpolation with the marked anchor points in Fig. 4.4. The spline's polynomial coefficients were calculated

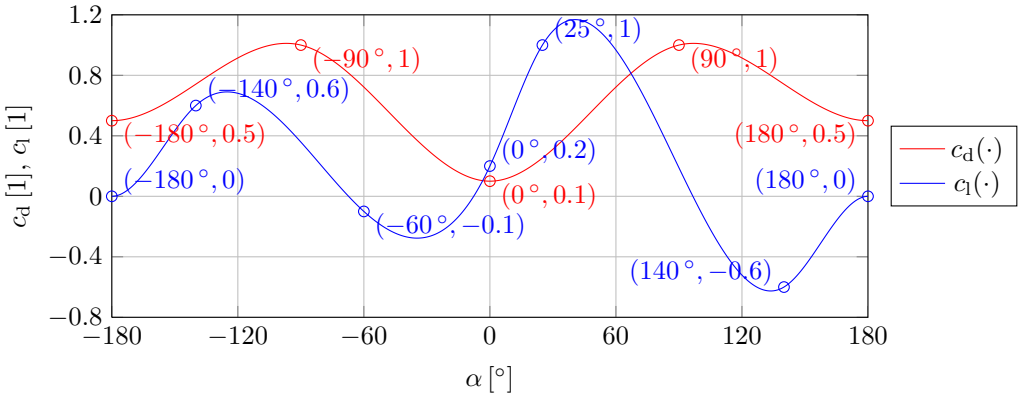


Figure 4.4: Drag and lift coefficients in dependency of the angle of attack, $c_{\mathrm{d}}(\alpha)$ and $c_{\mathrm{l}}(\alpha)$.

using the Matlab function *spline ()* with the code snippets from Lst. 4.1.

```
1 dragCoefficient_anchorAnglesOfAttack = [-180 -90 0 90 180];
2 dragCoefficient_anchorValues = [0.5 1.0 0.1 1.0 0.5];
3
4 dragCoefficient_splineCoefficients = spline(dragCoefficient_anchorAnglesOfAttack, [0
      ↪ dragCoefficient_anchorValues 0]);
5
```

```
liftCoefficient_anchorAnglesOfAttack = [-180 -140 -60 0 25 140 180];
liftCoefficient_anchorValues = [0 0.6 -0.1 0.2 1.0 -0.6 0];

liftCoefficient_splineCoefficients = spline(liftCoefficient_anchorAnglesOfAttack, [0
    ↪ liftCoefficient_anchorValues 0]);
```

Listing 4.1: Code snippet to calculate the spline's polynomial coefficients in Matlab.

All vectors of the mass point dynamics in Eq. (4.1) have to be given in the earth fixed cartesian coordinate system. It would be possible to calculate the aerodynamic force directly in earth fixed coordinate system by calculating the axis vectors $\boldsymbol{y}_{\mathrm{b}}^{\mathrm{e}}$ and $\boldsymbol{z}_{\mathrm{b}}^{\mathrm{e}}$. However, it is more common to transform the relative wind velocity $\boldsymbol{v}_{\mathrm{r}}^{\mathrm{e}} = \boldsymbol{v}_{\mathrm{w}}^{\mathrm{e}} - \boldsymbol{v}_{\mathrm{b}}^{\mathrm{e}}$ into the body fixed coordinate system $\boldsymbol{v}_{\mathrm{r}}^{\mathrm{b}}$ and to transform the calculated total aerodynamic force $\boldsymbol{F}_{\mathrm{ae}}^{\mathrm{b}}$ back into the earth fixed coordinate system $\boldsymbol{F}_{\mathrm{ae}}^{\mathrm{e}}$.

The body fixed coordinate system is rotated and translated dynamically against the earth fixed coordinate system. Since here only velocity and force vectors are to be transformed between the two coordinate systems (i.e. no position vector is transformed), the translation is dropped. A possibility to describe mathematically an arbitrary orientation in the three dimensional space is by consecutive rotations around the x, y and z axis of a cartesian coordinate system with the angles α (roll), β (pitch or elevation) and γ (yaw or azimuth). This is the x-y-z Euler angle rotation sequence whose steps are listed in detail below:

1. The rotation matrix

$$\boldsymbol{R}_{\boldsymbol{x}_{\mathrm{e}}}(\alpha) := \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

   with the roll angle α is applied to rotate a vector around the x axis of the earth fixed coordinate system, i.e. around the basis vector $\boldsymbol{x}_{\mathrm{e}}$.

2. The rotation matrix

$$\boldsymbol{R}_{\boldsymbol{y}_{\mathrm{e}}}(\beta) := \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}$$

   with the pitch or elevation angle β is applied to rotate a vector around the y axis of the earth fixed coordinate system, i.e. around the basis vector $\boldsymbol{y}_{\mathrm{e}}$.

3. The rotation matrix

$$\boldsymbol{R}_{\boldsymbol{z}_{\mathrm{e}}}(\gamma) := \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

   with the yaw or azimuth angle γ is applied to rotate a vector around the z axis of the earth fixed coordinate system, i.e. around the basis vector $\boldsymbol{z}_{\mathrm{e}}$. [29, pp. 10]

Altogether that forms the assembled transformation matrix

$$\begin{aligned} \boldsymbol{R}_{\boldsymbol{z}_{\mathrm{e}}\boldsymbol{y}_{\mathrm{e}}\boldsymbol{x}_{\mathrm{e}}}(\gamma,\beta,\alpha) &:= \boldsymbol{R}_{\mathrm{z_e}}(\gamma)\boldsymbol{R}_{\mathrm{y_e}}(\beta)\boldsymbol{R}_{\mathrm{x_e}}(\alpha) \\ &= \begin{pmatrix} \cos(\beta)\cos(\gamma) & \cos(\gamma)\sin(\alpha)\sin(\beta) - \cos(\alpha)\sin(\gamma) & \sin(\alpha)\sin(\gamma) + \cos(\alpha)\cos(\gamma)\sin(\beta) \\ \cos(\beta)\sin(\gamma) & \cos(\alpha)\cos(\gamma) + \sin(\alpha)\sin(\beta)\sin(\gamma) & \cos(\alpha)\sin(\beta)\sin(\gamma) - \cos(\gamma)\sin(\alpha) \\ -\sin(\beta) & \cos(\beta)\sin(\alpha) & \cos(\alpha)\cos(\beta) \end{pmatrix}. \end{aligned}$$

Note that matrix multiplications are performed from the right side so that the sequence of the last equation has the correct order. The three angles may be composed to the vector $\boldsymbol{\varphi} = (\alpha, \beta, \gamma)^\top$. Throughout this thesis

- all cartesian coordinate systems are right handed with east-north-up axes,
- these three angle symbols, α, β and γ, are always used for the rotations around the respective x, y or z axis, with the right hand rule[27] and
- only the presented x-y-z Euler rotation sequence with the axes of the earth fixed coordinate system as reference is used.

Although the orientation of the wing can be described with only one sequence of the three rotations, two rotation sequences are applied:

1. The first rotation is the control orientation where a pitch angle β_c and a yaw angle γ_c can be applied freely to the wing. The pitch angle is used to power/depower the wing and the yaw angle is used to steer the wing into the desired direction on the small earth sphere. Throughout this example lift AWE model, a roll angle different from zero is not considered.
2. Consider a pitch angle of $\beta_\mathrm{c} = 0$. When the tethered wing flies over the small earth sphere, it changes its orientation automatically such that a person standing at the tether escape point would always see the bottom of the wing. This behavior is neither covered by the 3 DOF mass point dynamics nor by the aerodynamic forces. So it is artificially added in this second orientation transformation which depends on the wing's elevation and azimuth angle. These angles are calculated via the transformation of the wing's position from the earth fixed cartesian coordinate system to the earth fixed spherical coordinate system whose origins are equal, shown in Fig. 4.5: If the position vector of the wing in the earth fixed

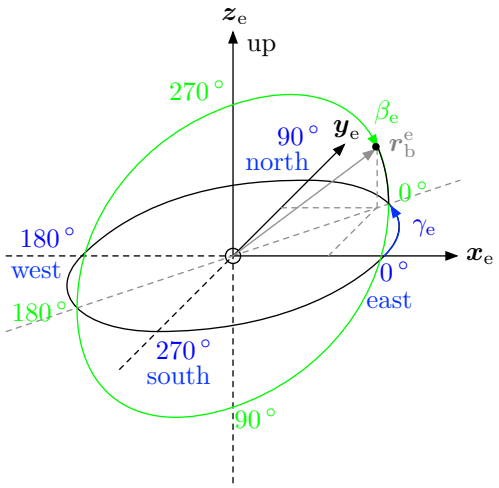


Figure 4.5: Earth fixed spherical coordinate system.

[27] In literature, e.g. in [1], often the cartesian coordinate system is right handed and east-north-up axes are used for the earth fixed reference frame, but east-south-down axes are used for the body fixed reference frame. Additionally, often the symbols ϕ (roll), θ (pitch) and Ψ (yaw) are used for the angles. The in this thesis used axes orientations and angle symbols are chosen differently because they seemed to be more consistent and easier to remember.

coordinate system is composed of $\boldsymbol{r}_{\mathrm{b}}^{\mathrm{e}} = (r_{\mathrm{b,x}}^{\mathrm{e}}, r_{\mathrm{b,y}}^{\mathrm{e}}, r_{\mathrm{b,z}}^{\mathrm{e}})^{\top}$, then the elevation and azimuth angles are determined by

$$\beta_{\mathrm{e}} = -\arctan\left(\frac{r_{\mathrm{b,z}}^{\mathrm{e}}}{\sqrt{{r_{\mathrm{b,x}}^{\mathrm{e}}}^2 + {r_{\mathrm{b,y}}^{\mathrm{e}}}^2}}\right)$$
$$\gamma_{\mathrm{e}} = \arctan\left(\frac{r_{\mathrm{b,y}}^{\mathrm{e}}}{r_{\mathrm{b,x}}^{\mathrm{e}}}\right).$$

So, for this position dependent orientation, firstly the wing is rotated around $\boldsymbol{y}_{\mathrm{e}}$ with the elevation angle $\beta_{\mathrm{e}} + 90\,^{\circ}$. The offset of $+90\,^{\circ}$ accounts for the basic orientation of the wing, shown Fig. 2.5 (right) on p. 18: Consider a control orientation of $\beta_{\mathrm{c}} = 0$ and $\gamma_{\mathrm{c}} = 0$ and an elevation angle of $\beta_{\mathrm{e}} = -45\,^{\circ}$ (all rotations are right handed), i.e. the wing flies towards the zenith and is on its half way exactly between zenith and maximum power position. Without an offset of $+90\,^{\circ}$ the person at the tether escape point would see the front of the wing instead of its bottom. Secondly, the wing is rotated around the $\boldsymbol{z}_{\mathrm{e}}$ axis with the azimuth angle γ_{e}, otherwise the person at the tether escape point would not see the bottom of the wing for arbitrary γ_{e}.

All together the wing's orientation is determined by the transformation matrix

$$\begin{aligned}\boldsymbol{T}_{\mathrm{b}\to\mathrm{e}} &:= \boldsymbol{R}_{\boldsymbol{z}_{\mathrm{e}}\boldsymbol{y}_{\mathrm{e}}\boldsymbol{x}_{\mathrm{e}}}(\gamma_{\mathrm{e}}, \beta_{\mathrm{e}} + 90\,^{\circ}, 0)\, \boldsymbol{R}_{\boldsymbol{z}_{\mathrm{e}}\boldsymbol{y}_{\mathrm{e}}\boldsymbol{x}_{\mathrm{e}}}(\gamma_{\mathrm{c}}, \beta_{\mathrm{c}}, 0) \\ &= \boldsymbol{R}_{\mathrm{z_e}}(\gamma_{\mathrm{e}})\boldsymbol{R}_{\mathrm{y_e}}(\beta_{\mathrm{e}} + 90\,^{\circ})\boldsymbol{R}_{\mathrm{x_e}}(0)\, \boldsymbol{R}_{\mathrm{z_e}}(\gamma_{\mathrm{c}})\boldsymbol{R}_{\mathrm{y_e}}(\beta_{\mathrm{c}})\boldsymbol{R}_{\mathrm{x_e}}(0) \\ &= \boldsymbol{R}_{\mathrm{z_e}}(\gamma_{\mathrm{e}})\boldsymbol{R}_{\mathrm{y_e}}(\beta_{\mathrm{e}} + 90\,^{\circ})\boldsymbol{R}_{\mathrm{z_e}}(\gamma_{\mathrm{c}})\boldsymbol{R}_{\mathrm{y_e}}(\beta_{\mathrm{c}}). \end{aligned} \tag{4.2}$$

In order to transform a velocity or a force vector of the earth fixed coordinate system, for the moment represented by $\boldsymbol{\chi}^{\mathrm{e}} \in \mathbb{R}^3$, into the body fixed coordinate system, $\boldsymbol{\chi}^{\mathrm{b}}$, the transformation

$$\boldsymbol{\chi}^{\mathrm{b}} = \boldsymbol{T}_{\mathrm{e}\to\mathrm{b}}\, \boldsymbol{\chi}^{\mathrm{e}} = \boldsymbol{T}_{\mathrm{b}\to\mathrm{e}}^{-1}\, \boldsymbol{\chi}^{\mathrm{e}}$$

has to applied with

$$\begin{aligned}\boldsymbol{T}_{\mathrm{e}\to\mathrm{b}} = \boldsymbol{T}_{\mathrm{b}\to\mathrm{e}}^{-1} &= \boldsymbol{R}_{\boldsymbol{z}_{\mathrm{e}}\boldsymbol{y}_{\mathrm{e}}\boldsymbol{x}_{\mathrm{e}}}^{-1}(\gamma_{\mathrm{c}}, \beta_{\mathrm{c}}, 0)\, \boldsymbol{R}_{\boldsymbol{z}_{\mathrm{e}}\boldsymbol{y}_{\mathrm{e}}\boldsymbol{x}_{\mathrm{e}}}^{-1}(\gamma_{\mathrm{e}}, \beta_{\mathrm{e}} + 90\,^{\circ}, 0) \\ &= \boldsymbol{R}_{\mathrm{x_e}}(0)\boldsymbol{R}_{\mathrm{z_e}}(-\gamma_{\mathrm{c}})\boldsymbol{R}_{\mathrm{y_e}}(-\beta_{\mathrm{c}})\, \boldsymbol{R}_{\mathrm{x_e}}(0)\boldsymbol{R}_{\mathrm{y_e}}(-\beta_{\mathrm{e}} - 90\,^{\circ})\boldsymbol{R}_{\mathrm{z_e}}(-\gamma_{\mathrm{e}}) \\ &= \boldsymbol{R}_{\mathrm{y_e}}(-\beta_{\mathrm{c}})\boldsymbol{R}_{\mathrm{z_e}}(-\gamma_{\mathrm{c}})\boldsymbol{R}_{\mathrm{y_e}}(-\beta_{\mathrm{e}} - 90\,^{\circ})\boldsymbol{R}_{\mathrm{z_e}}(-\gamma_{\mathrm{e}}). \end{aligned} \tag{4.3}$$

Note, as written in Eq. (4.3), the inverse transformation matrix $\boldsymbol{T}_{\mathrm{b}\to\mathrm{e}}^{-1}$ with the inverse rotations has to be applied to transform $\boldsymbol{\chi}^{\mathrm{e}}$ to $\boldsymbol{\chi}^{\mathrm{b}}$ because the wing or body was rotated by the ordinary (not-inverse) transformation matrix $\boldsymbol{T}_{\mathrm{b}\to\mathrm{e}}$ to its orientation. This can be visualized by rotating the wing only by an angle γ_{e} around $\boldsymbol{z}_{\mathrm{e}}$. From the view of the wing or body fixed coordinate system, the x axis of the earth fixed coordinate system $\boldsymbol{x}_{\mathrm{e}}$ is then rotated by $-\gamma_{\mathrm{e}}$ around the $\boldsymbol{z}_{\mathrm{e}}$ axis. So if $\boldsymbol{\chi}^{\mathrm{e}} = \boldsymbol{x}_{\mathrm{e}}^{\mathrm{e}}$, then $\boldsymbol{\chi}^{\mathrm{b}}$ would be determined by $\boldsymbol{\chi}^{\mathrm{b}} = \boldsymbol{R}_{\mathrm{z_e}}(-\gamma_{\mathrm{e}})\, \boldsymbol{\chi}^{\mathrm{e}}$ in that example.

In conclusion, the aerodynamic force in the earth fixed coordinate system is given by

$$\begin{aligned}\boldsymbol{F}_{\mathrm{ae}}^{\mathrm{e}} &= \boldsymbol{T}_{\mathrm{b}\to\mathrm{e}}\, \boldsymbol{F}_{\mathrm{ae}}^{\mathrm{b}} \\ &= \boldsymbol{T}_{\mathrm{b}\to\mathrm{e}}\, \left(\boldsymbol{F}_{\mathrm{ae,d}}^{\mathrm{b}} + \boldsymbol{F}_{\mathrm{ae,l}}^{\mathrm{b}}\right) \\ &= \boldsymbol{T}_{\mathrm{b}\to\mathrm{e}}\, \left(\frac{1}{2}\rho|\boldsymbol{v}_{\mathrm{r}}^{\mathrm{b}}|^2\, A\, c_{\mathrm{d}}(\alpha)\, \operatorname{dir}\boldsymbol{v}_{\mathrm{r}}^{\mathrm{b}} + \frac{1}{2}\rho|\boldsymbol{v}_{\mathrm{r}}^{\mathrm{b}}|^2\, A\, c_{\mathrm{l}}(\alpha)\, \operatorname{dir}\left(\boldsymbol{v}_{\mathrm{r}}^{\mathrm{b}} \times \boldsymbol{y}_{\mathrm{b}}^{\mathrm{b}}\right)\right) \end{aligned} \tag{4.4}$$

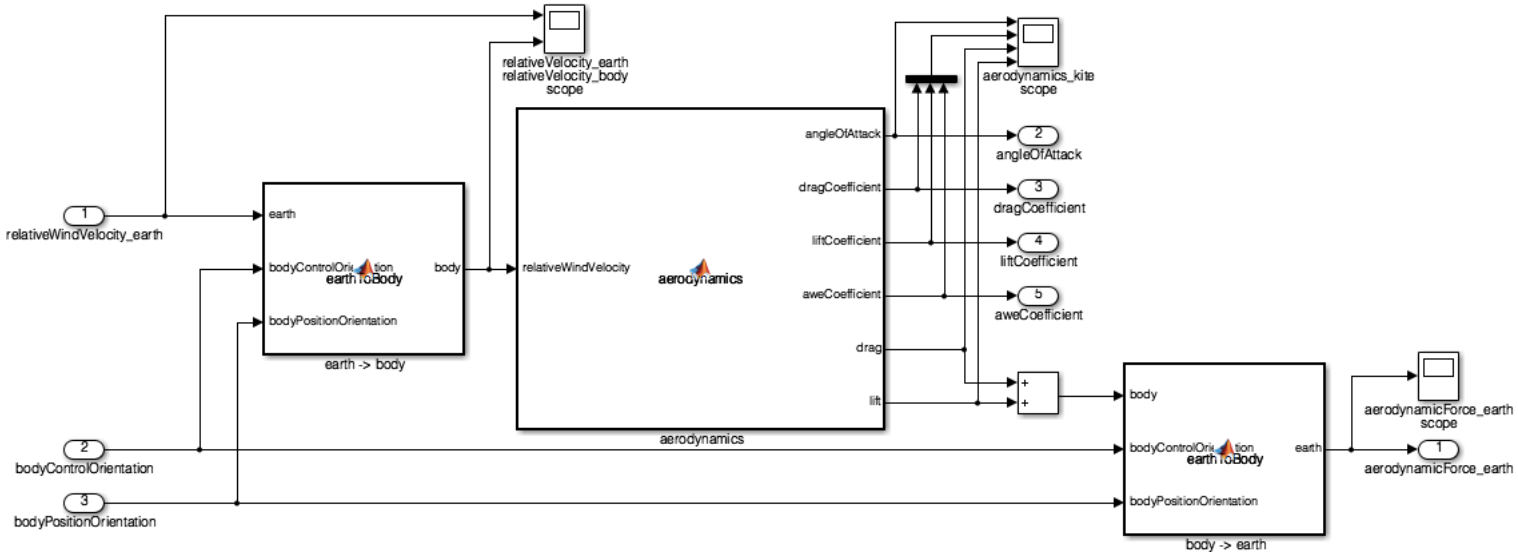


Figure 4.6: Screenshot of the "aerodynamics" subsystem in Simulink.

with

$$\alpha = 90\,^\circ - \arccos\left(\frac{\boldsymbol{v}_\mathrm{r}^\mathrm{b}}{|\boldsymbol{v}_\mathrm{r}^\mathrm{b}|} - \boldsymbol{z}_\mathrm{b}^\mathrm{b}\right) \tag{4.5}$$

and

$$\begin{aligned}\boldsymbol{v}_\mathrm{r}^\mathrm{b} &= \boldsymbol{T}_{\mathrm{e}\to\mathrm{b}}\,\boldsymbol{v}_\mathrm{r}^\mathrm{e} \\ &= \boldsymbol{T}_{\mathrm{e}\to\mathrm{b}}\,(\boldsymbol{v}_\mathrm{w}^\mathrm{e} - \boldsymbol{v}_\mathrm{b}^\mathrm{e})\,.\end{aligned} \tag{4.6}$$

A screenshot of the "aerodynamics" subsystem in Simulink is shown in Fig. 4.6. The aerodynamic force is calculated in the body fixed coordinate system. For this, the relative wind velocity $\boldsymbol{v}_\mathrm{r}^\mathrm{e} = \boldsymbol{v}_\mathrm{w}^\mathrm{e} - \boldsymbol{v}_\mathrm{b}^\mathrm{e}$ in the earth fixed coordinate system is transformed by the Matlab block function "earth -> body" whose code is shown in Lst. 4.2.

```
function body = earthToBody(earth, bodyControlOrientation, bodyPositionOrientation)

body = earth;
orientations = [bodyPositionOrientation, bodyControlOrientation];

for i = 1 : size(orientations, 2)
    % get angles
    alpha = orientations(1, i);
    beta = orientations(2, i);
    gamma = orientations(3, i);

    % rotation matrices
    R_x = [1, 0, 0;
           0, cos(alpha / 360 * (2 * pi)), -sin(alpha / 360 * (2 * pi));
           0, sin(alpha / 360 * (2 * pi)), cos(alpha / 360 * (2 * pi))];
    R_y = [cos(beta / 360 * (2 * pi)), 0, sin(beta / 360 * (2 * pi));
           0, 1, 0;
           -sin(beta / 360 * (2 * pi)), 0, cos(beta / 360 * (2 * pi))];
    R_z = [cos(gamma / 360 * (2 * pi)), -sin(gamma / 360 * (2 * pi)), 0;
           sin(gamma / 360 * (2 * pi)), cos(gamma / 360 * (2 * pi)), 0;
           0, 0, 1];

    body = R_x^-1 * R_y^-1 * R_z^-1 * body;
end
```

Listing 4.2: Source code of the Matlab block function "earth -> body".

The whole calculation of the aerodynamic force was also implemented inside a Matlab block function because it was faster to develop and to debug. The source code is shown in Lst. 4.3.

```
function [angleOfAttack, dragCoefficient, liftCoefficient, aweCoefficient, drag, lift]
    ↪ = aerodynamics(relativeWindVelocity)
%% alpha
angleOfAttack = 90 - acos(relativeWindVelocity' / norm(relativeWindVelocity) *
    ↪ -[0;0;-1]) / (2 * pi) * 360;
if angleOfAttack < -180
    angleOfAttack = angleOfAttack + 360;
elseif angleOfAttack > 180
    angleOfAttack = angleOfAttack - 360;
end


%% drag coefficient
dragCefficient_anchorPoints = [-180 -90 0 90 180];
```

```
dragCoefficient_splineCoefficients = [-1.78326474622771e-06 0.000222222222222222 0
    ↪ 0.500000000000000;2.05761316872428e-06 -0.000259259259259259
    ↪ -0.00333333333333333 1;-2.05761316872428e-06 0.000296296296296296 0
    ↪ 0.100000000000000;1.78326474622771e-06 -0.000259259259259259 0.00333333333333333
    ↪  1;];

dragCoefficient = 0;
for splineAlphaIndex = 1 : length(dragCefficient_anchorPoints)
    if dragCefficient_anchorPoints(splineAlphaIndex) <= angleOfAttack && angleOfAttack
        ↪ <= dragCefficient_anchorPoints(splineAlphaIndex + 1)
        dragCoefficient = dragCoefficient_splineCoefficients(splineAlphaIndex, 1) * (
            ↪ angleOfAttack - dragCefficient_anchorPoints(splineAlphaIndex))^3 ...
            + dragCoefficient_splineCoefficients(splineAlphaIndex, 2) * (angleOfAttack
                ↪ - dragCefficient_anchorPoints(splineAlphaIndex))^2 ...
            + dragCoefficient_splineCoefficients(splineAlphaIndex, 3) * (angleOfAttack
                ↪ - dragCefficient_anchorPoints(splineAlphaIndex))^1 ...
            + dragCoefficient_splineCoefficients(splineAlphaIndex, 4) * (angleOfAttack
                ↪ - dragCefficient_anchorPoints(splineAlphaIndex))^0;
        break;
    end
end


%% lift coefficient
liftCoefficient_anchorPoints = [-180 -140 -60 0 25 140 180];
liftCoefficient_splineCoefficients = [-1.07943755498536e-05 0.000806775021994144 0
    ↪ 0;2.75078193731700e-06 -0.000488550043988288 0.0127289991202342
    ↪ 0.600000000000000;2.03492707314944e-06 0.000171637620967791 -0.0126239947214055
    ↪ -0.100000000000000;-1.82365512609771e-05 0.000537924494134691 0.0299497321847434
    ↪  0.200000000000000;4.45092408658084e-06 -0.000829816850438595 0.0226524232771458
    ↪  1;-1.35093994928975e-05 0.000705751959431797 0.00838496081136407
    ↪ -0.600000000000000;];

liftCoefficient = 0;
for splineAlphaIndex = 1 : length(liftCoefficient_anchorPoints)
    if liftCoefficient_anchorPoints(splineAlphaIndex) <= angleOfAttack && angleOfAttack
        ↪  <= liftCoefficient_anchorPoints(splineAlphaIndex + 1)
        liftCoefficient = liftCoefficient_splineCoefficients(splineAlphaIndex, 1) * (
            ↪ angleOfAttack - liftCoefficient_anchorPoints(splineAlphaIndex))^3 ...
            + liftCoefficient_splineCoefficients(splineAlphaIndex, 2) * (angleOfAttack
                ↪ - liftCoefficient_anchorPoints(splineAlphaIndex))^2 ...
            + liftCoefficient_splineCoefficients(splineAlphaIndex, 3) * (angleOfAttack
                ↪ - liftCoefficient_anchorPoints(splineAlphaIndex))^1 ...
            + liftCoefficient_splineCoefficients(splineAlphaIndex, 4) * (angleOfAttack
                ↪ - liftCoefficient_anchorPoints(splineAlphaIndex))^0;
        break;
    end
end


%% AWE coefficient
aweCoefficient = sqrt(liftCoefficient^2 + dragCoefficient^2)^3 / dragCoefficient^2;


%% forces
% parameter
density = 1.2;
area = 10;

% drag
dragDirection = relativeWindVelocity / norm(relativeWindVelocity);
```

```
drag = 0.5 * density * norm(relativeWindVelocity)^2 * area * dragCoefficient *
    ↪ dragDirection;

% lift
liftDirection = cross(relativeWindVelocity, [0; 1; 0]);
if norm(liftDirection) ~= 0
    liftDirection = liftDirection / norm(liftDirection);
    lift = 0.5 * density * norm(relativeWindVelocity)^2 * area * liftCoefficient *
        ↪ liftDirection;
else
    lift = [0; 0; 0];
end
```

Listing 4.3: Source code of the Matlab block function "aerodynamics".

For the export to C++, the Matlab functions *spline()* and *ppval()* to evaluate a certain point of the spline polynomials are not available. Hence, the offline calculated spline coefficients are stored in arrays and the correct spline is chosen and evaluated for a given angle of attack α. This block outputs the most important aerodynamic signals which are connected to scopes for debugging and to outports for logging purposes. Here, *aweCoefficient* means

$$c_{\mathrm{AWE}} := \frac{c_{\mathrm{ae}}^3}{c_{\mathrm{d,i}}^2} = \frac{\left(\sqrt{\left(c_{\mathrm{d,i}} + c_{\mathrm{d,t}}\right)^2 + c_{\mathrm{l}}^2}\right)^3}{c_{\mathrm{d,i}}^2} \overset{c_{\mathrm{d,t}}=0}{=} \frac{\left(\sqrt{c_{\mathrm{d,i}}^2 + c_{\mathrm{l}}^2}\right)^3}{c_{\mathrm{d,i}}^2} \tag{4.7}$$

which is an important factor of the maximum extracted power of AWE in Eq. (2.11) on p. 21.

The total aerodynamic force in the body fixed coordinate system is finally transformed to the earth fixed coordinate system by the Matlab block function "body -> earth" whose code is shown in Lst. 4.4.

```
function earth = earthToBody(body, bodyControlOrientation, bodyPositionOrientation)

earth = body;
orientations = [bodyControlOrientation, bodyPositionOrientation];

for i = 1 : size(orientations, 2)
    % get angles
    alpha = orientations(1, i);
    beta = orientations(2, i);
    gamma = orientations(3, i);

    % rotation matrices
    R_x = [1, 0, 0;
           0, cos(alpha / 360 * (2 * pi)), -sin(alpha / 360 * (2 * pi));
           0, sin(alpha / 360 * (2 * pi)), cos(alpha / 360 * (2 * pi))];
    R_y = [cos(beta / 360 * (2 * pi)), 0, sin(beta / 360 * (2 * pi));
           0, 1, 0;
           -sin(beta / 360 * (2 * pi)), 0, cos(beta / 360 * (2 * pi))];
    R_z = [cos(gamma / 360 * (2 * pi)), -sin(gamma / 360 * (2 * pi)), 0;
           sin(gamma / 360 * (2 * pi)), cos(gamma / 360 * (2 * pi)), 0;
           0, 0, 1];

    earth = R_z * R_y * R_x * earth;
end
```

Listing 4.4: Source code of the Matlab block function "body -> earth".

The composition of the vectors with the rotation angles *bodyControlOrientation* and *bodyPositionOrientation* are implemented outside the "aerodynamics" subsystem and shown later in the assembled Simulink model.

4.1.3 Tether model

The tether is modeled as ideal spring, i.e. the spring is massless, volume less and lossless. A one dimensional spring exerts a force F_t (index t for tether) against its deformation Δl_t according to Hook's law with

$$F_\mathrm{t} = c_\mathrm{t}\,\Delta l_\mathrm{t}, \tag{4.8}$$

where $c_\mathrm{t} \in \mathbb{R}_{>0}\,[\mathrm{N/m}]$ is the spring constant. The deformation or length difference under those ideal assumptions is the difference of the actual length of the tether l_t and the norm or length of the wing's position vector $\boldsymbol{r}_\mathrm{b}$, i.e.

$$\Delta l_\mathrm{t} = l_\mathrm{t} - |\boldsymbol{r}_\mathrm{b}|. \tag{4.9}$$

Unlike a spring, a tether can only exert negative forces in regard to Eq. (4.8) and (4.9). Hence, the magnitude of the force is given by

$$F_\mathrm{t} = \begin{cases} c_\mathrm{t}\,(l_\mathrm{t} - |\boldsymbol{r}_\mathrm{b}|) & \text{for } c_\mathrm{t}\,(l_\mathrm{t} - |\boldsymbol{r}_\mathrm{b}|) < 0 \\ 0 & \text{else.} \end{cases} \tag{4.10}$$

This force is only a 1D vector or a scalar. The 3D tether force vector always points from the wing to the tether escape point on the ground. So Eq. (4.10) can be seen as the x coordinate of the 3D tether force vector that is described in a cartesian coordinate system which is rotated by the elevation and azimuth angles, β_e and γ_e. Here, this coordinate system is defined as tether coordinate system denoted by a superscript $^\mathrm{t}$. So, the tether force vector in the tether coordinate system is given by

$$\boldsymbol{F}_\mathrm{t}^\mathrm{t} = \begin{cases} \begin{pmatrix} c_\mathrm{t}\,(l_\mathrm{t} - |\boldsymbol{r}_\mathrm{b}|) \\ 0 \\ 0 \end{pmatrix} & \text{for } c_\mathrm{t}\,(l_\mathrm{t} - |\boldsymbol{r}_\mathrm{b}|) < 0 \\ \boldsymbol{0} & \text{else.} \end{cases}$$

Finally, the force has to be transformed into the earth fixed coordinate system by rotating it by β_e and γ_e using the transformation

$$\boldsymbol{T}_{\mathrm{t}\to\mathrm{e}} := \boldsymbol{R}_{\mathrm{z_e}}(\gamma_\mathrm{e})\boldsymbol{R}_{\mathrm{y_e}}(\beta_\mathrm{e}).$$

Altogether, the tether force in the earth fixed coordinate system is given by

$$\begin{aligned} \boldsymbol{F}_\mathrm{t}^\mathrm{e} &= \boldsymbol{T}_{\mathrm{t}\to\mathrm{e}}\,\boldsymbol{F}_\mathrm{t}^\mathrm{t} \\ &= \boldsymbol{R}_{\mathrm{z_e}}(\gamma_\mathrm{e})\boldsymbol{R}_{\mathrm{y_e}}(\beta_\mathrm{e}) \begin{cases} \begin{pmatrix} c_\mathrm{t}\,(l_\mathrm{t} - |\boldsymbol{r}_\mathrm{b}|) \\ 0 \\ 0 \end{pmatrix} & \text{for } c_\mathrm{t}\,(l_\mathrm{t} - |\boldsymbol{r}_\mathrm{b}|) < 0 \\ \boldsymbol{0} & \text{else.} \end{cases} \end{aligned} \tag{4.11}$$

A screenshot of the in Simulink implemented "tether" subsystem is shown in Fig. 4.7. The angles β_e and γ_e are calculated via a transformation of the position vector $\boldsymbol{r}_\mathrm{b}^\mathrm{e}$ of the earth fixed cartesian coordinate system to the earth fixed spherical coordinate system. Both transformations are again implemented in a Matlab function block whose sources are shown in Lsts. 4.5 and 4.6.

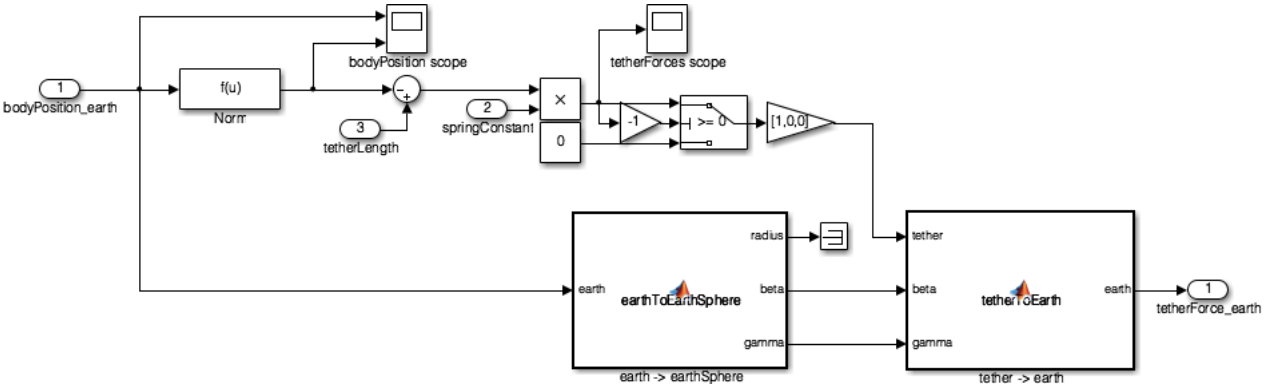


Figure 4.7: Screenshot of the "tether" subsystem in Simulink.

```
1  function [radius, beta, gamma] = earthToEarthSphere(earth)
2  % general
3  x = earth(1);
4  y = earth(2);
5  z = earth(3);
6
7  % radius and angels
8  radius = norm(earth);
9  beta = -atan2(z, sqrt(x^2 + y^2)) / (2 * pi) * 360;
10 gamma = atan2(y, x) / (2 * pi) * 360;
```

Listing 4.5: Source code of the Matlab block function "earth -> earthSphere".

```
1  function earth = tetherToEarth(tether, beta, gamma)
2
3  R_y = [cos(beta/360 * 2 * pi), 0, sin(beta/360 * 2 * pi);
4         0, 1, 0
5         -sin(beta/360 * 2 * pi), 0, cos(beta/360 * 2 * pi)];
6  R_z = [cos(gamma/360 * 2 * pi), -sin(gamma/360 * 2 * pi), 0;
7         sin(gamma/360 * 2 * pi), cos(gamma/360 * 2 * pi), 0;
8         0, 0, 1];
9
10 earth = R_z * R_y * tether;
```

Listing 4.6: Source code of the Matlab block function "tether -> earth".

4.1.4 Assembled lift AWE model

A screenshot of the model root of the assembled lift AWE plant model in Simulink is shown in Fig. 4.8. The four main blocks in light gray, "gravitation", "mass point dynamics", "aerodynamics" and "tether", where treated in detail in the previous sections. The properties air density and area of the wing are already defined in Lst. 4.3 on p. 49 in line 49 and 50 with the values $1.2\,\mathrm{kg/m^3}$ and $10\,\mathrm{m^2}$. A constant mass of the wing of $5\,\mathrm{kg}$ is entered as input for the gravitational force and for the mass point dynamics. There are also initial positions and initial velocities in constant blocks entered. All the forces are connected to the "mass point dynamics" subsystem block. Left to the "aerodynamics" subsystem block, the relative wind velocity is calculated. A constant wind velocity vector, which models a uniform wind field, is entered in the corresponding constant block. The control pitch and yaw angular speeds are chosen as inputs. They are integrated over time to angular positions and then assembled to a vector of control

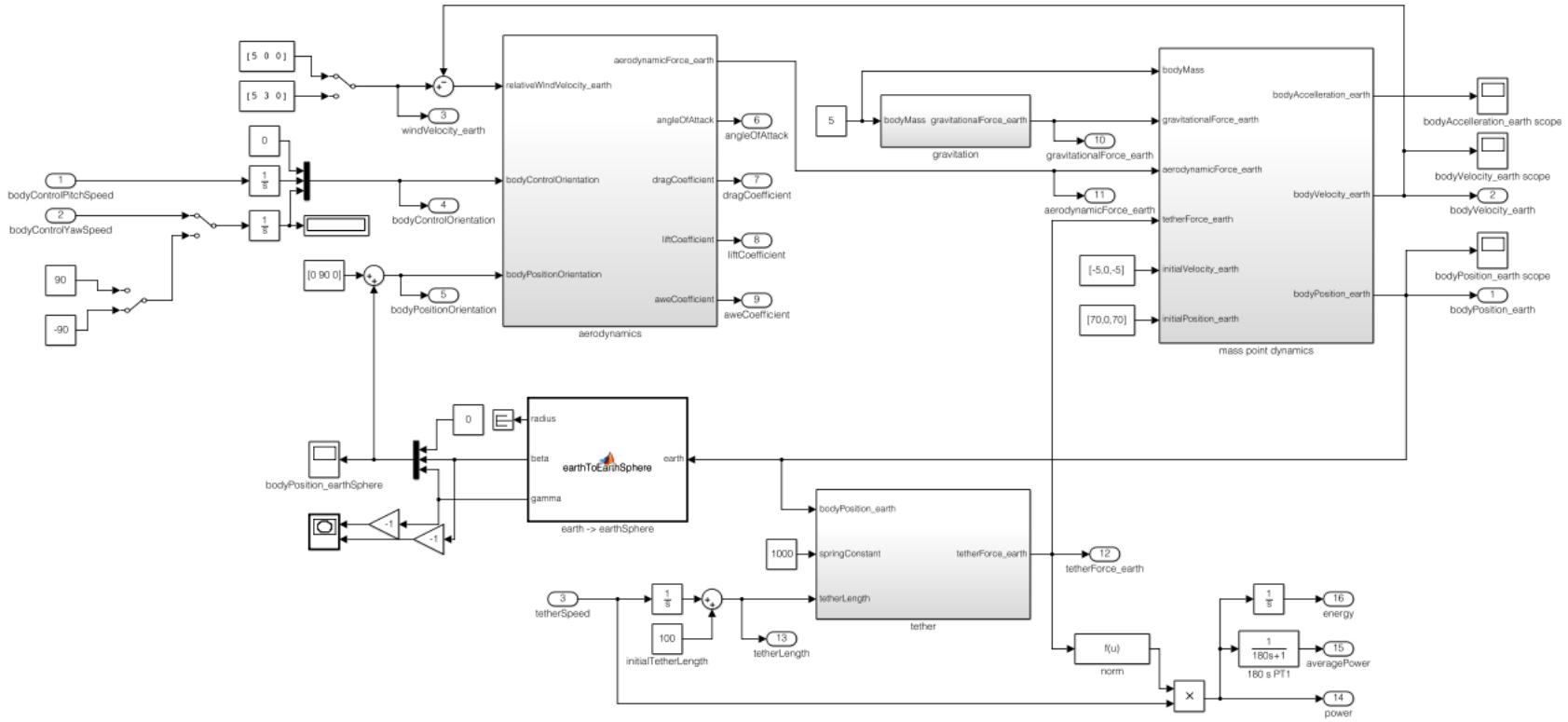


Figure 4.8: Screenshot of the model root of the assembled lift AWE plant model in Simulink.

orientations. This vector is connected to the "aerodynamics" subsystem block. The position orientation is calculated via the elevation and azimuth angles of the body's position vector. As described above, an offset of 90 ° is added to the elevation angle. The "tether" has a fictitious spring constant of 1000 N/m. The tether speed is chosen as third input variable. The integral of the tether speed over time is added to an initial tether length and connected to the "tether" block. Right to the "tether" block, the mechanical power is calculated as the multiplication of the tether speed and the norm of the tether force. A first order low pass filter with a time constant of 180 s is used to calculate an average power. The energy is calculated as the integral of the power over time. The most important signals are connected to outports for a visualization in the QtPLC Control Center.

4.1.5 Simulink export to C++

In this section, the procedure is described to export the Simulink model to C++. It was tested successfully on Mac OS X but works similar under Linux and Windows.

Before exporting the model to C++, Matlab's current directory was set to that one where Simulink should place the exported files. Then, "Configuration" and "Model Configuration Parameters" on the Simulink's menubar were clicked which opened the window depicted in Fig. 4.9. On this first "Solver" page of the configuration window, a fixed step solver was chosen.

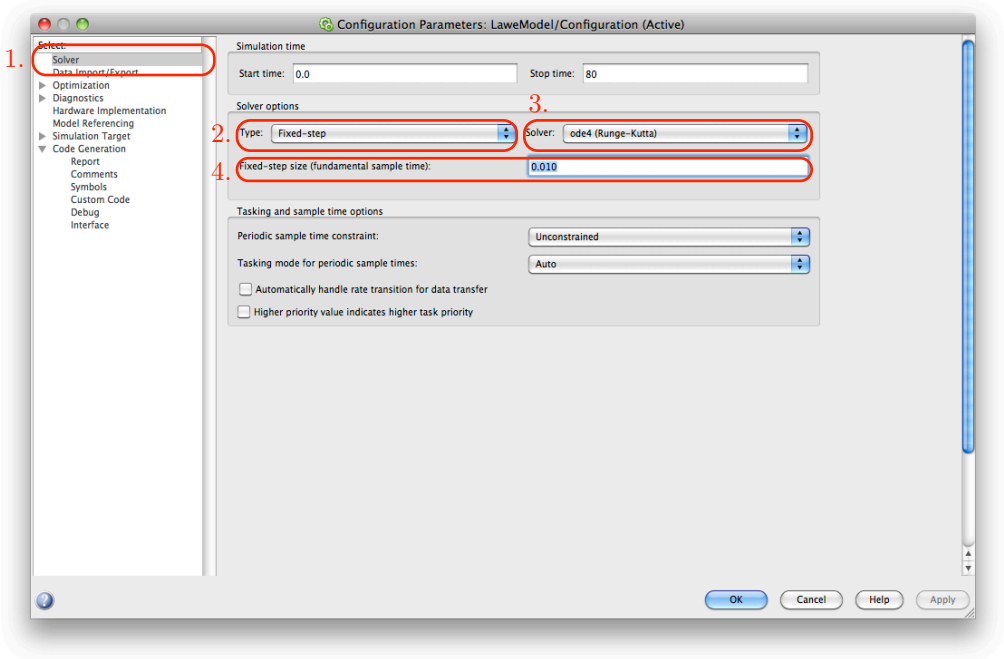


Figure 4.9: Screenshot of the Simulink "Solver" settings.

In the tests "ode4 (Runge-Kutta)" worked well. The fundamental step size was set to 0.010 s.

On the page "Code Generation"/"Interface", shown in Fig. 4.10, the "MAT-file logging" option was unselected. The main reason for this is that Simulink then does not create a variable with the name "signals". This would have produced compile errors later, because it is a keyword in Qt (it is already assigned by a *#define*).

On the page "Code Generation", shown in Fig. 4.11, the "Generic Real-Time Target" system

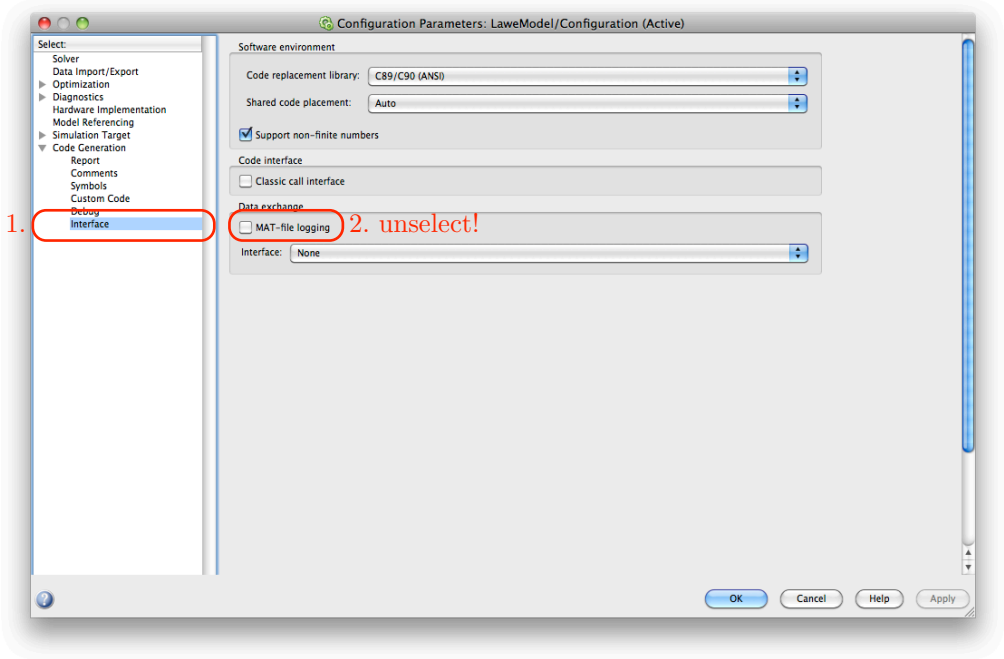


Figure 4.10: Screenshot of the Simulink "Code Generation"/"Interface" settings.

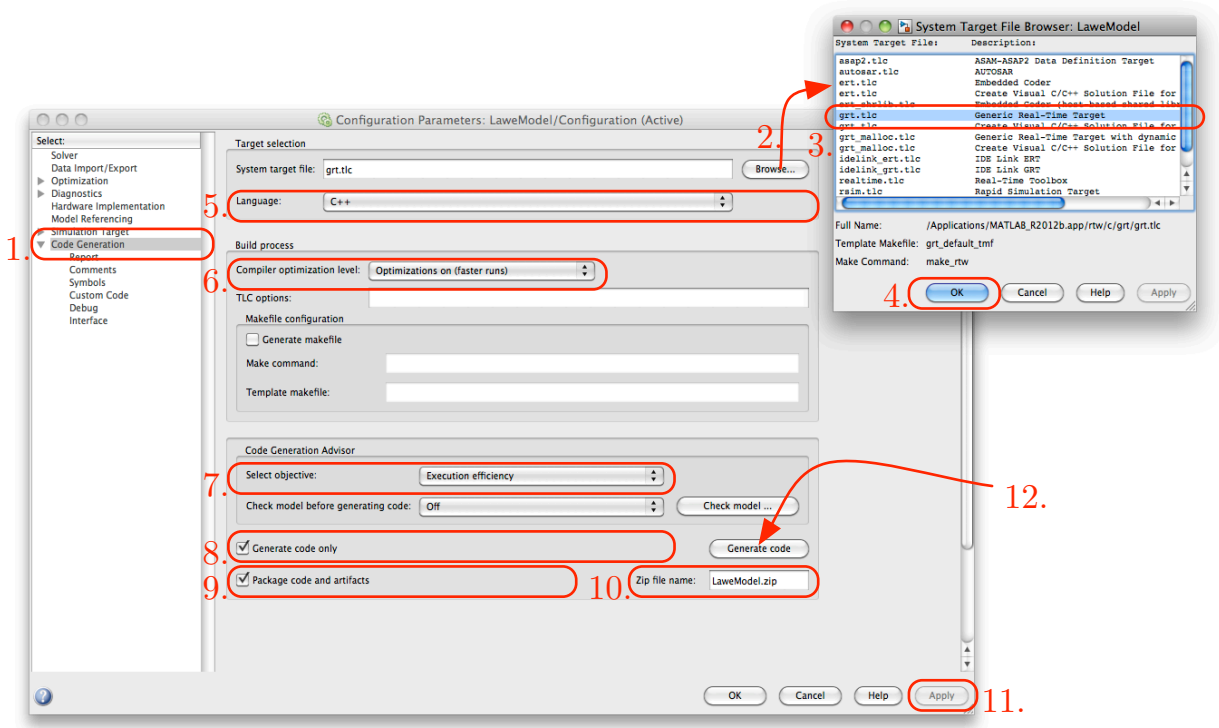


Figure 4.11: Screenshot of the Simulink "Code Generation" settings.

target file was selected after clicking on “Browse...”. It was important to select “C++” as language (step 5) so that Simulink creates “*.cpp” files although the exported code is C. But the later used GCC[28] compiler uses the ending to decide how to compile a file. If the files were compiled with pure C, then the C functions of that files could not be called as expected, because the function identifiers in Assembly from a C++ compilation differ from those of a C compilation. Step 6 and 7 were optional to achieve a faster code execution. “Generate code only” was checked (step 8), because the code will be added to the Qt/QtPLC project later. It was very helpful to select “Package code and artifacts” and enter a “Zip file name”, in this case “LaweModel.zip” (steps 9 and 10). If these last two steps were not performed, many general Matlab and Simulink header files where general datatypes and solvers are specified would have been missed. Finally, with the click on the buttons “Apply” and “Generate code”, all needed files are generated and packed in the “LaweModel.zip” file which is placed in Matlab’s current working directory. The decompressed ZIP file is the “LaweModel” directory with subdirectories, “*.h” and “*.cpp” files. This directory is later added to the Qt project.

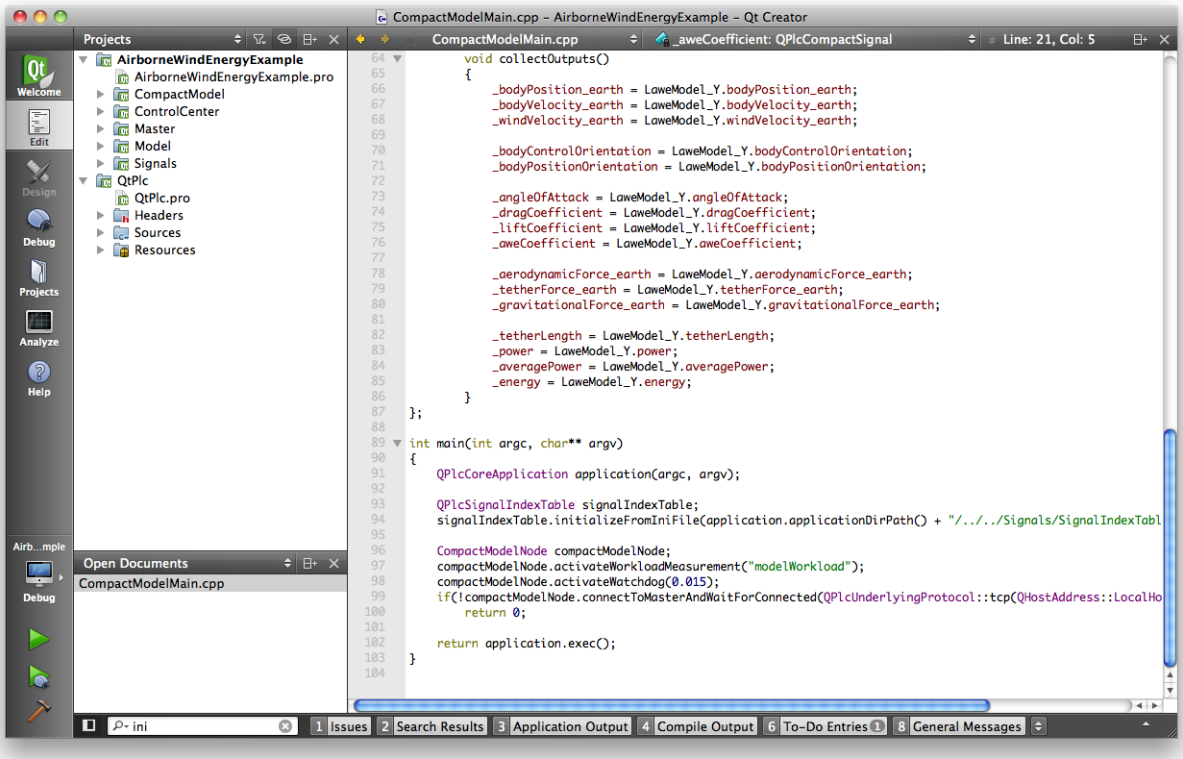


Figure 4.12: Qt Creator with the opened “AirborneWindEnergyExample” and “QtPlc” projects.

4.2 Implementation of the example lift AWE applications

The example lift AWE project has three applications, i.e. three processes:

- the master command line application,

[28] GNU Compiler Collection.

- the model command line application where the exported C++ files from Matlab/Simulink are utilized and
- the QtPLC Control Center GUI application.

Two solutions are presented for the model, a "normal" and a "compact" version, but only one is executed for the simulation. Fig. 4.12 gives an impression of the Qt Creator IDE where both, the final "AirborneWindEnergyExample" project and the "QtPlc" library project, are opened. The Qt version 4.8.5 was used.

The whole section may be read like a QtPLC tutorial, but detailed background information is given. Again, several source codes are presented, but not every single line is explained in detail. It is assumed that the reader has good knowledge in C++ and Qt.

4.2.1 General directory and project structure

Fig. 4.13 gives an overview of the directory structure. "AirborneWindEnergyExample" contains

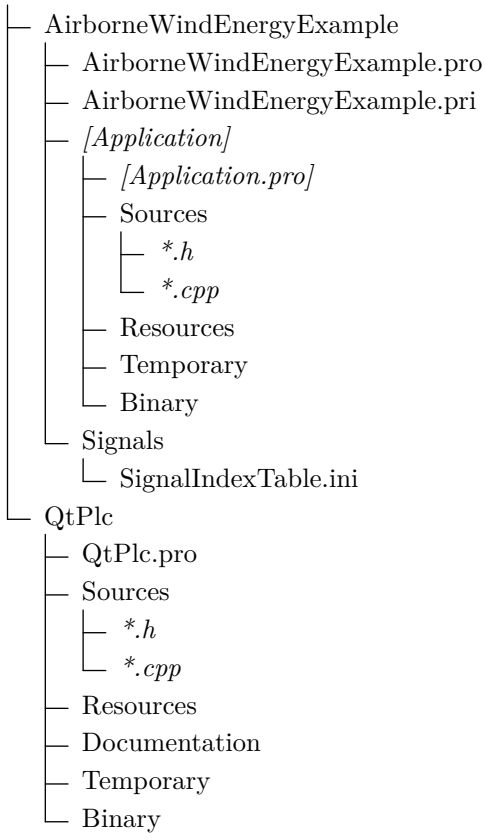


Figure 4.13: Overview of the directory structure of the "AirborneWindEnergyExample" and "QtPlc" projects.

the sources and the binaries of the applications and "QtPlc" contains the sources and the binary of the QtPLC library. The "AirborneWindEnergyExample" directory is divided into directories for the individual applications, i.e. the directories "Master", "Model", "CompactModel" and

"ControlCenter". In the additional "Signals" directory only the "SignalIndexTable.ini" file is stored. In this INI file all global signals are defined that can be exchanged between the processes. Any process can chose any of these signals as inputs and as outputs. This is similar to control engineering tools like Simulink where the different blocks are connected with arrows. In QtPLC, the blocks are the slave processes and the arrows are the signals. These signal connections are managed by the slave processes themselves by informing the master about their input and output signals, see also Fig. 3.2 on p. 29. The content of "SignalIndexTable.ini" is shown in Lst. 4.7.

```
;NETWORK NODES

[masterTime]
type = DateTime

[masterJitter]
type = Real

[masterWorkload]
type = Real

[modelWorkload]
type = Real



;CONTROL INPUTS

[bodyControlPitchSpeed]
type = Real

[bodyControlYawSpeed]
type = Real

[tetherSpeed]
type = Real



;STATES

[bodyPosition_earth]
type = RealTriple

[bodyVelocity_earth]
type = RealTriple

[windVelocity_earth]
type = RealTriple

[bodyControlOrientation]
type = RealTriple

[bodyPositionOrientation]
type = RealTriple

[angleOfAttack]
type = Real

[dragCoefficient]
type = Real

[liftCoefficient]
```

```
54 type = Real
55
56 [aweCoefficient]
57 type = Real
58
59 [gravitationalForce_earth]
60 type = RealTriple
61
62 [aerodynamicForce_earth]
63 type = RealTriple
64
65 [tetherForce_earth]
66 type = RealTriple
67
68 [tetherLength]
69 type = Real
70
71 [power]
72 type = Real
73
74 [averagePower]
75 type = Real
76
77 [energy]
78 type = Real
```

Listing 4.7: "SignalIndexTable.ini".

Here, the global signals for this project are defined. A signal is defined with its name in square brackets. Below that, a mandatory type has to be chosen which must be one of

- *Int8*,
- *Bool* (= *Int8*),
- *Int16*,
- *Int32*,
- *Int* (= *Int32*),
- *Int64*,
- *UInt8*,
- *Byte* (= *UInt8*),
- *UInt16*,
- *UInt32*,
- *UInt* (= *UInt32*),
- *UInt64*,
- *Real32*,
- *Real* (= *Real32*),
- *Real64*,
- *Real32Triple*,

- *RealTriple* (= *Real32Triple*),
- *Real64Triple*,
- *String* or
- *DateTime*.

In QtPLC a signal value of such a type is stored in a *QPlcValue* object which is like a *union* for these types and is thus comparable to *QVariant*. Additionally, a *QPlcValue* of any type can be converted to and from a *QByteArray*. For the primitive types like integer and floating point, these conversions are implemented with the correct byte order with the *union QSplitPrimitive* of the QtPLC library

The main project file "AirborneWindEnergyExample.pro" is shown in Lst. 4.8 and only includes all subdirectories.

```
1 TEMPLATE = subdirs
2 SUBDIRS = $$system(find . -type d -d 1)
```

Listing 4.8: "AirborneWindEnergyExample.pro".

Each application (directory) has its own project file and contains the following directories, see Fig. 4.13 on p. 58:

- the "Sources" directory in which all "*.h" and "*.cpp" files are located, optionally also inside subdirectories,
- optionally the "Resources" directory in which a Qt resource file and the actual resources are located,
- the "Temporary" directory in which temporary build files are located and
- the "Binary" directory where the final binary file is located.

The same directory structure is used for the "QtPLC" library project. Here, an additional "Documentation" directory contains the with *Doxygen*[29] generated documentation. The "Sources" directory of "QtPLC" further contains a "Core" and "Gui" directory to group the QtPLC classes. Additionally, in that "Sources" directory is the file "QtPlc" which is a header file without ".h" ending. This header file includes all header files of the QtPLC library. So in an QtPLC application, just *#include <QtPlc>* needs to be added to include the whole QtPLC library.

The project file of each application of the "AirborneWindEnergyExample" project is sparse. As an example, the content of the "Master.pro" project file is shown in Lst. 4.9.

```
1 TARGET = Master
2 CONFIG += Console
3
4 PRO = $$PWD
5 include(../AirborneWindEnergyExample.pri)
```

Listing 4.9: "Master.pro".

Here, only the *TARGET* is set and the application is configured to be a *Console* application. Then the project file directory path is stored in the *PRO* variable in line 4 and the "AirborneWindEnergyExample.pri" file is included which performs all tasks that are identical in each application's project. The content of this file is shown in Lst. 4.10.

[29] Van Heesch, Dimitri: "Doxygen". http://doxygen.org, accessed: December 06, 2013.

```
#HEADERS AND SOURCES

exists($$PRO/Sources) {
    SOURCEDIRS += $$system(find $$PRO/Sources -type d)
    SOURCES += $$system(find $$PRO/Sources -name '*.cpp')
    HEADERS += $$system(find $$PRO/Sources -name '*.h')
}
exists($$PRO/Resources): RESOURCES += $$system(find $$PRO/Resources -name '*.qrc')

INCLUDEPATH = $$SOURCEDIRS
DEPENDPATH = $$SOURCEDIRS



#QT CONFIGURATION

Console {
    CONFIG += console
    CONFIG -= app_bundle
}
TEMPLATE = app



#LIBRARIES

QT += core network gui opengl

QMAKE_CXXFLAGS += -std=c++11

CONFIG += staticLinkedQtPlc

staticLinkedQtPlc {
    INCLUDEPATH += ../../QtPlc/Sources \
        ../../QtPlc/Sources/Core \
        ../../QtPlc/Sources/Gui
    LIBS += -L../../QtPlc/Binary -lQtPlc
    #POST_TARGETDEPS += ../../QtPlc/Binary/libQtPlc.a
} else {
    SOURCEDIRS += $$system(find $$PRO/../../QtPlc/Sources -type d)
    SOURCES += $$system(find $$PRO/../../QtPlc/Sources -name '*.cpp')
    HEADERS += $$system(find $$PRO/../../QtPlc/Sources -name '*.h')
    exists($$PRO/Resources): RESOURCES += $$system(find $$PRO/../../QtPlc/Resources -
        ↪ name '*.qrc')

    INCLUDEPATH = $$SOURCEDIRS
    DEPENDPATH = $$SOURCEDIRS
}



#DIRECTORIES

DESTDIR = ./Binary

unix: COMILINGDIR = ./Temporary/Unix
macx: COMILINGDIR = ./Temporary/Mac
win32: COMILINGDIR = ./Temporary/Win
MOC_DIR = $$COMILINGDIR
OBJECTS_DIR = $$COMILINGDIR
UI_DIR = $$COMILINGDIR
```

```
61 RCC_DIR = $$COMILINGDIR
```

Listing 4.10: "AirborneWindEnergyExample.pri".

The headers and sources are not included in the standard Qt project file way where every single file would have been listed with its path. Instead, in lines 1 . . . 11 all directories, "*.h" and "*.cpp" files in the "Sources" directory and its subdirectories are fetched by a system call. The same procedure is used for the "*.qrc" resource file in the "Resources" directory. The advantage of this approach is that anything in "Sources" is added automatically to the project. This is very beneficial for the bunch of "*.h" and "*.cpp" files from the exported Simulink model whose uncompressed ZIP file just needs to be dropped in the "Sources" directory. However, the disadvantage is that the used system call is not platform independent. In lines 15 . . . 17 the Qt application type is set, in line 27 the used Qt modules are included and in line 29 the C++11 flag is added to the compiler flags. The whole QtPLC library uses C++11 features. So if this flag was not set, compiler errors would appear. In lines 31 . . . 47 the QtPLC library is included: Either the binary library file is linked statically or all sources and headers of QtPLC are added directly to the project. The latter can be obtained by commenting out line 31 and was useful for debugging the QtPLC classes. In the last lines 55 . . . 61 the output directories are defined.

The project file of the QtPLC library looks quite similar, but here no additional "*.pri" file is used. The content of "QtPlc.pro" is shown in Lst. 4.11.

```
1  #GENERAL CONFIGURATION
2
3  #Logging Policy
4  CONFIG += AbnormalLogging
5  #CONFIG += DebugLogging
6
7
8
9  #HEADERS AND SOURCES
10
11 PRO = $$PWD
12
13 SOURCEDIRS = $$system(find $$PRO/Sources -type d)
14 SOURCES += $$system(find $$PRO/Sources -name '*.cpp')
15 HEADERS += $$system(find $$PRO/Sources -name '*.h')
16 exists($$PRO/Resources): RESOURCES += $$system(find $$PRO/Resources -name '*.qrc')
17
18 HEADERS += Sources/QtPlc Sources/QtPlcCore Sources/QtPlcGui
19
20 INCLUDEPATH += $$SOURCEDIRS
21 DEPENDPATH += $$SOURCEDIRS
22
23
24
25 #APPLY GLOBAL DEFINITIONS FROM CONFIGURATION
26
27 #Logging Policy
28
29 DEFINES += AbnormalLogging=1
30 DEFINES += DebugLogging=2
31
32 AbnormalLogging: DEFINES += LoggingPolicy=AbnormalLogging
33 DebugLogging: DEFINES += LoggingPolicy=DebugLogging
34
35
36
37 #QT CONFIGURATION
38
```

```
39 TEMPLATE = lib
40 CONFIG += staticlib
41
42
43
44 #LIBRARIES
45
46 QT += core network gui opengl
47
48 QMAKE_CXXFLAGS += -std=c++11
49
50
51
52 #DIRECTORIES
53
54 DESTDIR = ./Binary
55
56 unix: COMILINGDIR = ./Temporary/Unix
57 macx: COMILINGDIR = ./Temporary/Mac
58 win32: COMILINGDIR = ./Temporary/Win
59 MOC_DIR = $$COMILINGDIR
60 OBJECTS_DIR = $$COMILINGDIR
61 UI_DIR = $$COMILINGDIR
62 RCC_DIR = $$COMILINGDIR
```

Listing 4.11: "QtPlc.pro".

For debugging a *CONFIG* flag for the logging policy is set in lines 3 . . . 5 which is later in lines 25 . . . 33 resolved in a *#define*. When setting that flag to *DebugLogging* then many debug messages are written to the console, e.g. the exact bytes of a sent or received message. The "*.h" and "*.cpp" files are included in the same way as above in lines 9 . . . 21, only the headers without "*.h" ending are added manually in line 18. In lines 37 . . . 40 the project is configured to be a static library, in lines 44 . . . 48 the required Qt modules and the C++11 compiler flag are added. In the last lines the output directories are defined.

4.2.2 Master application

The directory "AirborneWindEnergyExample/Master/Sources" contains only the file "MasterMain.cpp" whose content is shown in Lst. 4.12. Since the name of the file which contains the C++ *main()* function is arbitrary, the prefix "Master" has been added for easy distinction. This is also applied for the other applications.

```
1  #include <QtPlc>
2
3  int main(int argc, char** argv)
4  {
5      QPlcCoreApplication application(argc, argv);
6
7      QPlcSignalIndexTable signalIndexTable;
8      signalIndexTable.initializeFromIniFile(application.applicationDirPath() + "/../../
           ↪ Signals/SignalIndexTable.ini");
9
10     QPlcMasterNode master;
11     master.activateWorkloadMeasurement("masterWorkload");
12     master.activateJitterMeasurement("masterJitter");
13     master.setTimeSignal("masterTime");
14     master.setInterval(0.010);
15     if(!master.startServer(QPlcUnderlyingProtocol::tcp(QHostAddress::Any, 50000)))
16         return 0;
17
```

```
18      return application.exec();
19  }
```

Listing 4.12: "MasterMain.cpp".

Only 19 lines are necessary: In the first line the QtPLC headers are included. This also includes all important Qt headers of the Qt modules QtCore, QtNetwork, QtGui and QtOpenGL, whereby the the last two are not needed for this command line application. In line 5 a *QPlcCoreApplication* object is instantiated which is a *QCoreApplication* object where all codecs are set to UTF-8.

In line 7 a *QPlcSignalIndexTable* is instantiated which is needed in every QtPLC class that deals with signals. Here "signal" means the

- signal value and
- the signal name or signal index, respectively,

which are exchanged between the processes. Not the signals from the Qt signals and slots mechanism are meant. Those signals are denoted as "Qt signals" throughout this thesis. *QPlcSignalIndexTable* stores the name of the signals and adds an index to each signal. The *QPlcSignalIndexTable* object is initialized with the "SignalIndexTable.ini" file in line 8. This object is then used statically in many other QtPLC classes.

In line 10 a *QPlcMasterNode* object is instantiated and in the following two lines workload and jitter measurements are activated. The corresponding signal names are passed as argument in which the measurement results are stored. The measured signals from a timeout have a timestamp in form of a time signal. So in line 13, the mandatory time signal name is set. In line 14 the master cycle time is set to 0.010 s. In line 15, the master starts its TCP server on port 50000 and accepts incoming connections from anywhere. Finally, the application is executed, i.e. the event loop is started, in line 18.

4.2.3 Model application

The project file content of the model application is shown in Lst. 4.13 and is almost identical to the project file of the master application in Lst. 4.9.

```
1  TARGET = Model
2  CONFIG += Console
3
4  PRO = $$PWD
5  include(../AirborneWindEnergyExample.pri)
```

Listing 4.13: "Model.pro".

Besides the unpacked ZIP directory "LaweModel" from the C++ export of the Simulink model, only the "ModelMain.cpp" file is placed in the "Sources" directory, whose content is shown in Lst. 4.14.

```
1  #include <QtPlc>
2  #include "LaweModel.h"
3
4  class ModelNode: public QPlcSlaveNode
5  {
6      private:
7          QPlcSignalIndex _bodyControlPitchSpeedIndex;
8          QPlcSignalIndex _bodyControlYawSpeedIndex;
9          QPlcSignalIndex _tetherSpeedIndex;
10
11         QPlcSignalIndex _bodyPosition_earth_index;
12         QPlcSignalIndex _bodyVelocity_earth_index;
```

```
13          QPlcSignalIndex _windVelocity_earth_index;
14
15          QPlcSignalIndex _bodyControlOrientationIndex;
16          QPlcSignalIndex _bodyPositionOrientationIndex;
17
18          QPlcSignalIndex _angleOfAttackIndex;
19          QPlcSignalIndex _dragCoefficientIndex;
20          QPlcSignalIndex _liftCoefficientIndex;
21          QPlcSignalIndex _aweCoefficientIndex;
22
23          QPlcSignalIndex _aerodynamicForce_earth_index;
24          QPlcSignalIndex _tetherForce_earth_index;
25          QPlcSignalIndex _gravitationalForce_earth_index;
26
27          QPlcSignalIndex _tetherLengthIndex;
28          QPlcSignalIndex _powerIndex;
29          QPlcSignalIndex _averagePowerIndex;
30          QPlcSignalIndex _energyIndex;
31
32      public:
33          ModelNode():
34              QPlcSlaveNode("Model", ErrorZeroOrderHold)
35          {
36              LaweModel_initialize(); //initialize at least once to allocate storage
37          }
38
39      protected:
40          QPlcSignalIndexList declareInputSignals()
41          {
42              QPlcSignalIndexList inputSignalIndics;
43
44              _bodyControlPitchSpeedIndex = inputSignalIndics.insert("
                    ↪ bodyControlPitchSpeed");
45              _bodyControlYawSpeedIndex = inputSignalIndics.insert("bodyControlYawSpeed")
                    ↪ ;
46              _tetherSpeedIndex = inputSignalIndics.insert("tetherSpeed");
47
48              return inputSignalIndics;
49          }
50          QPlcSignalIndexList declareOutputSignals()
51          {
52              QPlcSignalIndexList outputSignalIndices;
53
54              _bodyPosition_earth_index = outputSignalIndices.insert("bodyPosition_earth"
                    ↪ );
55              _bodyVelocity_earth_index = outputSignalIndices.insert("bodyVelocity_earth"
                    ↪ );
56              _windVelocity_earth_index = outputSignalIndices.insert("windVelocity_earth"
                    ↪ );
57
58              _bodyControlOrientationIndex = outputSignalIndices.insert("
                    ↪ bodyControlOrientation");
59              _bodyPositionOrientationIndex = outputSignalIndices.insert("
                    ↪ bodyPositionOrientation");
60
61              _angleOfAttackIndex = outputSignalIndices.insert("angleOfAttack");
62              _dragCoefficientIndex = outputSignalIndices.insert("dragCoefficient");
63              _liftCoefficientIndex = outputSignalIndices.insert("liftCoefficient");
64              _aweCoefficientIndex = outputSignalIndices.insert("aweCoefficient");
65
66              _aerodynamicForce_earth_index = outputSignalIndices.insert("
                    ↪ aerodynamicForce_earth");
```

```
67              _tetherForce_earth_index = outputSignalIndices.insert("tetherForce_earth");
68              _gravitationalForce_earth_index = outputSignalIndices.insert("
                    ↪ gravitationalForce_earth");
69
70              _tetherLengthIndex = outputSignalIndices.insert("tetherLength");
71              _powerIndex = outputSignalIndices.insert("power");
72              _averagePowerIndex = outputSignalIndices.insert("averagePower");
73              _energyIndex = outputSignalIndices.insert("energy");
74
75              return outputSignalIndices;
76          }
77
78      protected:
79          QPlcSignalList initialOutputSignals()
80          {
81              LaweModel_initialize();
82              LaweModel_step(); //Matlab does not set its outports, so we need to call "
                    ↪ step" first
83              return collectModelOutputs();
84          }
85          void initialize(const QPlcSignalList& initialSignals)
86          {
87              distributeModelInputs(initialSignals);
88          }
89          QPlcSignalList realTimeout(const QPlcSignalList& inputSignals)
90          {
91              distributeModelInputs(inputSignals);
92              LaweModel_step();
93              return collectModelOutputs();
94          }
95
96      private:
97          void distributeModelInputs(const QPlcSignalList& inputSignals)
98          {
99              LaweModel_U.bodyControlPitchSpeed = inputSignals.value(
                    ↪ _bodyControlPitchSpeedIndex).toReal();
100             LaweModel_U.bodyControlYawSpeed = inputSignals.value(
                    ↪ _bodyControlYawSpeedIndex).toReal();
101             LaweModel_U.tetherSpeed = inputSignals.value(_tetherSpeedIndex).toReal();
102         }
103         QPlcSignalList collectModelOutputs()
104         {
105             QPlcSignalList outputSignals;
106
107             outputSignals.insert(_bodyPosition_earth_index, LaweModel_Y.
                    ↪ bodyPosition_earth);
108             outputSignals.insert(_bodyVelocity_earth_index, LaweModel_Y.
                    ↪ bodyVelocity_earth);
109             outputSignals.insert(_windVelocity_earth_index, LaweModel_Y.
                    ↪ windVelocity_earth);
110
111             outputSignals.insert(_bodyControlOrientationIndex, LaweModel_Y.
                    ↪ bodyControlOrientation);
112             outputSignals.insert(_bodyPositionOrientationIndex, LaweModel_Y.
                    ↪ bodyPositionOrientation);
113
114             outputSignals.insert(_angleOfAttackIndex, LaweModel_Y.angleOfAttack);
115             outputSignals.insert(_dragCoefficientIndex, LaweModel_Y.dragCoefficient);
116             outputSignals.insert(_liftCoefficientIndex, LaweModel_Y.liftCoefficient);
117             outputSignals.insert(_aweCoefficientIndex, LaweModel_Y.aweCoefficient);
118
```

```
119             outputSignals.insert(_aerodynamicForce_earth_index, LaweModel_Y.
                    ↪ aerodynamicForce_earth);
120             outputSignals.insert(_tetherForce_earth_index, LaweModel_Y.
                    ↪ tetherForce_earth);
121             outputSignals.insert(_gravitationalForce_earth_index, LaweModel_Y.
                    ↪ gravitationalForce_earth);
122
123             outputSignals.insert(_tetherLengthIndex, LaweModel_Y.tetherLength);
124             outputSignals.insert(_powerIndex, LaweModel_Y.power);
125             outputSignals.insert(_averagePowerIndex, LaweModel_Y.averagePower);
126             outputSignals.insert(_energyIndex, LaweModel_Y.energy);
127
128             return outputSignals;
129         }
130 };
131
132 int main(int argc, char** argv)
133 {
134     QPlcCoreApplication application(argc, argv);
135
136     QPlcSignalIndexTable signalIndexTable;
137     signalIndexTable.initializeFromIniFile(application.applicationDirPath() + "/../../
            ↪ Signals/SignalIndexTable.ini");
138
139     ModelNode modelNode;
140     modelNode.activateWorkloadMeasurement("modelWorkload");
141     modelNode.activateWatchdog(0.015);
142     if(!modelNode.connectToMasterAndWaitForConnected(QPlcUnderlyingProtocol::tcp(
            ↪ QHostAddress::LocalHost, 50000)))
143         return 0;
144
145     return application.exec();
146 }
```

Listing 4.14: "ModelMain.cpp".

In the first two lines the QtPLC library headers and the "LaweModel.h" header from the Simulink model are included.

Before walking through the *ModelNode* class, the main function which starts in line 132 is explained: First an application object and the signal index table are instantiated. Then the *ModelNode* is instantiated, its workload measurement and its watchdog with a waiting time of 0.015 s are activated. The timeout message from the master should be received by the model node every 0.010 s. The role of the watchdog is, to output an information if the master timeout message was not received on time and in this particular example with an allowed jitter of +0.005 s. Finally, the model node is connected to the master via TCP on port 50000. For this example it is assumed that the master application is executed on the same computer so its IP address is localhost (i.e. 127.0.0.1 for IPv4). Finally, the application object starts its event loop in line 145.

The *ModelNode*, starting in line 4, is a subclass of *QPlcSlaveNode* which manages the communication with the master, watchdogs and other. To avoid expansive string comparisons by using signal name strings, private variables for all signal indices are declared in lines 7 . . . 30. The type *QPlcSignalIndex* is a 16 bit unsigned integer (i.e. *typedef unsigned short*). Throughout this thesis all private variables start with an underscore "_".

The constructor starting in line 33 passes "Model" as name of the node and *ErrorZeroOrderHold* as realtime violation policy: If the model node does not send its output values to the master before the timer of the master triggers a new timeout, there is a real time violation. *ErrorZeroOrderHold* defines, that in this case the master should output an error message and should use the model's output values from the previous cycle also for the current cycle. The function *LaweModel_initialize ()* is one of two important functions provided by the C++

files that where exported from Simulink. In this function Matlab allocates storage, so this must be called at least once and which is done in the constructor.

In lines 40 . . . 49 the virtual function *declareInputSignals ()* is defined and must return a *QPlcSignalIndexList*. For this, the signal names are passed to the *QPlcSignalIndexList:: insert ()* function in lines 44 . . . 46. This function uses internally the *QPlcSignalIndexTable* instance to get the corresponding 16 bit *QPlcSignalIndex* value. *QPlcSignalIndexList:: insert ()* also returns that *QPlcSignalIndex* which is then stored in the *ModelNode*'s private variables for later usage. The same procedure is used for the output signals in the virtual *declareOutputSignals ()* function in lines 50 . . . 76.

The *declareInputSignals ()* and *declareOutputSignals ()* functions are called during the registration process of the slave to the master, visualized in Fig. 4.14. Right after a slave node is

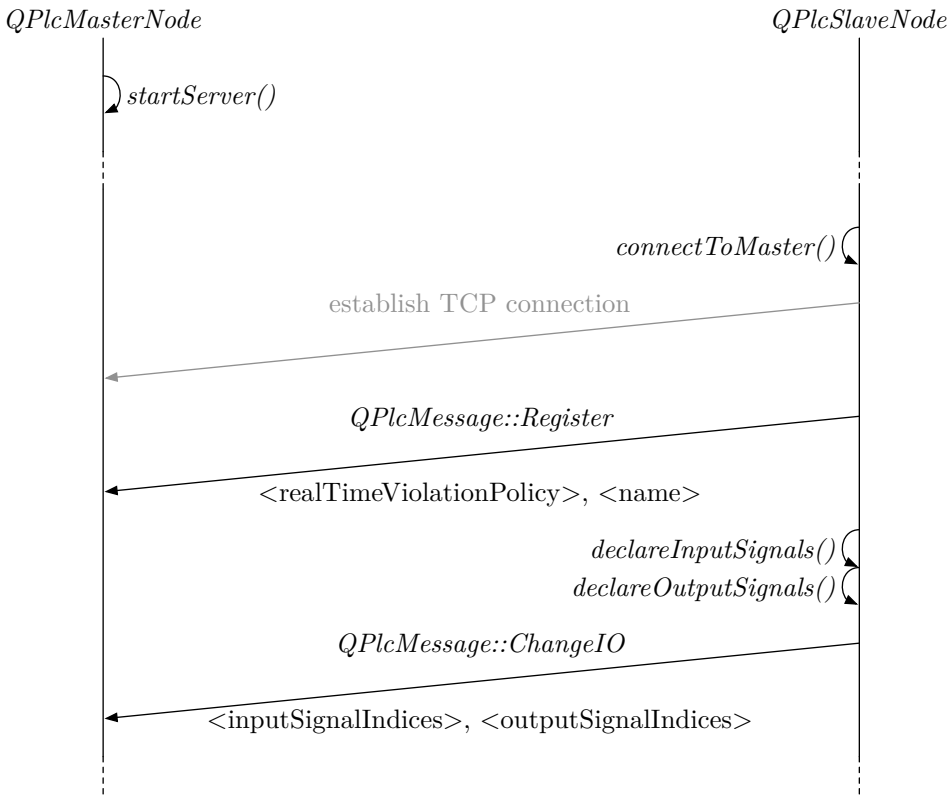


Figure 4.14: Register sequence diagram including the server start of the master.

connected successfully to the master, it registers itself to the master by sending a *Register* message which contains the *RealTimeViolationPolicy* and the node's name as *QString*. After this, the *QPlcSlaveNode* sends the input and output *QPlcSignalIndex*s to the master, so that the master knows which signals it must send to that slave node and expect from that slave node. This whole process is encapsulated and abstracted in the parent *QPlcSlaveNode* class which collects those input and output *QPlcSignalIndex*s by calling the virtual functions *declareInputSignals ()* and *declareOutputSignals ()*.

Before treating the initialization functions in lines 79 . . . 88, the implementation of the pure virtual *realTimeout()* function starting in line 89 is explained: This function is called by the parent *QPlcSlaveNode* during the real time phase, shown in the sequence diagram in Fig. 4.15: The

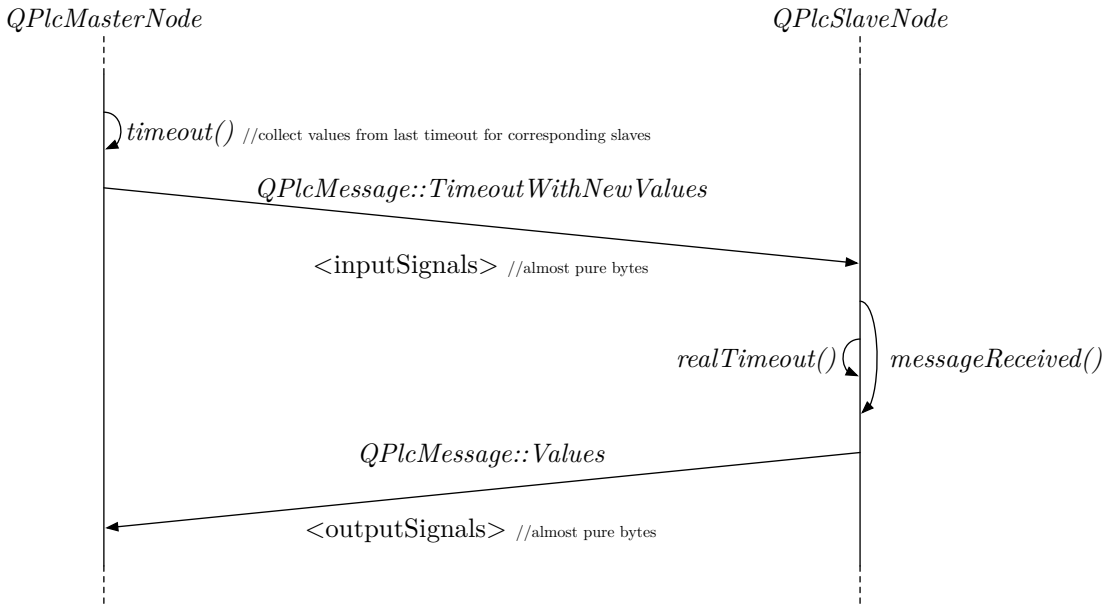


Figure 4.15: Real time sequence diagram for one cycle.

timer of the master triggers a timeout on which the master sends the *TimeoutWithNewValues* message. Attached to this message, the new input values for the corresponding slave node are attached. In order to achieve a minimum overhead, only the pure bytes of the signal values prefixed by the two byte *QPlcSignalIndex* are sent in these messages. The byte size of the values may very, e.g. an integer value has 4 bytes while a double value has 8 bytes. Since the type of a signal is stored in the *QPlcSignalIndexTable*, the values can be resolved easily. With this approach, the messages are still flexible because it is arbitrary on which position which signal is placed or even if the message has too many signals. This repeating task of resolving and packing the messages is done in the *QPlcMessage* class. In the *QPlcSlaveNode::messageReceived()* function, the resolved *QPlcSignalList* is passed to the pure virtual *realTimeout()* function. *QPlcSlaveNode* expects *realTimeout()* to return a list of output signal values with its output signal indices. Those values are then assembled again to a low overhead message and sent back to the master. In the implementation of the *realTimeout()* function in lines 89 . . . 94, first the values are distributed to Simulink's global *LaweModel_U* input variables structure within the *distributeModelInputs ()* function from line 97 . . . 102. The Simulink model performs a discrete time step by calling *LaweModel_step()* in line 92. Finally, in line 93 *collectModelOutputs()* is called, which collects the Simulink model's output variables of the global *LaweModel_Y* output structure in lines 103 . . . 129. Here, the list of signal values with the corresponding signal indices are return in a *QPlcSignalList*.

The two missing initialization functions in lines 79 . . . 88 are used to initialize the states of the controllers, models, etc. before the real time phase is started with the consecutive real time sequences. The initialization sequence is shown in Fig. 4.16. It can be started by a slave node by sending the *InitializeAllStates* message to the master. In this example implementation this can only done by the QtPLC Control Center. With this command, the master collects the initial output signals of each node by sending the *SendYourInitialOutputStates* message to each slave node. This message is processed by the parent *QPlcSlaveNode* class which then calls the virtual function *initialOutputSignals ()*, implemented in line 79 . . . 84: In line 81 . . . 82 the Simulink model is initialized with *LaweModel_initialize ()* and the step function *LaweModel_step()*

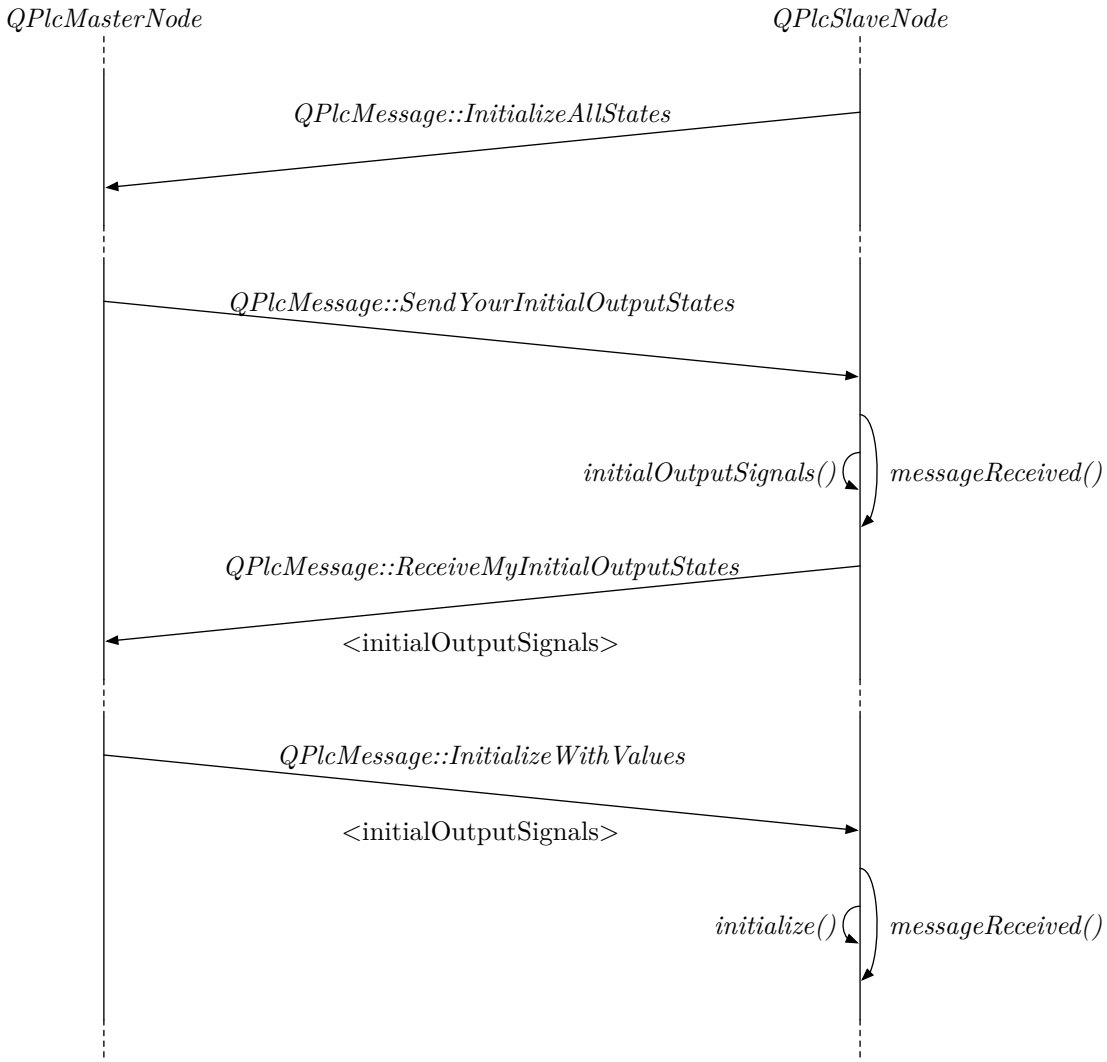


Figure 4.16: Initialization sequence diagram.

is called because Simulink does not set its output variables with *LaweModel_initialize ()* alone. Finally, the return value of the *collectModelOutputs()* function is further returned in line 83. The parent *QPlcSlaveNode* class then sends back the *ReceiveMyInitialOutputStates* message to the master. After the master has collected all initial output signals from all slaves, it sends the corresponding input signals as initial signals with the *InitializeWithValues* message to each slave node. This message is processed by the parent *QPlcSlaveNode* which calls the virtual *initialize()* function with the passed initial signals, implemented in line 85 . . . 88. This function only calls the *distributeModelInputs ()* function. So input signals are also used for initial signals. In this particular example all initial signals are hard coded in the Simulink model and thus the last function did not need to be reimplemented here (the default implementation in *QPlcSlaveNode* does nothing, it is not a pure virtual function). However, it is included for completeness to show how initial values can be transmitted.

The *QPlcMasterNode* has a state machine with the states

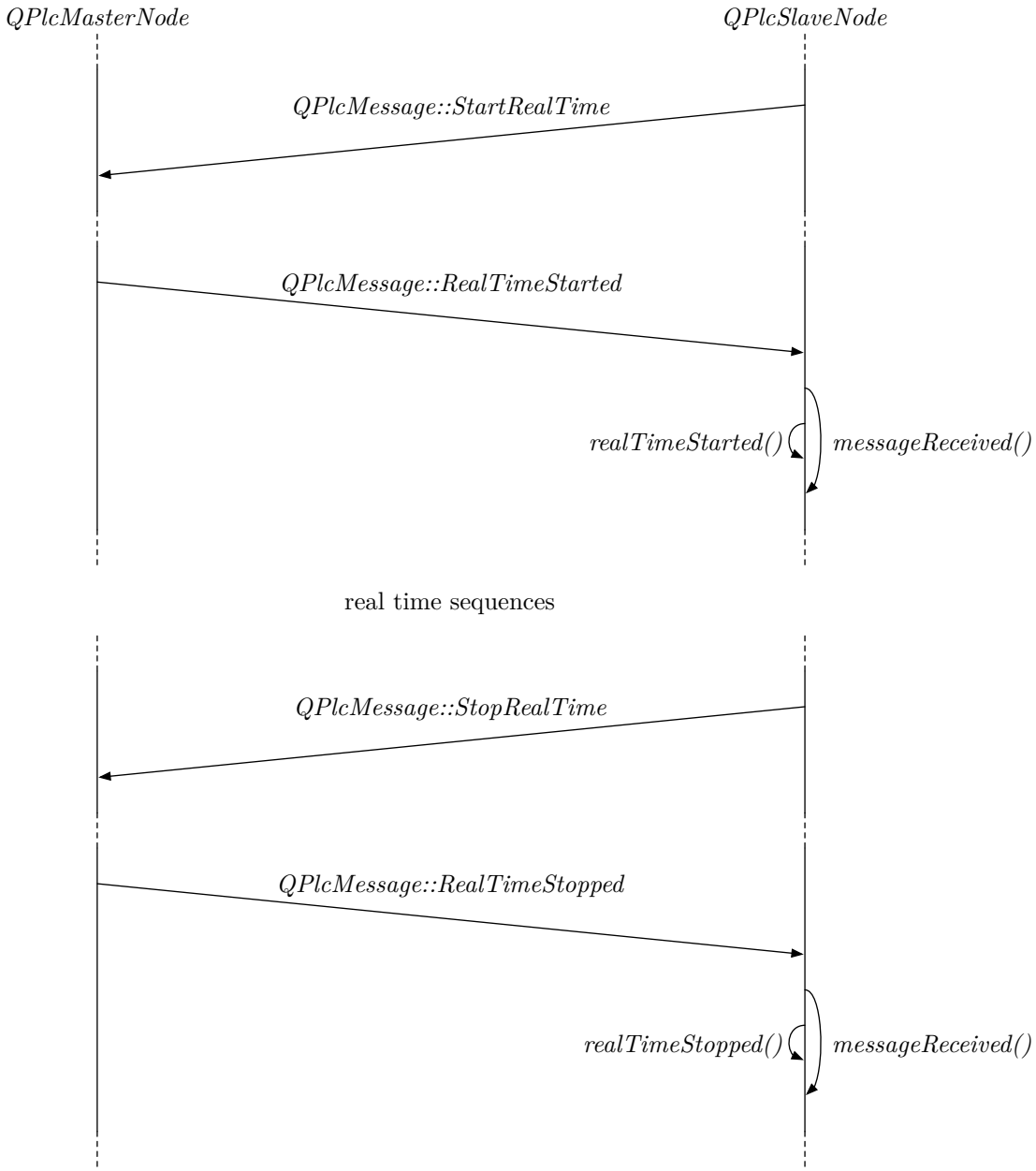


Figure 4.17: Start and stop sequence diagram.

- *Off*, which is the initial state,
- *Initializing*,
- *RealTime* and
- *WaitingForLateSlaves*.

The initialization and real time sequence can only be started, if the master is in the *Off* state. As soon as the master receives the *InitializeAllStates* message, the master goes into the *Initializing* state. In this state, it also starts a watchdog timer to aboard the initialization procedure, if during the sequence a slave does not respond with the *ReceiveMyInitialOutputStates* message. The start and stop sequences are visualized in Fig. 4.17: With the *StartRealTime* message the real time phase is started. This message can be sent by any slave, but in this example only by the QtPLC Control Center. The master informs all slaves about the start of the real time phase with the *RealTimeStarted* message and starts its timer. If the master's timer triggered a timeout and a slave with the *SilentWait* or *ErrorWait* real time policy has not answered, the master stops its timer and goes into the *WaitingForLateSlaves* state. These real time policies are beneficial for software-in-the-loop tests on non real time operating systems. As soon as all late slaves have sent their outputs, the master returns to the *RealTime* state. The real time sequence can be stopped by sending the *Stop* message to the master.

The whole communication is implemented with sockets, namely *QPlcMasterSocket* used by *QPlcSlaveNode* (since a slave writes messages to the master) and *QPlcSlaveSocket*s used by *QPlcMasterNode* (since the master writes messages to the slaves). Those sockets encapsulate the actual *QTcpSocket*. So in later implementations also other sockets like the *QUdpSocket* or sockets from other libraries can be chosen, by changing the choice of the *QPlcUnderlyingProtocol*.

4.2.4 Compact model application

The last section explained in detail how the communication works and how much already is abstracted by the *QPlcSlaveNode*. However, QtPLC provides several subclasses of *QPlcSlaveNode* for further simplifications: In the "CompactModel" application the model node is implemented with the *QPlcCompactSlaveNode*.

The "CompactModel" project uses the same from Simulink exported C++ files which are located in the "Model/Sources" directory. So the project file is slightly different from that of the last section and is shown in Lst. 4.15.

```
1  TARGET = CompactModel
2  CONFIG += Console
3
4  PRO = $$PWD
5
6  SOURCEDIRS += $$system(find $$PRO/../Model/Sources/LaweModel -type d)
7  SOURCES += $$system(find $$PRO/../Model/Sources/LaweModel -name '*.cpp')
8  HEADERS += $$system(find $$PRO/../Model/Sources/LaweModel -name '*.h')
9
10 include(../AirborneWindEnergyExample.pri)
```

Listing 4.15: "CompactModel.pro".

So the "Sources" directory only contains the "CompactModelMain.cpp" file. Its content is shown in Lst. 4.16.

```
1  #include <QtPlc>
2  #include "LaweModel.h"
3
4  class CompactModelNode: public QPlcCompactSlaveNode
5  {
6      private:
7          QPlcCompactSignal _bodyControlPitchSpeed = QPlcCompactSignal("
               ↪ bodyControlPitchSpeed", QPlcCompactSignal::Input, this);
8          QPlcCompactSignal _bodyControlYawSpeed = QPlcCompactSignal("bodyControlYawSpeed
               ↪ ", QPlcCompactSignal::Input, this);
9          QPlcCompactSignal _tetherSpeed = QPlcCompactSignal("tetherSpeed",
               ↪ QPlcCompactSignal::Input, this);
```

```
10
11          QPlcCompactSignal _bodyPosition_earth = QPlcCompactSignal("bodyPosition_earth",
                ↪  QPlcCompactSignal::Output, this);
12          QPlcCompactSignal _bodyVelocity_earth = QPlcCompactSignal("bodyVelocity_earth",
                ↪  QPlcCompactSignal::Output, this);
13          QPlcCompactSignal _windVelocity_earth = QPlcCompactSignal("windVelocity_earth",
                ↪  QPlcCompactSignal::Output, this);
14
15          QPlcCompactSignal _bodyControlOrientation = QPlcCompactSignal("
                ↪ bodyControlOrientation", QPlcCompactSignal::Output, this);
16          QPlcCompactSignal _bodyPositionOrientation = QPlcCompactSignal("
                ↪ bodyPositionOrientation", QPlcCompactSignal::Output, this);
17
18          QPlcCompactSignal _angleOfAttack = QPlcCompactSignal("angleOfAttack",
                ↪ QPlcCompactSignal::Output, this);
19          QPlcCompactSignal _dragCoefficient = QPlcCompactSignal("dragCoefficient",
                ↪ QPlcCompactSignal::Output, this);
20          QPlcCompactSignal _liftCoefficient = QPlcCompactSignal("liftCoefficient",
                ↪ QPlcCompactSignal::Output, this);
21          QPlcCompactSignal _aweCoefficient = QPlcCompactSignal("aweCoefficient",
                ↪ QPlcCompactSignal::Output, this);
22
23          QPlcCompactSignal _aerodynamicForce_earth = QPlcCompactSignal("
                ↪ aerodynamicForce_earth", QPlcCompactSignal::Output, this);
24          QPlcCompactSignal _tetherForce_earth = QPlcCompactSignal("tetherForce_earth",
                ↪ QPlcCompactSignal::Output, this);
25          QPlcCompactSignal _gravitationalForce_earth = QPlcCompactSignal("
                ↪ gravitationalForce_earth", QPlcCompactSignal::Output, this);
26
27          QPlcCompactSignal _tetherLength = QPlcCompactSignal("tetherLength",
                ↪ QPlcCompactSignal::Output, this);
28          QPlcCompactSignal _power = QPlcCompactSignal("power", QPlcCompactSignal::Output
                ↪ , this);
29          QPlcCompactSignal _averagePower = QPlcCompactSignal("averagePower",
                ↪ QPlcCompactSignal::Output, this);
30          QPlcCompactSignal _energy = QPlcCompactSignal("energy", QPlcCompactSignal::
                ↪ Output, this);
31
32      public:
33          CompactModelNode():
34              QPlcCompactSlaveNode("CompactModel", ErrorZeroOrderHold)
35          {
36              LaweModel_initialize(); //initialize at least once to allocate storage
37          }
38
39      protected:
40          void compactInitializeOutputSignals()
41          {
42              LaweModel_initialize();
43              LaweModel_step(); //Matlab does not set its outports, so we need to call "
                    ↪ step" first
44              collectOutputs();
45          }
46          void initialized()
47          {
48              collectInputs();
49          }
50          void compactRealTimeout()
51          {
52              collectInputs();
53              LaweModel_step();
54              collectOutputs();
```

```
55          }
56
57      private:
58          void collectInputs()
59          {
60              LaweModel_U.bodyControlPitchSpeed = _bodyControlPitchSpeed;
61              LaweModel_U.bodyControlYawSpeed = _bodyControlYawSpeed;
62              LaweModel_U.tetherSpeed = _tetherSpeed;
63          }
64          void collectOutputs()
65          {
66              _bodyPosition_earth = LaweModel_Y.bodyPosition_earth;
67              _bodyVelocity_earth = LaweModel_Y.bodyVelocity_earth;
68              _windVelocity_earth = LaweModel_Y.windVelocity_earth;
69
70              _bodyControlOrientation = LaweModel_Y.bodyControlOrientation;
71              _bodyPositionOrientation = LaweModel_Y.bodyPositionOrientation;
72
73              _angleOfAttack = LaweModel_Y.angleOfAttack;
74              _dragCoefficient = LaweModel_Y.dragCoefficient;
75              _liftCoefficient = LaweModel_Y.liftCoefficient;
76              _aweCoefficient = LaweModel_Y.aweCoefficient;
77
78              _aerodynamicForce_earth = LaweModel_Y.aerodynamicForce_earth;
79              _tetherForce_earth = LaweModel_Y.tetherForce_earth;
80              _gravitationalForce_earth = LaweModel_Y.gravitationalForce_earth;
81
82              _tetherLength = LaweModel_Y.tetherLength;
83              _power = LaweModel_Y.power;
84              _averagePower = LaweModel_Y.averagePower;
85              _energy = LaweModel_Y.energy;
86          }
87  };
88
89  int main(int argc, char** argv)
90  {
91      QPlcCoreApplication application(argc, argv);
92
93      QPlcSignalIndexTable signalIndexTable;
94      signalIndexTable.initializeFromIniFile(application.applicationDirPath() + "/../../
            ↪ Signals/SignalIndexTable.ini");
95
96      CompactModelNode compactModelNode;
97      compactModelNode.activateWorkloadMeasurement("modelWorkload");
98      compactModelNode.activateWatchdog(0.015);
99      if(!compactModelNode.connectToMasterAndWaitForConnected(QPlcUnderlyingProtocol::tcp
            ↪ (QHostAddress::LocalHost, 50000)))
100         return 0;
101
102     return application.exec();
103 }
```

Listing 4.16: “CompactModelMain.cpp”.

The main function in lines 89 . . . 103 is hardly different to that of Lst. 4.14, so only the *CompactModel* class definition starting in line 4 needs to be explained: Instead of using *QPlcSignalIndex*s, the main difference to the last section is the use of *QPlcCompactSignal*s. The private *QPlcCompactSignal* variables are instantiated and initialized in lines 7 . . . 30. The direct assignment of the variables is possible in C++11. A *QPlcCompactSignal* contains three important properties which are passed as arguments to the constructor:

- The first argument is the signal name string, which is internally resolved using the *QPlcSignalIndexTable* instance,
- the second argument specifies if it is an input or output and
- the third argument specifies for which slave node the second argument (input or output) is meant, which is in all cases *this*.

The *QPlcCompactSignal* constructor then appends itself to a list of input and output signals, respectively, of the *QPlcCompactSlaveNode* object. The signal indices are then passed to the parent *QPlcSlaveNode* class in the reimplemented virtual functions *QPlcCompactSlaveNode::declareInputSignals()* and *QPlcCompactSlaveNode::declareOutputSignals()*. Due to operator overloading in C++, *QPlcCompactSignal* objects can be used like intrinsic type variables, like *int*s or *float*s.

Except of the name string of the node, the constructor in lines 33 . . . 37 is identical to that of the ordinary model node of the last section. In line 50 the pure virtual function *compactRealTimeout* is implemented. This function now neither accepts input signal values as argument nor returns output signal values. This is not intuitive and is one reason why both slave node implementation variants are presented. After the the node received the timeout message from the master, the parent *QPlcCompactSlaveNode* sets its input *QPlcCompactSignal*s to the correct value. Then it calls the pure virtual *compactRealTimeout()* function, which is implemented in lines 50 . . . 55. Finally, it collects the values of the output *QPlcCompactSignal*s and returns them to the master. Similar to the last section, the two functions *collectInputs()* and *collectOutputs()* are called in *compactRealTimeout()* before and after the call of *LaweModel_step()*, respectively. In the *collectInputs()* and *collectOutputs()* functions in lines 58 . . . 63 and 64 . . . 86, the *QPlcCompactSignal* variables are used like intrinsic variables to set and read the global Simulink structures *LaweModel_U* and *LaweModel_Y*, respectively.

In conclusion, this source code is much more compact – a third less lines than the model code from the last section. Additionally, the code is better readable because it is more abstract.

4.2.5 Control center application

Before digging into the source code of the QtPLC Control Center application, the final result is discussed. A screenshot is shown in Fig. 4.18. The similarity of the design to Qt Creator in Fig. 4.12 on p. 57 is not by accident, since the library is named "QtPLC". However, not a single line of the Qt Creator source code was used for the Qt PLC Control Center.

The whole GUI is build up modularly and customizable: There is one main window into which several views or "widgets" are docked. With the toolbar buttons on the left side, more widget instances can be opened and the QtPLC Control Center can be connected to or disconnected from the master. A new widget is opened in a new window and may be docked in the main window at any position. Any widget may be undocked from the main window by clicking the undock button with the up-arrow icon. The widgets' sizes can be adjusted by moving their borders with the mouse. A widget is dismissed by clicking the "x" button.

In Fig. 4.18 only four different types of widgets are shown:

- One "Keyboard Controller" widget in the top left corner with which the three control variables pitch speed, yaw speed and tether speed are set using the corresponding keyboard buttons.
- Four "Gauge" widgets, three below the keyboard controller widget and one in the bottom right corner. The gauges have different visualization settings and show the values of different signals. The "Gauge" widget is included in the QtPLC library.

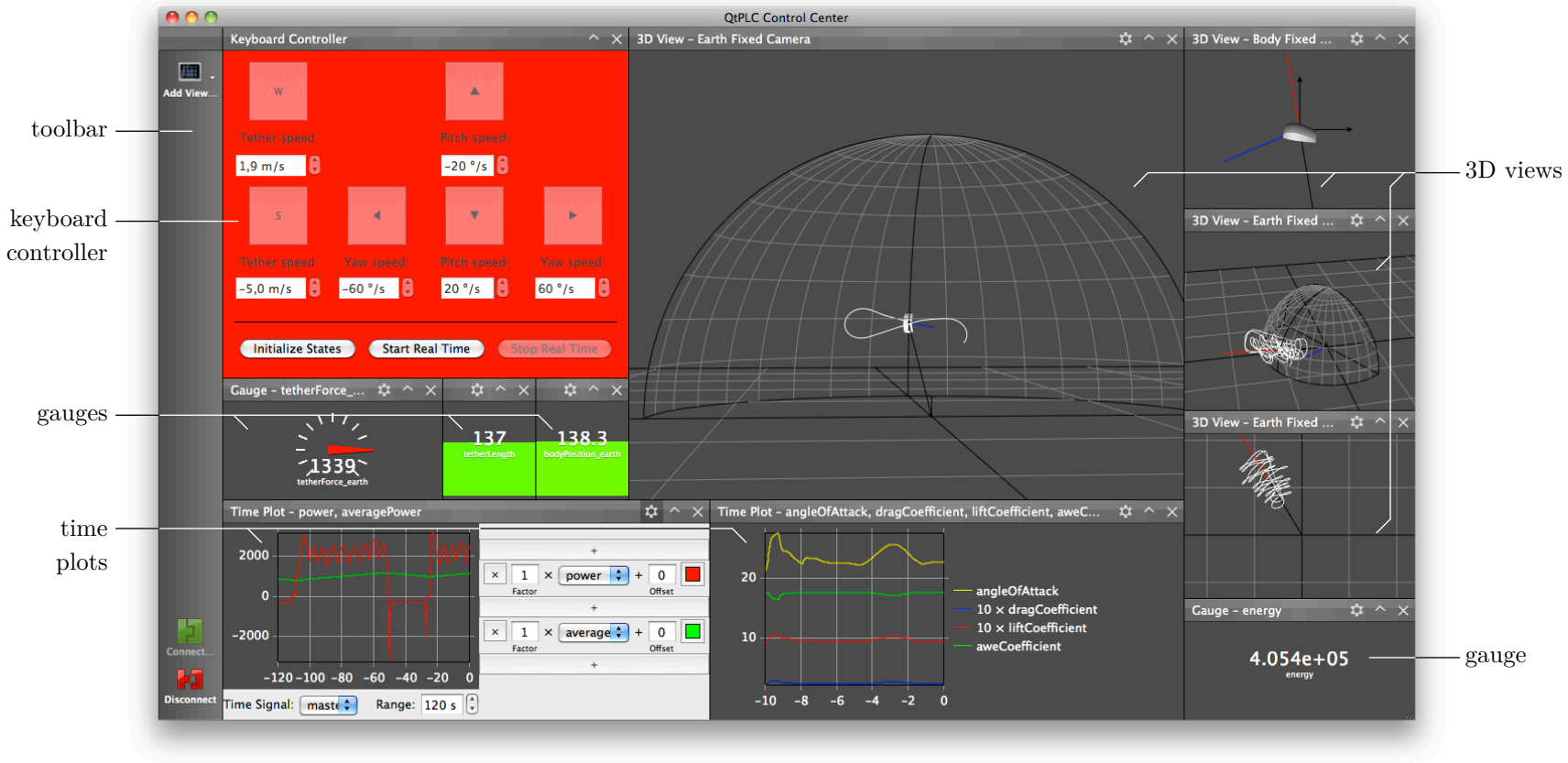


Figure 4.18: Screenshot of the final QtPLC Control Center.

- Two “Time Plot” widgets in the bottom left corner to visualize the trend of different signal values over time. For the left “Time Plot” widget, the settings are opened by clicking on the toothed wheel symbol. Such settings are available with that button for almost any widget. In the “Time Plot” widget the plotted signals can be selected, added and removed. Additionally, a gain, an offset and the color of the signal graphs can be chosen individually. The master time signal and the time range is chosen on the bottom. The range of the vertical axis is adjusted automatically. The “Time Plot” widget is also included in the QtPLC library.

- Four “3D View” widgets in the top right area, for which different camera and visualization settings are chosen: The biggest “3D View” in the middle shows the system from the earth fixed camera, the view on the top right shows the system close to the body from the body fixed camera and the last two views show the system also from the earth fixed camera but from different positions. In addition to that, the last two views visualize the flight path of the body for the last 180 s, while the “3D View” in the middle shows the flight path only for the last 10 s. In all four views, the relative wind velocity vector is visualized in blue and the total aerodynamic force vector is visualized in red. The wing or body is visualized with a 3D model of a kite taken from a Google SketchUp file[30]. It is also possible to hide the coordinate systems and the small earth sphere. The camera position can be changed using the mouse and the “Shift” and “Alt” keys. This widget is implemented with the use of the Qt3D library, which is (not yet) part of the Qt framework.

It is quite simple to write a QtPLC Control Center application with custom widgets. The project file of the control center application is shown in Lst. 4.17.

```
1 TARGET = ControlCenter
2 CONFIG += Gui
3 CONFIG += Qt3D
4
5 PRO = $$PWD
6 include(../AirborneWindEnergyExample.pri)
```

Listing 4.17: “CompactModel.pro”.

The differences to the other project files is, that now the application is configured as “Gui” and the Qt3D library is included.

The main file “ControlCenterMain.cpp” is shown in Lst. 4.18.

```
1  #include <QtPlc>
2  #include "KeyboardController.h"
3  #include "View3D.h"
4
5  int main(int argc, char** argv)
6  {
7      //gui thread
8          QPlcGuiApplication application(argc, argv);
9          application.thread()->setPriority(QThread::LowestPriority);
10
11
12     //plc thread
13         QThread* plcThread = new QThread;
14         plcThread->start(QThread::TimeCriticalPriority);
15
16
17     //slave node
```

[30] 3D model (edited): Noble, Jeremy: “Parafoil Kite”. http://sketchup.google.com/3dwarehouse/details?mid=8544eaee0d6b5f946daf1f9856de2fa0&prevstart=0, accessed: October 21, 2013.

```
18          QPlcSignalIndexTable signalIndexTable;
19          #ifdef Q_OS_MAC
20              signalIndexTable.initializeFromIniFile(application.applicationDirPath() + "
                    ↪ /../../../../../Signals/SignalIndexTable.ini");
21          #else
22              signalIndexTable.initializeFromIniFile(application.applicationDirPath() + "
                    ↪ /../../Signals/SignalIndexTable.ini");
23          #endif
24
25          QPlcIOBufferSlaveNode* slaveNode = new QPlcIOBufferSlaveNode("ControlCenter",
                ↪ QPlcIOBufferSlaveNode::SilentZeroOrderHold, false);
26          slaveNode->moveToThread(plcThread);
27
28          QPlcFrameRateLimiter* frameRateLimiter = new QPlcFrameRateLimiter(slaveNode);
29          frameRateLimiter->setFrameRateLimit(50);
30
31
32      //control center
33          QPlcControlCenter* controlCenter = new QPlcControlCenter;
34          controlCenter->addAvailableWidget(QPlcWidget::registerFactory(new
                ↪ KeyboardControllerFactory));
35          controlCenter->addAvailableWidget(QPlcWidget::registerFactory(new
                ↪ QPlcGaugeFactory));
36          controlCenter->addAvailableWidget(QPlcWidget::registerFactory(new
                ↪ QPlcTimePlotFactory));
37          controlCenter->addAvailableWidget(QPlcWidget::registerFactory(new View3DFactory
                ↪ ));
38          controlCenter->restoreView();
39          controlCenter->show();
40
41
42      //connections
43          QObject::connect(frameRateLimiter, SIGNAL(timedOut(QList<QPlcSignalList>)),
                ↪ controlCenter, SLOT(realTimeout(QList<QPlcSignalList>)));
44
45          QObject::connect(controlCenter, SIGNAL(connectRequested(QPlcUnderlyingProtocol)
                ↪ ), slaveNode, SLOT(connectToMaster(QPlcUnderlyingProtocol)));
46          QObject::connect(slaveNode, SIGNAL(connectedToMaster()), controlCenter, SLOT(
                ↪ connectedToMaster()));
47          QObject::connect(slaveNode, SIGNAL(errorWithMaster(QPlcAbstractSocket::
                ↪ SocketError)), controlCenter, SLOT(errorWithMaster(QPlcAbstractSocket::
                ↪ SocketError)));
48
49          QObject::connect(controlCenter, SIGNAL(disconnectRequested()), slaveNode, SLOT(
                ↪ disconnectFromMaster()));
50          QObject::connect(slaveNode, SIGNAL(disconnectedFromMaster()), controlCenter,
                ↪ SLOT(disconnectedFromMaster()));
51
52          QObject::connect(controlCenter, SIGNAL(initialOutputSignalsChanged(
                ↪ QPlcSignalList)), slaveNode, SLOT(setInitialOutputSignals(QPlcSignalList
                ↪ )));
53          QObject::connect(slaveNode, SIGNAL(hasInitialized(QPlcSignalList)),
                ↪ controlCenter, SLOT(initialize(QPlcSignalList)));
54
55          QObject::connect(slaveNode, SIGNAL(realTimeHasStarted()), controlCenter, SLOT(
                ↪ started()));
56          QObject::connect(slaveNode, SIGNAL(realTimeHasStopped()), controlCenter, SLOT(
                ↪ stopped()));
57
58          QObject::connect(controlCenter, SIGNAL(inputSignalIndicesChanged(
                ↪ QPlcSignalIndexList)), slaveNode, SLOT(setInputSignalIndices(
                ↪ QPlcSignalIndexList)));
```

```
59          QObject::connect(controlCenter, SIGNAL(outputSignalIndicesChanged(
                ↪ QPlcSignalIndexList)), slaveNode, SLOT(setOutputSignalIndices(
                ↪ QPlcSignalIndexList)));
60          QObject::connect(controlCenter, SIGNAL(sendNewOutputSignals(QPlcSignalList)),
                ↪ slaveNode, SLOT(setOutputSignals(QPlcSignalList)));
61          QObject::connect(controlCenter, SIGNAL(sendMessage(QPlcMessage)), slaveNode,
                ↪ SLOT(sendMessageToMaster(QPlcMessage)));
62
63
64      //initialize
65          slaveNode->setInputSignalIndices(controlCenter->inputSignalIndices());
66          slaveNode->setOutputSignalIndices(controlCenter->outputSignalIndices());
67          slaveNode->setInitialOutputSignals(controlCenter->initialOutputSignals());
68
69
70      //event loop
71          return application.exec();
72  }
```

Listing 4.18: "ControlCenterMain.cpp".

In the first lines 7 . . . 14 a *QPlcGuiApplication* object is instantiated. This is a *QApplication* object where all codecs are set to UTF-8 and the QtPLC resources are initialized. Its GUI thread priority is set to the lowest. Then the PLC thread is instantiated and started with the highest priority. So, here the GUI and the communication with the master are performed in different threads, such that real time violations are not likely to occur. As for any other slave node, in lines 17 . . . 23 the *QPlcSignalIndexTable* is instantiated and initialized with the INI file. Only the path for Mac OS X is slightly different because executables in Mac OS X are packed with most of the needed resources in a "*.app" folder. In line 25 a *QPlcIOBufferSlaveNode* is instantiated with the given name string and the real time violation policy *SilentZeroOrderHold*, i.e. if the control center does not answer within the master cycle time, the master just takes silently the values from the previous cycle without an error message. Then the slave node is moved to the PLC thread.

QPlcIOBufferSlaveNode is a thread save subclass of *QPlcIOSlaveNode* which further is a subclass of *QPlcSlaveNode*. *QPlcIOBufferSlaveNode* works closely together with the *QPlcFrameRateLimiter* which is instantiated in line 28. The frame rate limit is set to 50 frames per second in line 29. For high control cycle frequencies, the frame rate limiting in combination with the *QPlcIOBufferSlaveNode* object is an essential feature to avoid that the GUI lags more and more behind the master and consumes all system resources: On timeout, the *QPlcIOBufferSlaveNode* receives new values from the master and stores them in a list – or, for better imagination, in a table where in each column the signals from the same timeout are stored. Then the *QPlcIOBufferSlaveNode* object of the PLC thread emits the Qt signal *QPlcSlaveNode::realTimedOut()* which is caught by the *QPlcFrameRateLimiter* of the GUI thread. Since a GUI is on the highest level in the automation pyramid, see Fig. 3.1 on p. 28, usually it does not output that high time critical values to the PLC. So *QPlcIOSlaveNode* sends directly back to the master the anytime before set output values with the slot *QPlcIOSlaveNode::setOutputSignals()*. When a new frame is to be rendered, i.e.

- if the last frame is older than the minimum frame period time (i.e. the reciprocal of the frame rate limit) and
- if the *QPlcFrameRateLimiter* caught a new *QPlcSlaveNode::realTimedOut()* Qt signal,

then the *QPlcFrameRateLimiter* takes all buffered signal values by calling the thread safe *returnAndClearInputSignalsBuffer()* function of the *QPlcIOBufferSlaveNode*. The returned list of signal values for the past cycles are then sent to the GUI – the *QPlcControlCenter* class –

by emitting the *QPlcFrameRateLimiter::timedOut()* Qt signal. This specific Qt signal and slot connection is established in line 43. The widgets in the *QPlcControlCenter* then add these new values to their internal data structures and repaint themselves only once.

The most important part of this file is in lines 32 . . . 39. Here, the *QPlcControlCenter* is instantiated and configured. In lines 34 . . . 37 all the widgets, that shall be available in the *QPlcControlCenter*, are registered. Factories are used to enable *QPlcControlCenter* to instantiate *new* arbitrary widgets. Before this mechanism is explained, the main function is completed: In line 38 the *QPlcControlCenter* is restored to the same state when it was closed. In line 39 it is shown with call of *show()*. In the rest of the main file the thread safe Qt signals and slots connections between the *QPlcControlCenter* and *QPlcIOBufferSlaveNode* are established. Additionally, the slave node is initialized and the application's event loop is started.

In the following the factory mechanism is explained: All the four mentioned widget types in Fig. 4.18 on p. 77 are a subclass of *QPlcWidget*, which may be seen as the GUI version of *QPlcSlaveNode*. Snippets of the most important parts of its header file "QPlcWidget.h" are shown in Lst. 4.19.

```
1  #ifndef QPLCWIDGET_H
2  #define QPLCWIDGET_H
3
4  #include "QtHeaders.h"
5  #include "QPlcSignalIndexTable.h"
6  #include "QPlcSignalList.h"
7  #include "QPlcSignalIndexList.h"
8  #include "QPlcMessage.h"
9
10 class QPlcWidget;
11
12 class QPlcWidgetFactory
13 {
14     public:
15         virtual int type() const = 0;
16         virtual QString name() const = 0;
17         virtual ~QPlcWidgetFactory() {}
18         virtual QPlcWidget* createNew(QWidget* parent = 0) const = 0;
19 };
20
21 class QPlcWidget: public QWidget
22 {
23     //CONSTRUCT
24         Q_OBJECT
25         //construct
26         public:
27             QPlcWidget(QWidget* parent = 0);
28
29         //type
30         public:
31             enum Type
32             {
33                 Gauge,
34                 TimePlot,
35                 UserType
36             };
37             virtual int type() const = 0;
38
39         //factory
40         private:
41             static QList<QPlcWidgetFactory*> _factories;
42         public:
43             static QPlcWidgetFactory* registerFactory(QPlcWidgetFactory* factory);
```

```
44              static QPlcWidgetFactory* factory(int type);
45              virtual QPlcWidgetFactory* factory() const {return factory(type());}
46
47          //...
48
49
50
51      //CONFIGURATION
52          //input signals
53          //...
54          protected:
55              void setInputSignalIndices(const QPlcSignalIndexList& inputSignalIndices);
56          //...
57
58          //output signals
59          //...
60          protected:
61              void setOutputSignalIndices(const QPlcSignalIndexList& outputSignalIndices)
                    ↪ ;
62          //...
63
64
65      //IO
66          //...
67
68          //values
69          public slots:
70              virtual void realTimeout(const QList<QPlcSignalList>& /*signalValues*/) {}
71          //...
72  };
73
74  #endif
```

Listing 4.19: Important snippets of "QPlcWidget.h".

In lines 51 . . . 62 the input and output signal indices can be set by calling the corresponding functions. In lines 65 . . . 71 the most important communications to the master can be accessed by reimplementing those virtual functions in a subclass. In lines 23 . . . 37 the constructor and a pure virtual *type()* function are declared.

Since the *QPlcControlCenter* should be able to instantiate new widgets dynamically, which is not possible with the command *new MyQPlcWidgetSubclass* because types cannot be passed in C++, this is implemented with the factory approach [30, pp. 85]: Each subclass of *QPlcWidget* also creates a subclass of *QPlcWidgetFactory*, defined in lines 12 . . . 19. An example subclass of that is the *QPlcGaugeFactory*, shown in Lst. 4.20.

```
1  class QPlcGaugeFactory: public QPlcWidgetFactory
2  {
3      public:
4          int type() const {return QPlcWidget::Gauge;}
5          QString name() const {return "Gauge";}
6          QPlcWidget* createNew(QWidget* parent = 0) const {return new QPlcGauge(parent)
               ↪ ;}
7  };
```

Listing 4.20: *QPlcGaugeFactory* implementation in "QPlcGauge.h".

So the creation of new objects is embedded in the implemented pure virtual function *QPlcGaugeFactory::createNew()*. Since only one widget factory per widget type is needed, the widget factory is registered with the static function *QPlcWidget::registerFactory()* in line 43 in Lst. 4.19. This stores the passed factory object in the static private *_factories* variable in line 41. This list can be accessed by the functions in lines 44 . . . 45.

With this background, lines 34 . . . 37 of the "ControlCenterMain.cpp" file in Lst. 4.18 are easier understandable: A new instance of the widget factory is created with *new* and passed directly to the static *QPlcWidget::registerFactory()* function. For simplicity, this function returns its argument – the new factory – which is further passed to *addAvailableWidget()* of the *QPlcControlCenter*. The *QPlcControlCenter* then uses the *createNew()* function of the factories to instantiate new widgets.

These are the most important concepts for the QtPLC Control Center. Although it was interesting to dig further in the actual implementations of the different widgets, this is waived here, because more or less standard Qt GUI implementation concepts were used. Only a few key words shall be mentioned: Except for the comparably simple "Keyboard Controller" widget, each widget has a canvas widget to render the actual data and some configuration widgets, which can be shown via a click on the toothed wheel button. The "Time Plot" widget renders its graphs with OpenGL using the *QGLWidget*. The "3D View" widget is the most complex widget. Its canvas is a subclass of *QGLView* from the Qt3D library, so it uses OpenGL, too. All the 3D items are a subclass of *QGLSceneNode*. The coordinate transformations are performed by the appropriate creation of a *QGLSceneNode* tree and applying *QGraphicsTransform3D*s, similar to the Simulink implementation.

4.3 Execution performance

The processes were tested software-in-the-loop with two test setups. In both cases, the applications were compiled in "release mode" which does basically not add debug information into the Assembly, i.e. no "-g" option for the GCC compiler, but compiles the sources with the highest optimization, i.e. "-O2" option for the GCC compiler. So the processes ran with maximum speed.

4.3.1 Ordinary operating system setup

In the first setup, all three processes ran on a MacBook with a 2 GHz dual core processor on the "ordinary" operating system Mac OS X 10.6.8 (Snow Leopard). The processor load was monitored with the *Activity Monitor* application.

The master process occupied 3.2 % of the capacity of one processor core, i.e. 1.6 % of the overall processor capacity. The model process, no matter if the "ordinary" or the "compact" version was used, occupied even less: only 1.8 % of the capacity of one processor core, i.e. only 0.9 % of the overall processor capacity. This is an interesting figure, because if the model was simulated in Simulink, the load of the Matlab/Simulink application exceeded clearly 100 % of the capacity of one processor core, while the simulation time ran more than approximately three times slower than real time. So, by exporting the model to C++, an execution speed increase of a factor of more than 150 was reached. The QtPLC Control Center application occupied almost 100 % of the load of one processor, which is due to the high frame rate of 50 Hz.

Occasionally, the model reported watchdog timeouts and the master reported that the model did not answer on time. Depending on what other applications ran, these notifications came up in the magnitude of seconds. With the control cycle period time of 0.010 s, that implies that in more than 99 % of the time, the real time constraints were met. This is again an astonishing figure, since this was achieved on an ordinary operating system. Additionally, the master jitter, master workload and model workload times were measured and plotted in a "Time Plot" widget. A screenshot of that widget is shown in Fig. 4.19. The measurements are implemented in the QtPLC library with *QElapsedTimer*s. On Mac OS X there seems to be something that is triggered approximately every second, which would explain the periodic peaks.

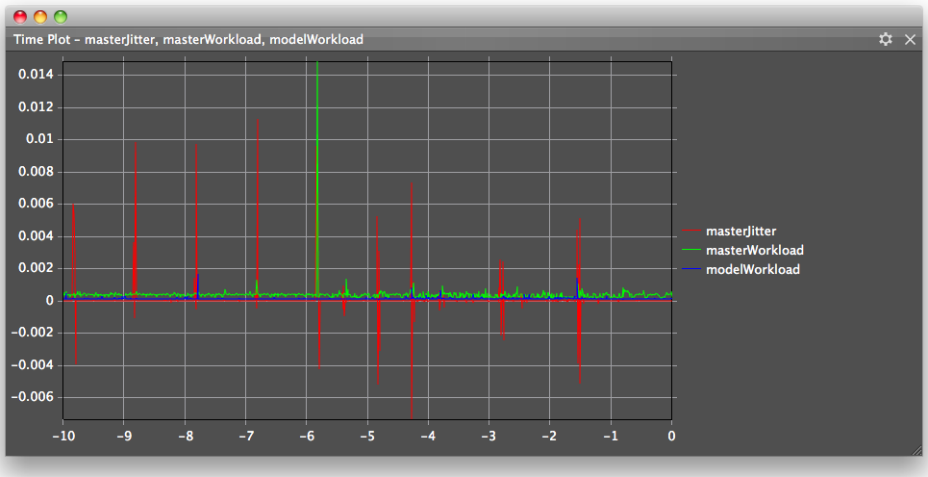


Figure 4.19: Screenshot of the "Time Plot" widget displaying the master jitter, master workload and model workload times while all processes ran on a MacBook with Mac OS X 10.6.8 (Snow Leopard).

4.3.2 Preempt-RT real time Linux operating system setup

In the second setup, a Raspberry Pi Model B with a 700 MHz processor and a Preempt-RT patched Raspbian real time operating system with the Linux kernel 3.8.13-rt14+ was used. On that, the master and the compact model where executed as *chrt* user with the recommended priority 49 [14]. The QtPLC Control Center ran on the MacBook. Raspberry Pi and Macbook where connected with a cross over ethernet cable. It was also possible to start the master and model processes from the MacBook via SSH.

The system load on the Raspberry Pi was monitored with the "top" command line tool. On the Raspberry Pi, the Master occupied 32 % and the model 17 % of the processor capacity. The network load, measured on the MacBook with Activity Monitor, was only at 26 kByte/s for incoming data and at 9 kByte/s for outgoing data.

In that setup, not a single real time violation was reported, even not when the watchdog timer of the model was set to 0.011 s. A screenshot of the "Time Plot" widget for this setup is shown in Fig. 4.20. Note that the range of the vertical axis compared to Fig. 4.19 is smaller. Even under heavy load on the Raspberry Pi, by opening other applications, the values kept within that low magnitude. Note that the workload time does not show the real workload. E.g. for the slave nodes, the corresponding *QElabsedTimer* for this measurement is started after the bytes of the timeout message from the master were received and packed to a *QPlcMessage*. Before that, also the Qt events are dispatched in the event queue. So this overhead is not measured. Additionally, although the *QElabsedTimer* is stopped after the answer message was built and sent, there is still some overhead by returning from the functions and other Qt or operating system routines. This is similar for the master. This all explains the discrepancy between the measured system load and the measured workload times with the "top" command line tool (divided by the control cycle time of 0.010 s).

After that test, the master and the model processes on the Raspberry Pi were also started without real time priority. In that case occasional real time violations in the magnitude of

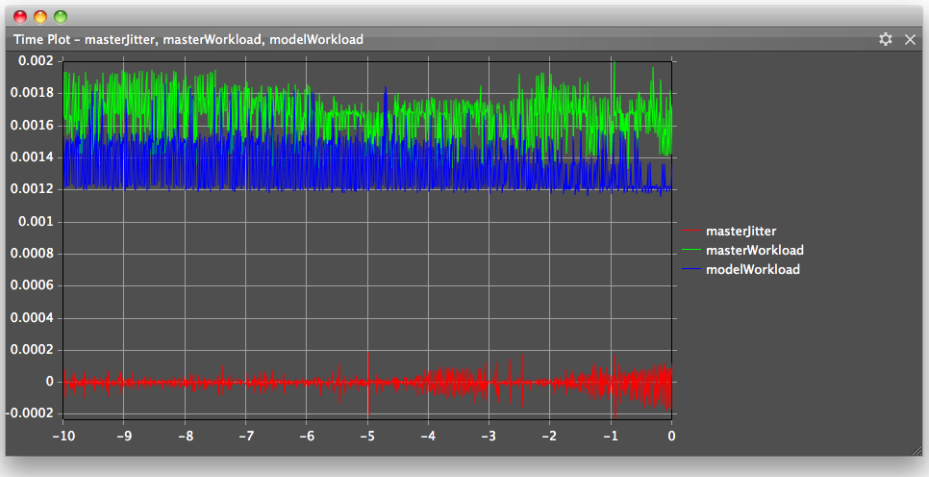


Figure 4.20: Screenshot of the "Time Plot" widget displaying the master jitter, master workload and model workload times while master and model ran with real time priority on a Raspberry Pi with a Preempt-RT patched Raspbian with the 3.8.13-rt14+ kernel.

seconds occurred, similar to the result from the last section. So again, in more than 99 % of the time, the real time constraints were met.

4.4 Simulation results

Important results of the simulations are already illustrated in Fig. 4.18 on p. 77, in particular in the "Time Plot" widgets on the bottom. Here, a uniform wind field was simulated with wind velocity vector $\boldsymbol{v}_\mathrm{w}^\mathrm{e} = (5\,\mathrm{m/s}, 3\,\mathrm{m/s}, 0)^\top$. To reach the maximum power, the tether reel out speed was set according to Eq. (2.15) on p. 23 to

$$v_\mathrm{t,out} = \frac{1}{3}|\boldsymbol{v}_\mathrm{w}| = \frac{1}{3}|(5\,\mathrm{m/s}, 3\,\mathrm{m/s}, 0)^\top| \approx 1.9\,\mathrm{m/s}$$

while pressing the "W" key. All the other values of the "Keyboard Controller" widget which are applied while pressing the other keys were chosen freely, see Fig. 4.18.

The kite was flown by hand: In the reel out phase the kite was steered in figure eights with an optimal pitch angle, i.e. a pitch angle so that the plotted AWE coefficient in the right "Time Plot" in Fig. 4.18 became approximately maximal. At that optimal pitch angle, the magnitudes of the aerodynamic values can be explained by the aerodynamic model in Fig. 4.4 on p. 44. The the drag and lift coefficients in dependency of the angle of attack are plotted again in Fig. 4.21 where the AWE coefficient c_AWE as in Eq. (4.7) on p. 51 is added. Its maximum is $\hat{c}_\mathrm{AWE} = 17.54 \approx 18$ at $\alpha \approx 22\,°$. The power output in the reel out phase was around $P \approx 2000\,\mathrm{W}$. The periodic variations had the period time of the time that was needed to fly a half figure eight. So the variations result from flying of curves in the figure eight, which always adds losses as mentioned in Sec. 2.2.4 on pp. 23. The magnitude of the power can also be explained with the maximum power extraction as in Eq. (2.11) on p. 21. With air density $\rho = 1.2\,\mathrm{kg/m^3}$ and kite

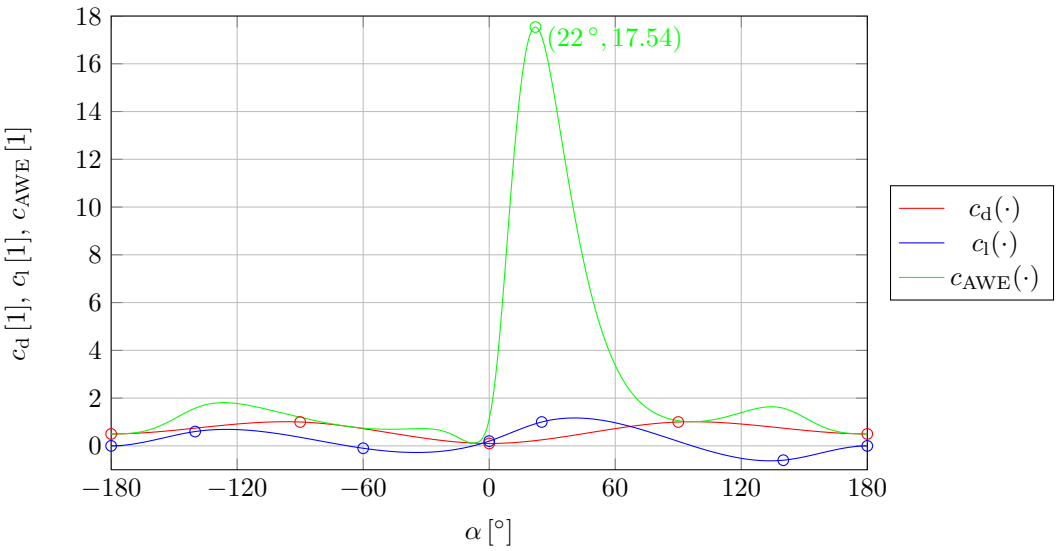


Figure 4.21: Drag, lift and AWE coefficients in dependency of the angle of attack.

area $A = 10\,\mathrm{m}^2$, the theoretical maximum power is

$$\begin{aligned}\hat{P} &= \frac{2}{27}\rho v_\mathrm{w}^3\, A\, \frac{c_\mathrm{ae}^3}{c_\mathrm{d,i}^2} \\ &= \frac{2}{27}\rho |\boldsymbol{v}_\mathrm{w}|^3\, A\, c_\mathrm{AWE} \\ &= \frac{2}{27}\cdot 1.2\,\mathrm{kg/m}^3 \cdot |(5\,\mathrm{m/s}, 3\,\mathrm{m/s}, 0)^\top|^3 \cdot 10\,\mathrm{m}^2 \cdot 17.54 \\ &\approx 3091\,\mathrm{W}.\end{aligned}$$

The losses can be explained as follows: Firstly, during the simulation the average elevation angle of the kite was $\beta_\mathrm{e} \approx 20\,°$. Secondly, the kite was modeled with mass $m_\mathrm{b} = 5\,\mathrm{kg}$. In the derivation of the maximum extractable power both values were assumed to be zero. But in the left "Time Plot" widget in Fig. 4.18 on p. 77 the positive peaks of the power reached $\approx 3\,\mathrm{kW}$ which confirms the whole model. In a later simulation the kite was flown through the maximum power point in the reel out phase, where $P = 3091\,\mathrm{W}$ was reached quite exactly.

The tether length was kept between $l_\mathrm{t} \approx 100 \ldots 200\,\mathrm{m}$. In the reel in phase the kite was pitched to $\beta_\mathrm{e} \approx -90\,°$ such that the total aerodynamic force was minimized while the altitude of the kite kept approximately constant. In this phase only a power of $P \approx 200\mathrm{W}$ was consumed. The whole flight path for about 1.5 pumping cycles was visible in the two lower "3D Views" on the right side of Fig. 4.18 on p. 77.

It was interesting to see in the right "Time Plot" widget of Fig. 4.18 that the angle of attack and thus drag, lift and AWE coefficient kept approximately constant. So if the kite would have been steered without a change of the pitch angle, an even simpler aerodynamic model which uses constant values for the drag and lift coefficients would have been sufficient. Such a model is actually often used for analytical derivations and was also tested successfully in another simulation.

In the theory in Sec. 2.1.2 on pp. 14, it was stated, that rigid airplanes have a higher lift to drag ratio than flexible wings because of a lower drag coefficient. The force equilibrium of

Fig. 2.7 on p. 22 with the marked similar triangles implies that the magnitude of $-\boldsymbol{v}_{\mathrm{b,r}}$ would be higher with a lower drag coefficient. This was tested in the implemented Simulink model by multiplying the final drag coefficient with 1/5. In that simulation, as expected the wing flow around 5 times faster.

When the wing was flown close to or into the zenith, it showed unrealistic behavior. This results from the calculation of the elevation and azimuth angles which are used for the wing's orientation. There is a singularity, because all azimuth angles are valid for the zenith position. One way to avoid that is to change the rotation order [1, pp. 141]. However, this would only shift the singularity into another position. Because the singularity hardly troubles the simulations and would imply bigger efforts to solve, it is kept untreated for now.

5 Conclusions

In this master thesis, the relatively new AWE technology was presented with special focus on crosswind and lift AWE. It uses a comparably low amount of parts, namely a ground based electrical machine, a tether and a wing. Both, the theoretical power density and the already built prototypes by researchers and companies, are promising and it seems to be possible to achieve at least similar electrical power outputs as in classical wind energy. Some technological aspects e.g. rigid and flexible wing concepts and the different tether concepts, were outlined. A simple nested control strategy which consists of an orientation controller, course controller and power controller was discussed. The key sensors and key actuators are the electrical machines with current, angular speed and angular position sensors. Additionally, sensors for the position of the wing, e.g. tether angle sensors or an IMU mounted at the wing, are needed. Since sensors and actuators may be placed at different locations, a distributed PLC is crucial.

In order to build a research prototype, a cost effective and open source PLC approach was proposed in this thesis: The PLC was designed according to the KISS principle and uses only ordinary and inexpensive embedded computers like the Raspberry Pi or BeagleBone Black. They are operated with a Preempt-RT patched real time Linux. With this real time operating system it is possible to run any application with real time priority and so the Qt framework with the Qt Creator IDE was proposed to be used as base. The communication of the nodes of the distributed PLC was implemented with standard IP links and a simplistic master-slave communication scheme, that does not require time synchronization. To make the development as simple as possible and to encapsulate the repeating tasks of the distributed PLC, the QtPLC library was developed with special focus on a simple and intuitive API and on low communication overhead. QtPLC also contains the QtPLC Control Center GUI API. The presented QtPLC API is the result of a couple of iterations and became an intuitive way to write a node of a distributed PLC in C++. Together with the modular and extensible QtPLC Control Center this is a key result of this thesis.

The presented PLC approach was evaluated according to the actual requirements for a PLC for a lift AWE research prototype. All specifications are evaluated as fulfilled, but the real time ability with the non-real time ethernet links have to be proven in the final system. Additionally, this approach was compared to the quite different solutions used by other AWE researchers and companies. Most similarities were found with the solution of the TU Delft.

A simple lift AWE plant was modeled where the tether is considered as ideal spring and the wing as mass point with an aerodynamics model based on lift and drag coefficients. The model was implemented in Matlab/Simulink, exported to C++ and used in an example QtPLC project. The project contains three applications, a master, a model and the QtPLC Control Center with which data was visualized and the wing was controlled with the keyboard. The implementation was documented in detail with background information of the QtPLC library. The three applications were tested with two test setups: Firstly, all processes ran on a MacBook and secondly, the master and model processes ran on a Raspberry Pi with a Preempt-RT patched real time Linux while the QtPLC Control Center ran on a MacBook. Here, the communication was established with a cross over ethernet cable. The control cycle time was 0.010 s. In the first setup, in more than 99 % of the time the real time requirement of a maximum jitter of +0.005 s was met. In the second setup, not a single real time violation even with a maximum jitter of

+0.001 s was noticed. Additionally, the qualitative and quantitative behavior of the model was understandable.

The simulation worked well and may be extended and used for further studies. In conclusion, the proposed PLC approach together with the QtPLC library seems to be a suitable solution for both, simulations and a PLC for a research prototype.

6 Outlook

The functionality of the QtPLC library was proven with the example lift AWE plant implementation. However, the QtPLC library is far away from being perfect and there is a list of tasks to be done: E.g. the API documentation needs to be finished. The next development steps include the implementation of other underlying protocols, in particular UDP, and other communication schemes, e.g. with a lower dead time. The jitter time measurements and real time violation detections should be refined. There are also some ideas for improvement of the QtPLC Control Center but they are time consuming, e.g. the use of several screen tabs in the main window and saving the screen configuration in a distinct document file.

The next steps for the QtPLC approach will also focus on the hardware side, i.e. interfacing the embedded computers and QtPLC, respectively, with sensors and actuators. For this the QtPLC API should be extended in the same cross platform style as Qt itself: E.g. there should be only one CAN bus API for the different embedded computers. The actual implementation may differ between the embedded computers, but the correct implementation should be selected internally in QtPLC for the used embedded computer at compile time. For a hardware setup with sensors and actuators, also a jitter measurement campaign would be due in order to prove the real time ability. The Linux was not further optimized so far except of installing the Preempt-RT patch. Gaining lower latencies should be possible by slimming the Linux to the necessary applications. Additionally, the QtPLC library should be compared to the TU Delft's framework, when it is published, and to the OROCOS framework. From that new improvements may be gained or parts of those frameworks may be used.

The implemented lift AWE model may be used for further investigations. With electrical machine models, the use of empirical models for the yaw rate of the wing as in [5] and [31] and with the use of more detailed tether models as in [32], the whole system model would become quickly much more realistic and can be used to test controllers. The model is also easily extensible to a drag AWE plant by adding an additional drag force on the wing in Matlab/Simulink.

Bibliography

Direct citations are marked by citation marks ("...") except if the citation is a number. For those citations, the corresponding reference marks (in square brackets, [...]) are placed straight behind. Indirect citations are marked by reference marks as follows: Reference marks before a full stop relate to the sentence. Reference marks behind a full stop relate to everything before that reference mark up to the last reference mark of an indirect citation or the beginning of the paragraph. If direct citations are not echoed exactly, changes are marked inside square brackets.

[1] Ahrens, Uwe; Diehl, Moritz; Schmehl, Roland (Eds.): "Airborne Wind Energy", Springer, 2014.

[2] SkySails GmbH: "Advantages". Webpage, http://www.skysails.info/english/skysails-marine/skysails-propulsion-for-cargo-ships/advantages/, accessed: November 06, 2013.

[3] Digia: "Qt". http://qt.digia.com, http://qt-project.org, accessed: December 09, 2013.

[4] Loyd, Miles: "Crosswind Kite Power". Journal of Energy, vol. 4, no. 3, pp. 106-111, 1980.

[5] Jehle, Claudius: "Automatic Flight Control of Tethered Kites for Power Generation". Diploma thesis, Technische Universität München, May 2012.

[6] Breukels, J.: "An engineering methodology for kite design". PhD thesis, Delft University of Technology, 2011. http://repository.tudelft.nl/assets/uuid:cdece38a-1f13-47cc-b277-ed64fdda7cdf/Thesis_jeroen_breukels.pdf, accessed: Mai 08, 2013.

[7] Fechner, Uwe; Schmehl, Roland: "High level control and optimization of kite power systems". 8th PhD Seminar on Wind Energy in Europe, Zurich, Switzerland, September 12-13, 2012. http://www.kitepower.eu/images/stories/publications/fechner12a.pdf, accessed: November 23, 2013.

[8] Katz, Joseph; Plotkin, Allen: "Low-Speed Aerodynamics". Cambridge University Press, 2001.

[9] Fechner, Uwe: "Dynamic Kite Power System Modelling". Airborne Wind Energy Conference 2013, Berlin, Germany, September 10-11, 2013. http://www.awec2013.de/pdfs/Uwe_Fechner.KitepowerModelling.pdf, accessed: December 04, 2013.

[10] SkySails GmbH: "2. Demonstrator (1 MW)". Webpage, http://www.skysails.info/english/power/development/2-demonstrator-1-mw/, accessed: November 26, 2013.

[11] Fechner, Uwe; Schmehl, Roland: "Design of a Distributed Kite Power Control System". IEEE Multi-Conference on Systems and Control, Dubrovnik, Croatia, 3.-5.10.2012. http://www.kitepower.eu/images/stories/publications/fechner12b.pdf, accessed: November 26, 2013.

[12] Weidauer, J.: "Elektrische Antriebstechnik: Grundlagen, Auslegung, Anwendungen, Lösungen". Wiley, 2013.

[13] Preempt-RT: “Frequently Asked Questions”. https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions, accessed: November 18, 2013.

[14] Preempt-RT: “RT PREEMPT HOWTO”. https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO, accessed: November 27, 2013.

[15] GL Garrad Hassan: “Bladed Multibody dynamics V4”. Brochure, 2013. http://www.gl-garradhassan.com/assets/downloads/Bladed_Brochure.pdf, accessed: November 24, 2013.

[16] Embedded Linux Wiki: “BeagleBone”. http://elinux.org/BeagleBone, accessed: November 24, 2013.

[17] Embedded Linux Wiki: “RPi Hardware”. http://elinux.org/RPi_Hardware, accessed: November 24, 2013.

[18] Bauer, Florian: “Optimierung des Pitchantriebsreglers der AREVA Wind M5000”. Bachelor thesis, Fachhochschule Bielefeld/AREVA Wind, 2012.

[19] Lunze, Jan: “Regelungstechnik 2: Mehrgrößensysteme, Digitale Regelung”. Springer, 2008.

[20] Brown, Dr. Jeremy H.; Martin, Brad: “How fast is fast enough? Choosing between Xenomai and Linux for real-time applications”. Twelfth Real-Time Linux Workshop, Nairobi, Kenya, 25-27.10.2010. https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf, accessed: November 18, 2013.

[21] Buttazzo, G.C.: “Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications”. Springer, 2011.

[22] Ciocarlie, Horia; Simon, Lavinia: “Definition of a High Level Language for Real-Time Distributed Systems Programming”. EUROCON 2007 – The International Conference on “Computer as a Tool”, pp. 828-834, 2012.

[23] Rostedt, Steven: “Intro to Real-Time Linux for Embedded Developers”. Interview on “linux.com”, 21. March 2013. https://www.linux.com/news/featured-blogs/200-libby-clark/710319-intro-to-real-time-linux-for-embedded-developers, accessed: August 07, 2013.

[24] Halang, W.A.; Sacha, K.M.: “Real-time Systems: Implementation of Industrial Computerised Process Automation”. World Scientific, 1992.

[25] Barbalace, A. et al.: “Ethernet Real-Time Communication for Distributed Plasma Control Systems”. Fusion Engineering and Design, Volume 83, Issues 2–3, Pages 520–524, April 2008. Poster: http://202.127.204.30/d07_new/d07_old/6thiaea/pdf/POSTER/P1-40-Luchetta.pdf, accessed: November 18, 2013.

[26] Kiszka, Jan; Wagner, Bernardo; Zhang, Yuchen; Broenink, Jan: “RTnet – A Flexible Hard Real-Time Networking Framework”. 10th IEEE International Conference on Emerging Technologies and Factory Automation, Catania, Italy, September 19.-22, 2005.

[27] Lee, Kyung Chang; Lee, Suk: “Performance evaluation of switched Ethernet for real-time industrial communications”. Computer Standards & Interfaces, Volume 24, Issue 5, November 2002, Pages 411-423.

[28] Soetens, Peter: “The Orocos Component Builder’s Manual”. http://people.mech.kuleuven.be/~orocos/pub/stable/documentation/rtt/current/doc-xml/orocos-components-manual.html, 2007. Accessed: December 04, 2013.

[29] Siciliano, B.; Khatib, O.: “Springer Handbook of Robotics”. Springer, 2008.

[30] Reddy, M.: “API Design for C++”. Elsevier Science, 2011.

[31] Erhard, Michael; Strauch, Hans: “Control of Towing Kites for Seagoing Vessels”. IEEE Transactions on Control Systems Technology, Volume 21, Issue 5, pp. 1629-1640, September 2013.

[32] Williams, P.; Lansdorp, B.; Ockels, W.J.: “Modeling and Control of a Kite on a Variable Length Flexible Inelastic Tether”. AIAA 2007-6705, AIAA Modeling and Simulation Technologies Conference and Exhibit, Hilton Head, SC, USA, August 20-23, 2007. http://www.kitepower.eu/images/stories/publications/williams07e.pdf, accessed: December 09, 2013.