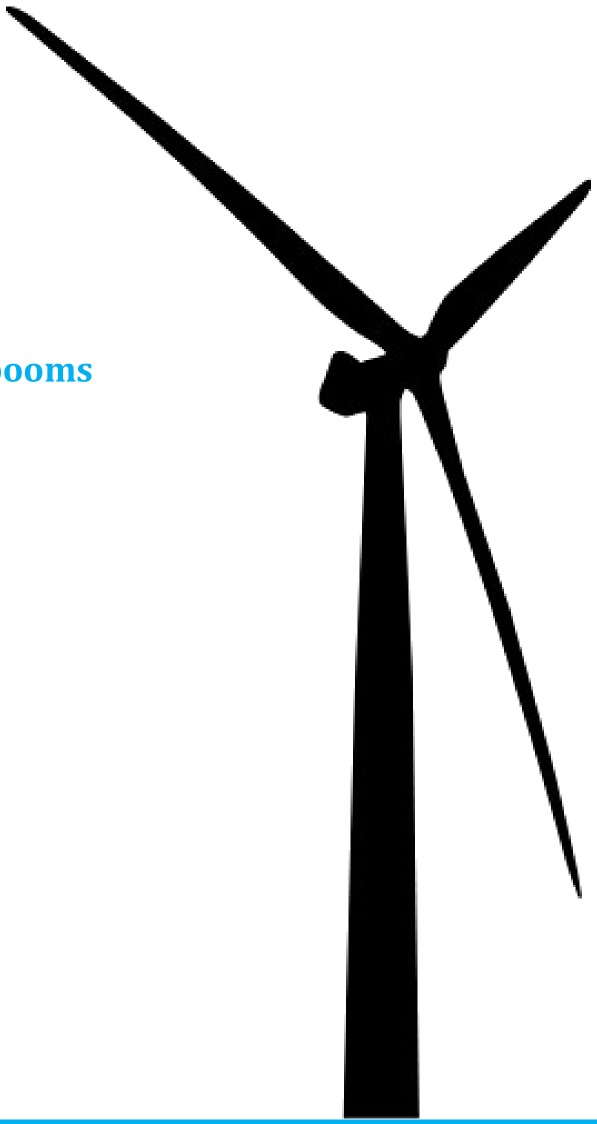
Python's wind turbine design package manual



Author: Dennis van Dommelen

September 2013

Supervisor: Dr. ir. W.A.A.M Bierbooms

Delft University of Technology

Contents

Nomenclature

Latin Symbols

Symbol	Explanation	Unit
a	Induction factor	$[-]$
C	Coefficient	$[-]$
c	Chord length	$[m]$
D	(Drag) force	$[N]$
d	Damping coefficient	$N \cdot s/m$
J	Inertia	$[kg \cdot m^2]$
k	Stiffness	$[N/m]$
L	Lift force	$[N]$
M	Torque/moment	$[N \cdot m]$
m	Mass	$[kg]$
P	Power	$[W]$
R	Resultant force ($\vec{L} + \vec{D}$)	$[N]$
R	Blade total radius	$[m]$
r	Blade local radius	$[m]$
s	Laplace parameter	$[-]$
V	Wind speed	$[m/s]$
W	Resultant velocity	$[m/s]$
x	Tower top displacement	$[m]$

Greek Symbols

Symbol	Explanation	Unit
α	Angle of attack	$[^\cdot]$
β	Flapping angle	$[^\cdot]$
ϵ	Torsion angle transmission	$[-]$
ζ	Damping coefficient	$N \cdot s/m$
η	Efficiency	$[-]$
θ	Pitch angle	$[^\cdot]$
λ	Tip speed ratio	$[-]$
ρ	Air density	$[kg/m^3]$
υ	Transmission ratio	$[-]$
φ	Angle of inflow	$[^\cdot]$
Ω	Rotor angular velocity	$[rad/s]$
ω	Frequency	$[rad/s]$

Subscripts

Subscript	Explanation
ax	Axial direction
b	Flap direction
β	flap direction
d	drag (2D)
g	generator
l	lift (2D)
n	Natural (eigen)/nominal
p	perpendicular
r	rotor/radial direction
sh	Shaft
t	tangential
tot	total

Abbreviations

NACA	National Advisory Committee for Aeronautics

Chapter 1

Introduction

This is the manual which is explaining the Python package for modeling wind turbines. It explains the important files in chapter 2, chapter 3 is explaining how to use the program. The last chapter, chapter 4 shows how an input wind turbine file would look like. The program was originally written in MATLAB, therefor for the MATLAB users appendix C is created, in which the biggest differences between MATLAB and Python is explained. The descriptions given in chapter 2 are copied from the original description file [1] and modified to the Python program.

Chapter 2

Descriptions

In this chapter descriptions of aero.py, bem.py & fun_bem.py, dynmod.py, transfer.py, gener.py and equi.py & fun_equi.py & fun_power.py are given. The other files are self explanatory, this is done by the command lines. The descriptions in this chapter help understand the calculation behind the program.

2.1 Description of aero.py

Determination of the aerodynamic forces, moments and power by means of the blade element method; for known mean wind speed, induction factor etc.
Simplifications:

- uniform flow (i.e. wind speed constant over rotor plane; no yawed flow, windshear or tower shadow)
- no wake rotation (i.e. no tangential induction factor)
- no blade tip loss factor

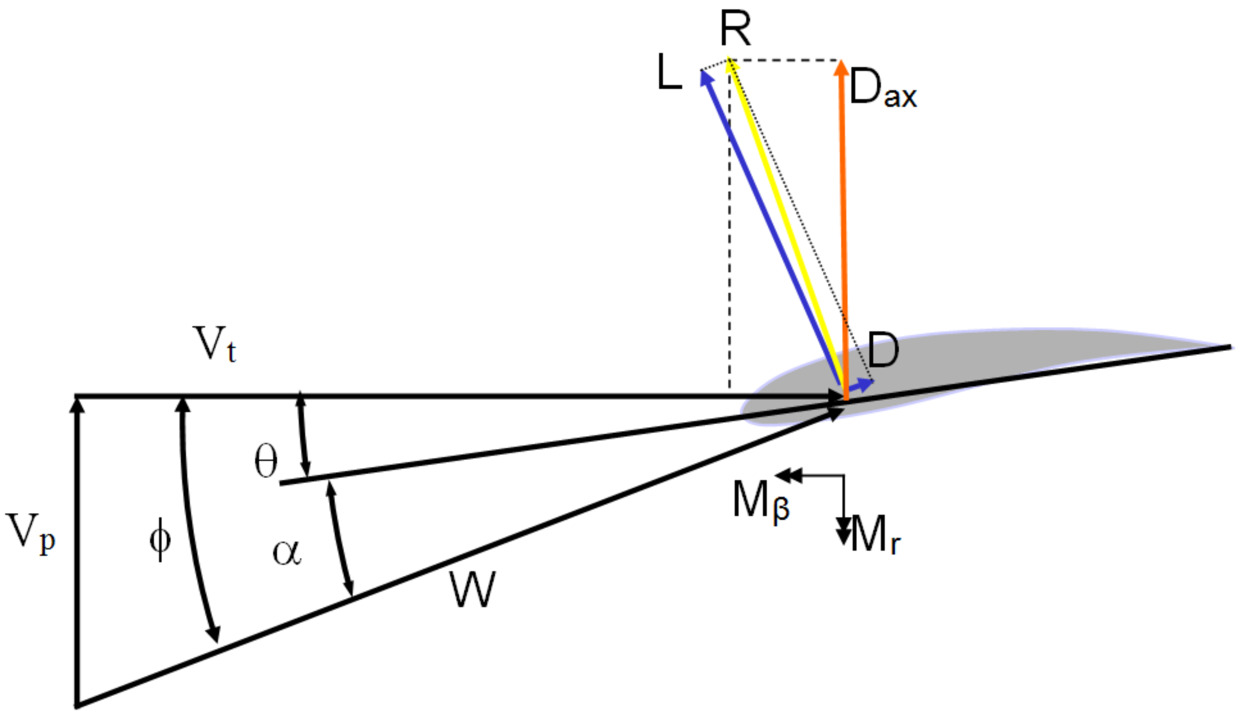


Figure 2.1: Aerofoil forces and velocities at a blade section

V_t: tangential velocity component
V_p: perpendicular velocity component
W: resultant velocity
α: angle of attack
θ: pitch angle
φ: angle of inflow
L: lift force (per definition perpendicular to W)
D: drag force (per definition parallel to W)

R: resultant force
D_{ax}: axial force
M_r: rotor torque
M_β: aerodynamic flap moment

For each blade section (with length dr and chord c) the lift- and drag force equal:

$$L = C_l \frac{1}{2} \rho W^2 c dr \tag{2.1}$$

$$D = C_d \frac{1}{2} \rho W^2 c dr \tag{2.2}$$

with ρ the air density
and Cl and Cd the lift and drag coefficient resp. of the particular aerofoil; they are both functions of the angle of attack α.

The aerodynamic forces depend on the velocities as seen by the rotating blade, so not only the the wind speed is taken into account but also the blade rotation and the velocities due to flexibility of the wind turbine. In our case (see description of dynmod) we have to consider the flapping motion of the rotor blade and the motion of the tower top.

$$V_t = \Omega r \tag{2.3}$$

with Ω the rotational speed of the wind turbine
and r the radial position of the blade section

$$V_p = V(1-a) - \dot{\beta} r - \dot{x} \tag{2.4}$$

with V the undisturbed wind velocity
a the induction factor
β the flapping angle of the blade
x the tower top displacement
For the total forces, moments on a rotor blade the forces over all blade sections have to be add. To obtain the forces and moments on the total rotor the forces, moments over all blades are added.

The (aerodynamic) power equals the product of rotational speed and rotor torque:

$$P = \Omega M_r \tag{2.5}$$

By definition the thrust and power coefficient equal:

$$C_{d_{ax}} = \frac{D_{ax}}{\frac{1}{2}\rho \pi R^2 V^2} \tag{2.6}$$

$$C_P = \frac{P}{\frac{1}{2}\rho \pi R^2 V^3} \tag{2.7}$$

2.2 Description of bem.py & fun_bem.py

Determintion of the aerodynamic forces, moments and power by means of the blade element - momentum method (BEM); for known wind speed, pitch angle, etc.
Simplifications:

- uniform flow (i.e. wind speed constant over rotor plane; no yawed flow, windshear or tower shadow)
- no wake rotation (i.e. no tangential induction factor)

- no blade tip loss factor
- just one annular section (the total rotor plane)

The main problem of rotor aerodynamics is that on one hand the aerodynamic forces depend on the induction factor and on the other hand the induction factor depend on the aerodynamic forces. In order to overcome this problem a combination of two methods are used: blade element method (see aero.py) and momentum theory (see fun_bem.py). According to momentum theory the thrust coefficient equals:

$$C_{D_{ax}} = 4a(1-a) \tag{2.8}$$

with a the induction factor
For values values of the induction factor larger than 0.5 (partial) flow reversal occur so momentum theory no longer can be applied; instead some empiral relation is used. Generally the rotor disc is divided into several angular sections (annuli; in the figure below 3 angular sections are shown), each with its own induction factor.

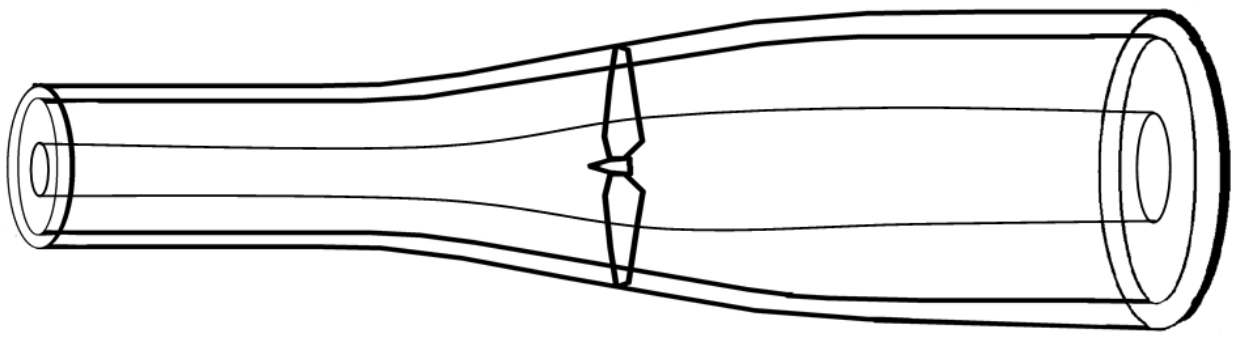


Figure 2.2: Aerofoil forces and velocities at a blade section

For simplicity in fun_bem the rotor disc is treated as just 1 angular section.

In text books BEM is usually explained by an iteration loop for the calculation of the induction factor:

- Choose an initial value for a
- Calculate $C_{D_{ax}}$ with the aid of blade element theory
- From $C_{D_{ax}}$ follows a new value for a, by application of momentum theory (see equation above)
- Continue until a reaches a constant value

In bem.py and fun_bem.py this iteration loop is not directly visible since use is made of the standard scipy.optimize routine fsolve in order to determine the induction factor, for which the thrust coefficient according to blade element theory equals the thrust coefficient according to momentum theory. The difference in thrust coefficient is calculated in fun_bem, so when fsolve determines the value for the induction factor which makes the output of fun_bem equal to zero, the required induction factor is obtained.

2.3 Description of dynmod.py

Equations of motion of wind turbine (dynamic model): time derivatives of the states and outputs of the wind turbine as function of the states and inputs.
The following degrees of freedom are considered (see also the figures).
Flap angle β
Tower top displacement x

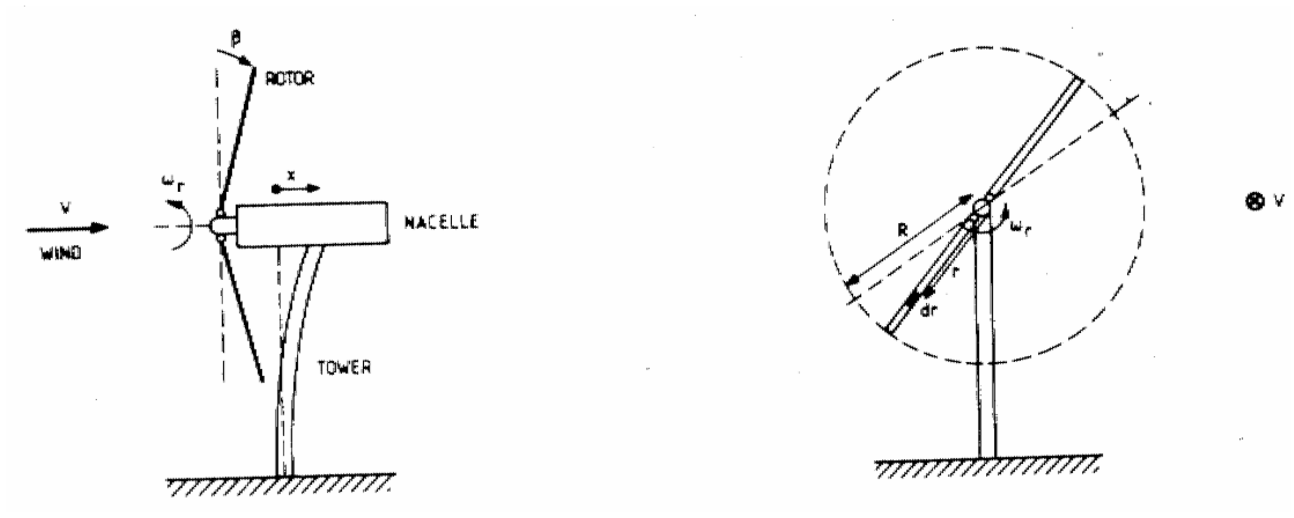


Figure 2.3: Dynamic model wind turbine

Rotor angular velocity $\Omega(\omega r)$
Torsion angle transmission ϵ

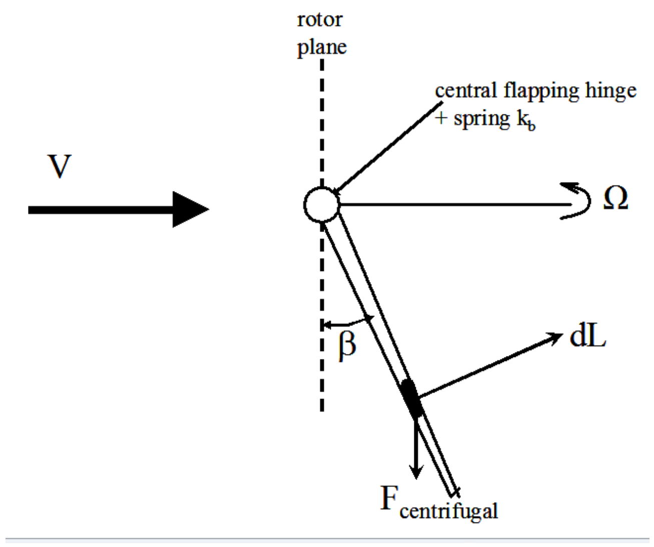


Figure 2.4: Dynamic model rotor blade

Applying several simplifications (e.g. flap angle is small, so $\sin\beta \approx \beta$ and $\cos\beta \approx 1$, gravity is neglected) the dynamics of all subsystem are reduced to 'mass-spring-damper' systems:

$$J_b\ddot{\beta} + (k_b + J_b\Omega^2)\beta = M_\beta \tag{2.9}$$

$$m_t\ddot{x} + d_t\dot{x} + k_t x = D_{ax} \tag{2.10}$$

$$J_r\dot{\Omega} + d_r\dot{\epsilon} + k_r\epsilon = M_r \tag{2.11}$$

$$J_{tot}\ddot{\epsilon} + d_r\dot{\epsilon} + k_r\epsilon = \frac{J_{tot}}{J_r}M_r + \frac{J_{tot}}{\upsilon^2 J_g}\upsilon M_g \tag{2.12}$$

with

$$J_{tot} = \frac{\upsilon^2 J_r J_g}{J_r + \upsilon^2 J_g} \tag{2.13}$$

The equations for the transmission are obtained via reducing the 2-shaft system to an equivalent 1-shaft system (low speed shaft); in doing so the values for the generator angular velocity, generator

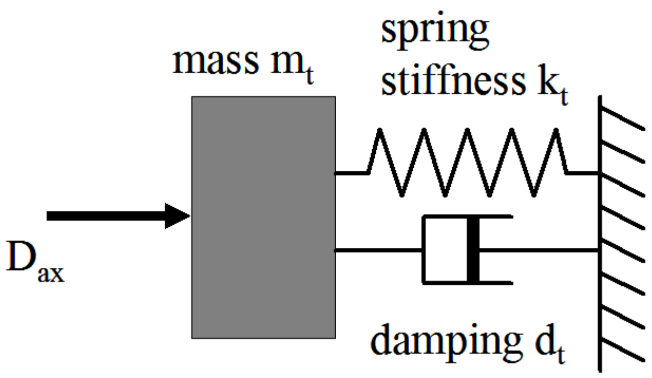


Figure 2.5: Dynamic model tower

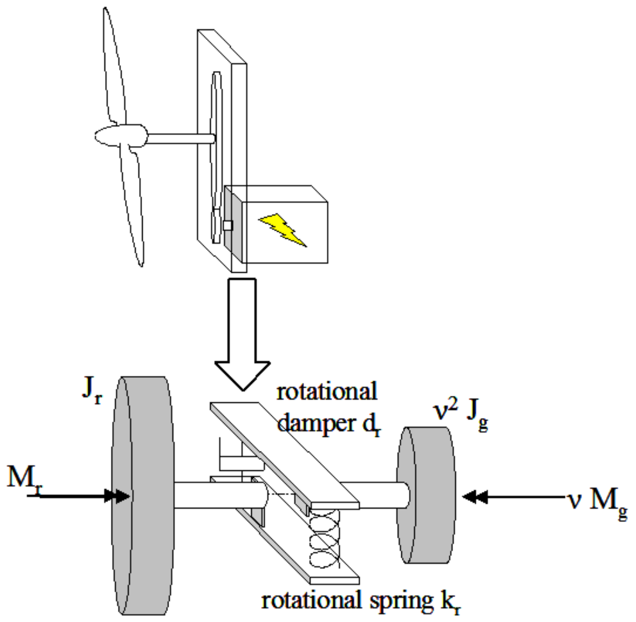


Figure 2.6: Dynamic model transmission; the 2-shaft system is reduced to an equivalent 1-shaft system (low speed shaft)

torque and generator inertia should be properly corrected (depending on the transmission ratio υ).

In dynmod the above 2nd order differential equations are rewritten as two 1st order differential equations. In total 7 differential equations are obtained, defining the 7 the states of the wind turbine:

- Flap angle of rotor blade β
- Flap angular velocity of rotor blade $\dot{\beta}$
- Tower top displacement x
- Tower top speed $\dot{x}$
- Rotor angular velocity Ω
- Torsion angle transmission ϵ
- Torsion angular velocity transmission $\dot{\epsilon}$

The dynamics of the generator and power electronics is much faster than the dynamics of the mechanical part of the wind turbine. This implies that it is justified to model the generator by means of a static 'torque-rotational speed' relation; see gener.py.

2.4 Description of transfer.py

Determination of the transfer function of the wind turbine. The transfer function H(s) relates all wind turbine inputs to all wind turbine outputs

$$H(s) = \frac{NUM(S)}{DEN(S)} \tag{2.14}$$

inputs of wind turbine:

1. blade pitch angle theta [degrees]
2. undisturbed wind speed V [m/s]

outputs of wind turbine:

1. axial force Dax [N]
2. aerodynamic flap moment Mbeta [Nm]
3. aerodynamic rotor torque Mr [Nm]
4. generator power Pg [W]
5. blade pitch angle theta [degrees]
6. undisturbed wind speed V [m/s]

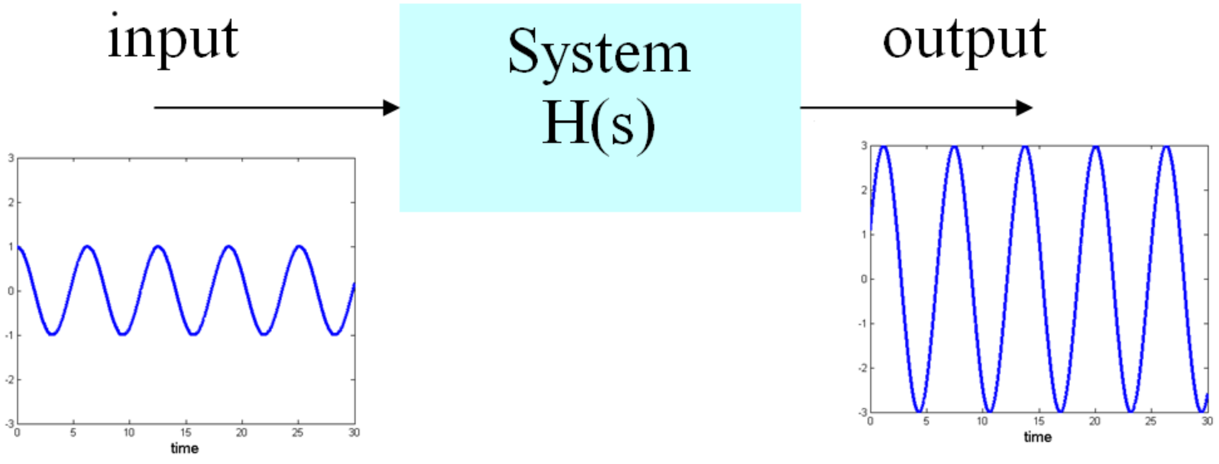


Figure 2.7: Schematic overview of transfer system

The response of a linear system to a harmonic input is again harmonic (with the same frequency). The relation between the output signal and the input signal, for each frequency, is given by H, which is in general a complex number. The absolute value of H is the ratio between the amplitudes of the output and the input; the phase of H is the phase difference between the output and the input. E.g. the transfer function of a second order systeem (eigenfrequency ω_n and critical damping ζ) is:

$$H(\omega) = \frac{1}{\omega_n^2 - \omega^2 + j2\zeta\omega\omega_n} \tag{2.15}$$

Note: generally the transfer function is expressed as function of the Laplace parameter $s = j\omega$ rather than ω.

The transfer function of the wind turbine is determined by linearising the equations of motion as given in dynmod.py. This linearisation could be done analytically by use of Taylor expansion of the equations. In transfer.py it is done numerically by disturbing the inputs and states by a small amount and considering the resulting change in the outputs. The obtained linearised equations can be put in standard form (so-called state space format):

$$\dot{x} = Ax + Bu \tag{2.16}$$

$$y = Cx + Du \tag{2.17}$$

with x the states of the wind turbine:

- Flap angle of rotor blade β
- Flap angular velocity of rotor blade $\dot{\beta}$
- Tower top displacement x
- Tower top speed $\dot{x}$
- Rotor angular velocity Ω
- Torsion angle transmission ϵ
- Torsion angular velocity transmission $\dot{\epsilon}$

and y, u the outputs and inputs of the wind turbine (see above)

Finally the control.matlab module commands ss and tf are used to obtain the transfer function.

Note: numerical linearisation is outside the scope of the wind energy course, so it is not necessary that the listing of transfer.m is totally understood.

2.5 Description of gen.py

Determintion of the generator parameters, torque and power.
Simplifications:

- For a constant lambda, the shaft power is proportional to the rotational velocity to the power 3

The nominal generator velocity is calculated according to the next formula:

$$\omega_n = \nu * \lambda_n * V_n / R \tag{2.18}$$

In this equation, ν is the transmission ratio, λ_n is the nominal tip speed ratio, V_n is the nominal wind speed and R is the rotor radius.
The mechanical shaft power is proportional to the rotational velocity to the 3rd power:

$$P_{sh} = (\frac{\omega}{\omega_n})^3 * \frac{P_n}{\eta} \tag{2.19}$$

Here P_n is the nominal power and η is the generator efficiency.
The electrical generator power is:

$$P_g = (\frac{\omega}{\omega_n})^3 * P_n = P_{sh} * \eta \tag{2.20}$$

The torque is simply the mechanical relation between power and torque:

$$M_g = \frac{P_{sh}}{\omega} \tag{2.21}$$

2.6 Description of equi.py, fun_equi.py, fun_power.py

Determination of the operating point of the wind turbine for known wind speed; this is the steady state after equilibrium between all acting forces on the wind turbine.
Partial load conditions ($V \leq V_n$): blade pitch angle $\theta = \theta_n$ (subscript n stands for nominal conditions).
Note: it is not assumed that the wind turbine automatically operates at optimal tip speed ratio.
Full load conditions ($V > V_n$): rotor rotational speed $\Omega = \Omega_n$; blade pitch θ such that power equals nominal power.

During steady wind conditions any transient response will be damped out so the wind turbine will go to some equilibrium condition (specified by the rotor speed, blade flap angle, etc.), also called operating point. The equilibrium condition is determined by equilibrium of all forces and moments acting on the wind turbine; in general this will depend on the mean wind speed. In partial load there will be an equilibrium between the aerodynamic rotor torque and generator torque. So it would be possible to calculate the rotor speed at equilibrium via an iteration loop during which the rotor speed is varied until the aerodynamic rotor torque equals the generator torque. In equi.py this is again done using the routine from scipy.optimize fsolve in combination with fun_equi.py; in the latter the difference between the aerodynamic rotor torque and the generator torque is calculated for given rotor speed.
At full load conditions the pitch angle is changed such that the power equals nominal power. It is assumed that the pitch control is such that the blade is rotated towards the wind (i.e. towards zero-lift conditions). This time fsolve is used in combination with fun_power.py; the latter calculates the difference between the aerodynamic power, for given pitch angle, and nominal power.

The equilibrium states of the degrees of freedom (i.e. blade flap angle, tower top displacement, rotor angular velocity, torsion angle transmission) are given by the equations of motion (see dynmod.py) in which all terms which are time dependent (i.e. time derivatives) are omitted.

Chapter 3

How to use the program

To use this program, all python files has to be in the same folder. This is because each part of the program is dependent on another part. To make use of the program and generate plots and calculate the wind turbine parameters some demos are created. These demos can help to understand how the program should be used. Also time simulation and responses to different inputs can be created using this program. The use of the program and how to create the simulations is explained in this chapter. This Python program needs some certain extension packages. These packages and an installation manual are found in Appendix A.

3.1 Program overview

When you are using the program for the first time, the amount of files can be vary disturbing. Therefor an overview of the files is given below.

File	**Short explanation**
aero.py	Determination of the aerodynamic forces, moments and power
bem.py	Determination of the induction factor, aerodynamic forces, moments and power
cplambda.py	Determination of Cdax, Cp and induction factor with lambda
drag.py	Drag curve of applied blade aerofoil
dynmod.py	Dynamic model wind turbine
equi.py	Determination of the operating point of the wind turbine for known wind speed
fun_bem.py	Used in bem.py to calculate the induction factor
fun_equi.py	Used in equi.py to calculate omr0 or theta
fun_power.py	Used in powercurve1 to calculate theta
gen.py	Ideal torque-rpm characteristic of a generator including converter
gust1.py	Smooth wind gust
gust2.py	Wind gust with the shape of a sine
lift.py	Lift curve of applied blade aerofoil
powercurve1.py	Determination of the characteristics of a variable speed regulated wind turbine
powercurve2.py	Determination of the characteristics of a constant speed wind turbine
transfer.py	Determination of the transfer function of the wind turbine
Resulting files	
plots.py	Generation of all relevant graphs of a wind turbine
simulation.py	Generation of plots for a time simulation
Wind turbine files	
NREL_5MW.py	NREL_5MW wind turbine parameters
NREL_5mw.txt	txt file generated by NREL_5MW.py
S88.py	S88 wind turbine parameters
S88.txt	txt file generated by S88.py
V90.py	V90 wind turbine parameters
V90.txt	txt file generated by V90.py
V902.py	Modified V90 turbine with resonance at V equal to 8 m/s
V902.txt	txt file generated by V902.py

Demo files	
demo_windsim.py	The demo main file
cplambda_demo.py	Explanation on how to create plots with lambda on the x-axis
powercurve_demo.py	Explanation on how to create plots with the wind speed on the x-axis
BEM_demo.py	Explanation of the calculation of the induction factor

3.2 Demo

The program includes a demo, to run this demo start demo_windsim.py. Then the program gives you 3 options. To select one of the options press the number and after that enter. Option 1 will start the Cp-lambda demo, 2 the powercurve demo and 3 the BEM demo. If one demo has ended, the menu shows again. To exit press 4 or press the cross in the top right corner. In each demo, to proceed press enter or close the plotting window.

3.2.1 Cp-lambda demo

The Cp-lambda demo explains how a plot can be created with lambda on the x-axis. This is done in an interactive way, at first the input parameters of the cplambda function is shown. Then it explains how the list containing all lambda numbers is created, after which you can choose the wind turbine for which the demo has to run. When the wind turbine is chosen, press enter and the plots will appear. Then the demo will explain how to use simple plot commandos to improve the plot.

3.2.2 Powercurve demo

The powercurve demo is quite similar to the cp-lambda demo, except that now the powercurve1 function is used, a list containing all wind speeds is created and with that the power versus wind speed graph is plotted. Also a comparison is made between a fixed speed and a variable speed wind turbine. The fixed wind speed turbine is calculated using the powercurve2 function. It compares the power, axial force and power coefficient for both turbines.

3.2.3 BEM demo

The BEM demo is there to explain the BEM and the methods used for obtaining the induction factor. At first, the input parameters of the function aero are given. Then a wind turbine can be chosen, for which the induction factor will be calculated. Then some input parameters are given, after which 'a' is calculated using the cross section between the thrust coefficient calculated by the BEM method, and the thrust coefficient calculated by the momentum theorem. At first it is visualized in the plot, then it is calculated by the program itself. Note: for some turbines the converging points method cannot be graphically plotted, as the program fails when imaginary numbers are obtained. However a result is given by the calculation method.

3.3 Plots

To plot graphs, a plotting file has been made available. This is plots.py, to use this program, open it in idle and run the program. Then use the command 'plots('Windturbine',True)', with replacing Windturbine with the name of the wind turbine it has to plot. The True can also be replaced by False, this is for plotting multiple thetas (True) or just the nominal theta (False). After pressing enter the program will create the plots, and store these in the folder plots/Windturbine. It is therefor important to create a folder of the wind turbine you want to plot in advance in the plots folder. Otherwise errors will appear! This program might take a minute to create the plots. When the plots are all made, a text message will appear for conformation.
This program plots the parameters as given in the following table:

Input Parameter	Output Parameter
λ	$C_{D_{ax}}$
λ	Cp
λ	a
V	D_{ax}
V	M_β
V	M_r
V	P
V	$C_{D_{ax}}$
V	Cp
V	a
V	θ
V	ω_r

3.4 Simulation

The program is also able to simulate the wind turbines response over time to a simple gust, a sinusoidal gust and a step input. Gust 1 is a smooth wind gust which starts after 5 seconds and has amplitude 1. Gust 2 is wind gust with the shape of a sine; the frequency equals the rotor angular velocity (1P). The simulation is already in the package, simulation.py. This program is built using the python command cm.lsim. This is part of the control.matlab package. The simulation looks at 7 different time responses of the system:

- flap angle [rad]
- flap angular velocity [rad/s]
- tower top displacement [m]
- tower top speed [m/s]
- rotor angular velocity [rad/s]
- torsion angle transmission [rad]
- torsion angular velocity transmission [rad/s]

To use the program simply run it. Then use the following command simulation(windturbine, V, gust=0, time=100, flap=False, tower=False, rotor=False, torsion=False).
The command needs some parameters. Some are set but can also be changed if needed. The following table shows how to use these parameters.
So to plot the complete tower response to a simple gust (gust 1) for a nominal wind speed of 12 and duration of 80 seconds, for the V90 windturbine use the following command: simulation('V90', 12, gust=1, time=80, flap=True, tower=True, rotor=True, torsion=True).

Parameter	Example	Form	Explanation
windturbine	'V90'	string	This is the wind turbine for which the simulation should be done
V	8	integer	Wind speed
gust	gust=1	integer	Type of gust for which the simulation should be done. choose 1 for gust 1, choose 2 for gust 2, other values result in a step input
time	time=100	integer	Time in seconds for the duration of the simulation
flap	flap=True	boolean	If True; plot the response of the flap angle and flap angular velocity
tower	tower=True	boolean	If True; plot the response of the tower displacement and velocity
rotor	rotor=True	boolean	If True; plot the response of the rotor angular velocity
torsion	torsion=True	boolean	If True; plot the response of the torsion angle transmission and the torsion angular velocity transmission

Chapter 4

Wind turbine input file

A wind turbine file is consisting of a matrix, consisting of 4 arrays of different sizes. How a wind turbine file is build up is discussed in this file.

4.1 The writing command

In the beginning of a wind turbine file a document is being opened. This is done by the command:

- document = open('V90.txt', 'w')

Even when the txt file does not exist yet, this command has to be there. The w in the command is important as in the end of the script, the code will be written in the txt file. This txt file should be there, as there is 'easy' support of the data in the program otherwise.
At the end of the wind turbine file, the complete matrix is put together in a string and the writing command is given. After the writing command the closing command is given, the complete code at the end of the wind turbine file looks like:

- text=str(P1)+','+str(P2)+','+str(P3)+','+str(P4)
- document.write(text)
- document.close()

The arrays P1, P2, P3 and P4 are elaborated in the following sections.

4.2 The aerodynamic parameters P1

The array for the aerodynamic parameters should consist of:

Name	Symbol	in Python	unit
Air density	ρ	rho	$[kg/m^3]$
Power loss factor	kp	kp	[-]

This power loss factor is there for the simplifications in BEM.py.

4.3 Turbine parameters P2

The turbine parameters array should consist of the following elements:

Name	Symbol	in Python	unit
Rotor radius	R	R	[m]
Number of blades	N_b	Nb	[-]
Inertia of the blade (w.r.t. flapping hinge)	J_b	Jb	[kgm^2]
Stiffness flap spring	k_b	kb	[Nm/rad]
Equivalent mass tower	m_t	mt	[kg]
Damping tower	d_t	dt	[N/(m/s)]
Stiffness tower	k_t	kt	[N/m]
Transmission ratio	ν	nu	[-]
Inertia rotor	J_r	Jr	[kgm^2]
Damping transmission	d_r	dr	[Nm/(rad/s)]
Stiffness transmission	kr	kr	[N/rad]
Inertia generator	J_g	Jg	kgm^2
Nominal (electrical) generator power	P_n	Pn	W
Efficiency generator	η	eta	[-]

4.4 Blade geometry P3

The blade geometry array is an array which contains 3 other arrays. These 3 other arrays are:

List name	symbol	in Python	Unit
Radial positions	r	r	[m]
Chord	c	c	[m]
Twist	θ_t	thetat	[deg]

The radial positions, chord and twist arrays must have the same size. The values at the blade are given at the borders, thus the size equals the number of blade element sections + 1. To check whether the arrays contain the right amount of elements the following check is used:

- if (len(r) != Ns+1):
-         print 'number of radial positions not correct'
- if (len(c) != Ns+1):
-         print 'number of chord values not correct'
- if (len(thetat) != Ns+1):
-         print 'number of twist values not correct'

4.5 Nominal values P4

The nominal values array consists of the following parameters:

Name	Symbol	in Python	unit
Nominal wind speed	V_n	Vn	[m/s]
Nominal tip speed ratio	λ_n	lambdan	[-]
Nominal blade pitch angle	θ_n	thetan	[deg]

4.6 Creating the txt file

When running the script, automatically a txt file is generated. This file is used by the wind turbine program and simply contains all the arrays and numbers, for example:

```
[1.25, 0.9], [45.0, 3.0, 3900000.0, 128943750.0, 160000.0, 7360.0, 846399.9999999999, 98.0,
11700000.0, 596553.57071587991, 180000000.0, 60.0, 3000000, 0.9], [[4, 6.6, 10.6, 18.5, 30.4,
41, 45], [3.1, 3.9, 3.9, 3.1, 2.1, 1.3, 0.03], [13, 13, 11, 7.8, 3.3, 0.3, 0]], [12.0, 7.8, -1.5]]
```

An example of a complete script is given in Appendix B.

References

[1] Dr. ir. W.A.A.M Bierbooms. descriptions, November 2004.

[2] Inc. company Dice Holdings. Control matlab installation. `http://sourceforge.net/p/python-control/wiki/Download/`. [Online; accessed 03-01-2014].

[3] Python Software Foundation. Python. `http://python.org`. [Online; accessed 15-10-2013].

Appendix A

Required packages and installation

In this appendix the required packages and installation manual are given. The program is written in Python 2.7.5, thus the packages are all compatible with this version. For newer versions of Python it is advised to look if the packages are compatible.

A.1 Required packages

- SciPy/Numpy
- Matplotlib
- Control Matlab

For the basic calculations (array calculations, square roots) the scipy and numpy packages is required. Matplotlib is the package which will enable plotting. The simulation requires the control.matlab package.

A.2 Installation manual

The SciPy/Numpy and matplotlib packages are easy to install using a windows installer. For the Control Matlab it is more difficult. Therefor an installation manual is given below. [2]

A.2.1 Standard Python libraries

In order to run control-python, you must first install some standard python packages:

- SciPy - Open source library of scientific tools: `http://www.scipy.org`
- Matplotlib - Plotting library for python: `matplotlib.sourceforge.net`
- ipython (optional) - interactive python shell: `http://ipython.scipy.org`

A.2.2 Slicot

Some of the underlying functions in python−control are carried out using the [`http://www.slicot.org/SLICOT`] software library. The python−control library uses the [`http://github.com/avventi/Slycotslycot`] python wrapper developed by Enrico Avventi at KTH.
Slycot is only required for functions that make use of SLICOT routines (eg, linear quadratic regulators, Kalman filtering, H_∞ control)
The slycot library is currently under development and the Application Program Interface (API) is not yet fixed. Some errors may occur if the version of slycot and python−control are incompatible

A.2.3 Download

The python-control package can be downloaded from SourceForge `https://sourceforge.net/projects/python-control/files`
The files are distributed as compressed tar files. To unpack and installed, run the following from the command line:

- tar xzf control-N.mx.tar.gz
- cd control-N.mx
- python setup.py install

where N-mx is the latest release (eg, 0.3c).

To see if things are working, you can run the script [`http://www.cds.caltech.edu/~murray/software/python-control/examples/secord-matlab.pysecord-matlab.py`] (using either python or ipython -pylab). It should generate a step response, Bode plot and Nyquist plot for a simple second order linear system.

A.2.4 Installation notes

slycot

To compile for 64 bit architecture on OS X, edit setup.py to include the lines

- extra_link_args=['−arch x86_64']

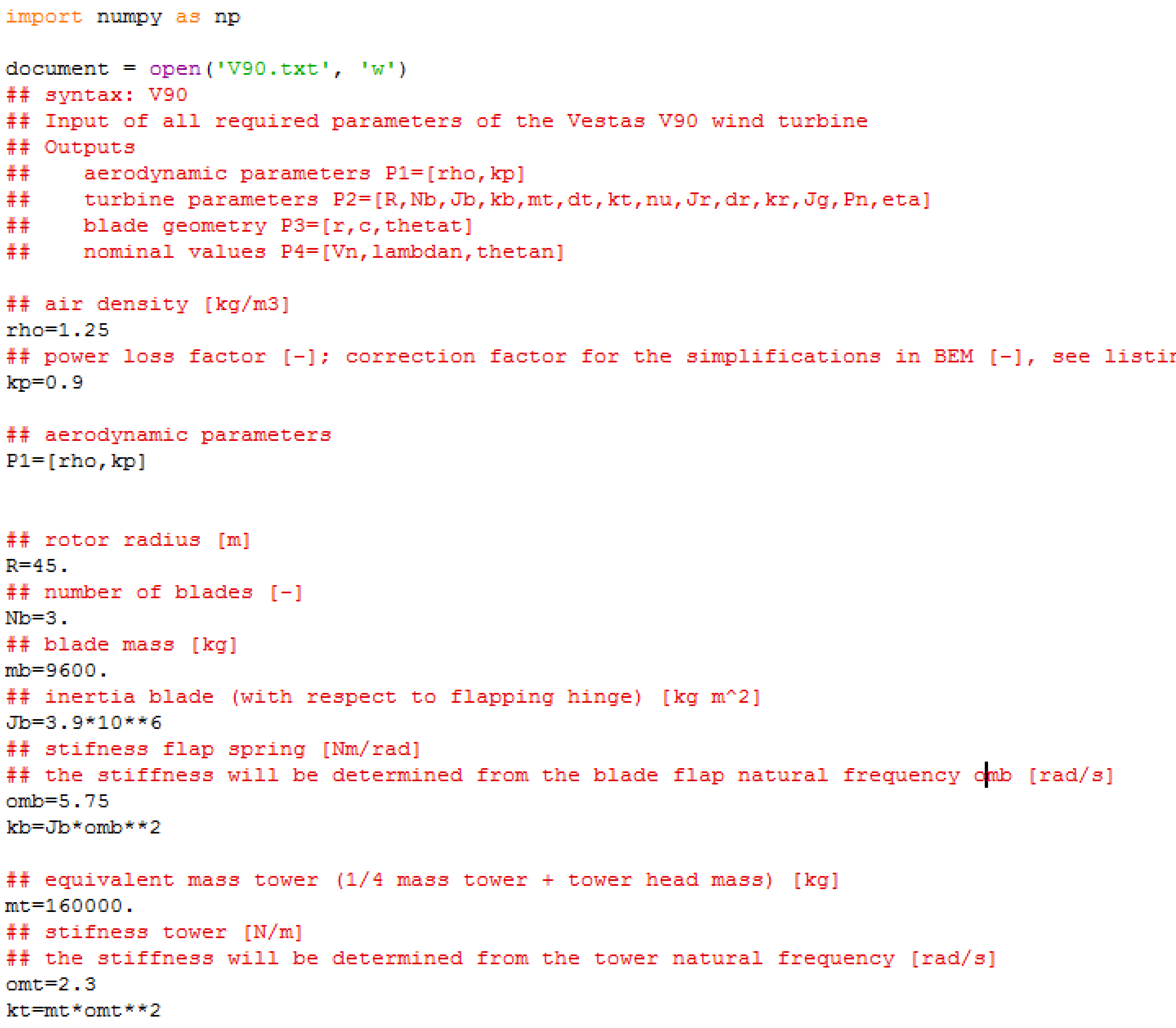
and then run setup.py as

python setup.py config_fc –arch="-arch x86_64" build

Appendix B

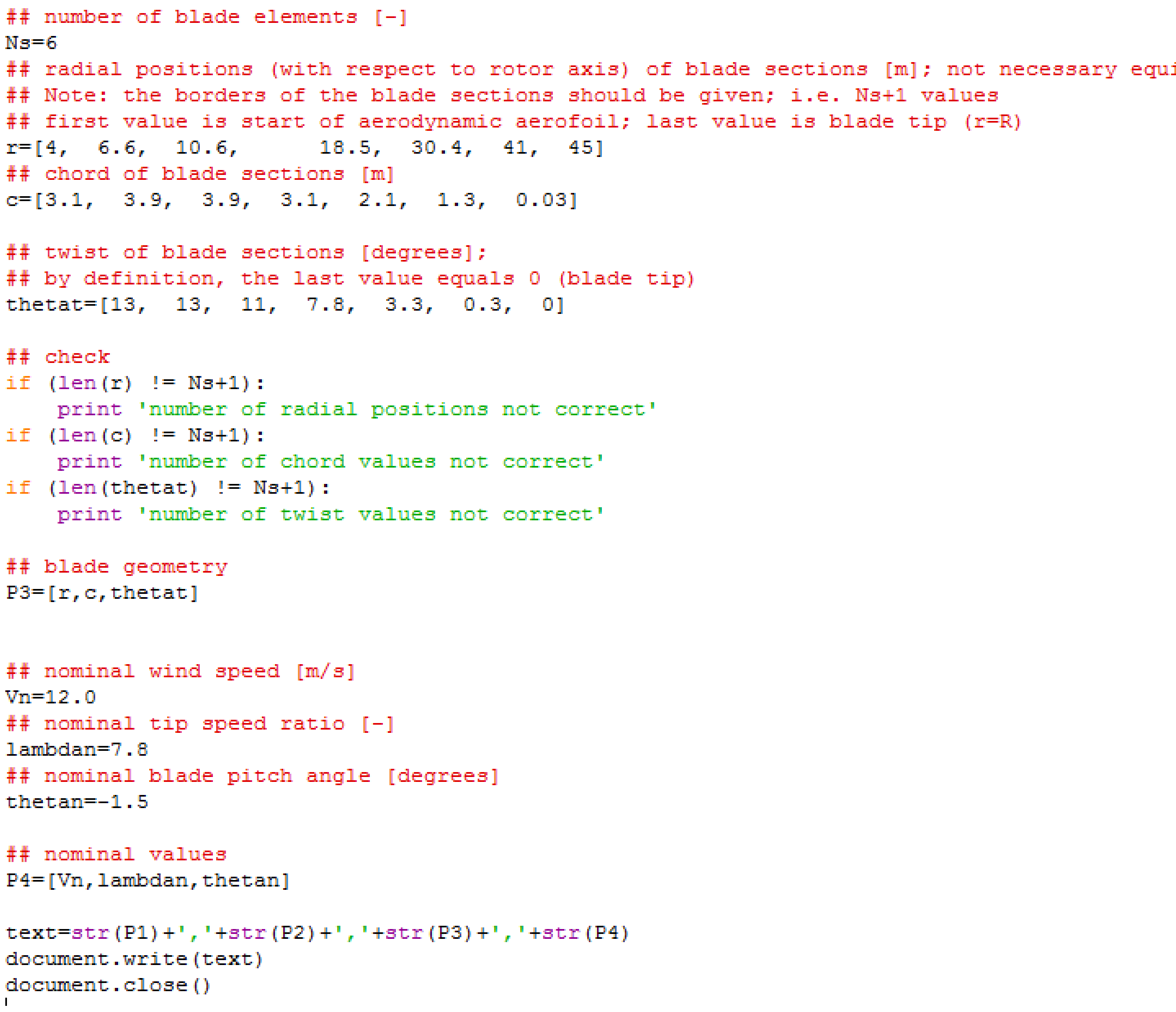
Example of a wind turbine input file

This appendix contains an example of a wind turbine file, it shows exactly how it is build op, as explained in chapter 4.



```
import numpy as np

document = open('V90.txt', 'w')
## syntax: V90
## Input of all required parameters of the Vestas V90 wind turbine
## Outputs
##    aerodynamic parameters P1=[rho,kp]
##    turbine parameters P2=[R,Nb,Jb,kb,mt,dt,kt,nu,Jr,dr,kr,Jg,Pn,eta]
##    blade geometry P3=[r,c,thetat]
##    nominal values P4=[Vn,lambdan,thetan]

## air density [kg/m3]
rho=1.25
## power loss factor [-]; correction factor for the simplifications in BEM [-], see listi
kp=0.9

## aerodynamic parameters
P1=[rho,kp]


## rotor radius [m]
R=45.
## number of blades [-]
Nb=3.
## blade mass [kg]
mb=9600.
## inertia blade (with respect to flapping hinge) [kg m^2]
Jb=3.9*10**6
## stifness flap spring [Nm/rad]
## the stiffness will be determined from the blade flap natural frequency omb [rad/s]
omb=5.75
kb=Jb*omb**2

## equivalent mass tower (1/4 mass tower + tower head mass) [kg]
mt=160000.
## stifness tower [N/m]
## the stiffness will be determined from the tower natural frequency [rad/s]
omt=2.3
kt=mt*omt**2
```

```
## equivalent mass tower (1/4 mass tower + tower head mass) [kg]
mt=160000.
## stifness tower [N/m]
## the stiffness will be determined from the tower natural frequency [rad/s]
omt=2.3
kt=mt*omt**2
## damping tower [N/(m/s)]; 1## critical damping assumed
dt=2*0.01*np.sqrt(kt*mt)

## inertia generator [kg m^2]
Jg=60.
## nominal (electrical) generator power [W]
Pn=3*10**6
## efficiency generator [-]
eta=0.9

## transmission ratio [-]
nu=98.
## inertia rotor [kg m^2]
Jr=Nb*Jb
## stiffness transmission [Nm/rad]
kr=1.8*10**8
## total inertia transmission [kg m^2]
Jtot=(nu**2*Jg*Jr)/(nu**2*Jg+Jr)
## damping transmission [Nm/(rad/s)]; 3## critical damping assumed
dr=2*0.03*np.sqrt(kr*Jtot)

## turbine parameters
P2=[R,Nb,Jb,kb,mt,dt,kt,nu,Jr,dr,kr,Jg,Pn,eta]
```



```
## number of blade elements [-]
Ns=6
## radial positions (with respect to rotor axis) of blade sections [m]; not necessary equi
## Note: the borders of the blade sections should be given; i.e. Ns+1 values
## first value is start of aerodynamic aerofoil; last value is blade tip (r=R)
r=[4,   6.6,   10.6,       18.5,   30.4,   41,   45]
## chord of blade sections [m]
c=[3.1,   3.9,   3.9,   3.1,   2.1,   1.3,   0.03]

## twist of blade sections [degrees];
## by definition, the last value equals 0 (blade tip)
thetat=[13,   13,   11,   7.8,   3.3,   0.3,   0]

## check
if (len(r) != Ns+1):
    print 'number of radial positions not correct'
if (len(c) != Ns+1):
    print 'number of chord values not correct'
if (len(thetat) != Ns+1):
    print 'number of twist values not correct'

## blade geometry
P3=[r,c,thetat]


## nominal wind speed [m/s]
Vn=12.0
## nominal tip speed ratio [-]
lambdan=7.8
## nominal blade pitch angle [degrees]
thetan=-1.5

## nominal values
P4=[Vn,lambdan,thetan]

text=str(P1)+','+str(P2)+','+str(P3)+','+str(P4)
document.write(text)
document.close()
```

Appendix C

Explanation for MATLAB users

There are some differences between Python and MATLAB. Of course, the biggest difference is that Python is an open source program and therefor available for free [3], while MATLAB is a very expensive tool. In this appendix the differences between the code and usage is explained in further details.

C.1 Difference in code

C.1.1 Numbering and loops

The first main difference in code is the numbering, in MATLAB every list starts at 1, where in Python it starts at 0. Also the end of the list is not included in Python. For example: for i in range(0,11,1). This range starts at 0, and contains every number up and including 10.

C.1.2 Variables

All variables in Python are globals. Therefor when using a variable it will keep it's value for the whole program, until it is changed later in the script

C.1.3 Lists and Arrays

The biggest difference between MATLAB and Python is the usage in arrays and lists. If you need for example a calculation with lists or arrays, in MATLAB you are able to use the dot-calculation where in Python you have to use a for loop, see figure C.1. The .calculation multiplies, divides, adds or subtracts the value on the (i,j)th location with the value at the same position in another array.

This also results in a shorter code in MATLAB, it automatically realizes that the outcome variable will be a list or array, therefor it is not necessary to call the variable name in the code before, where this is needed in Python.

C.1.4 Semicolon

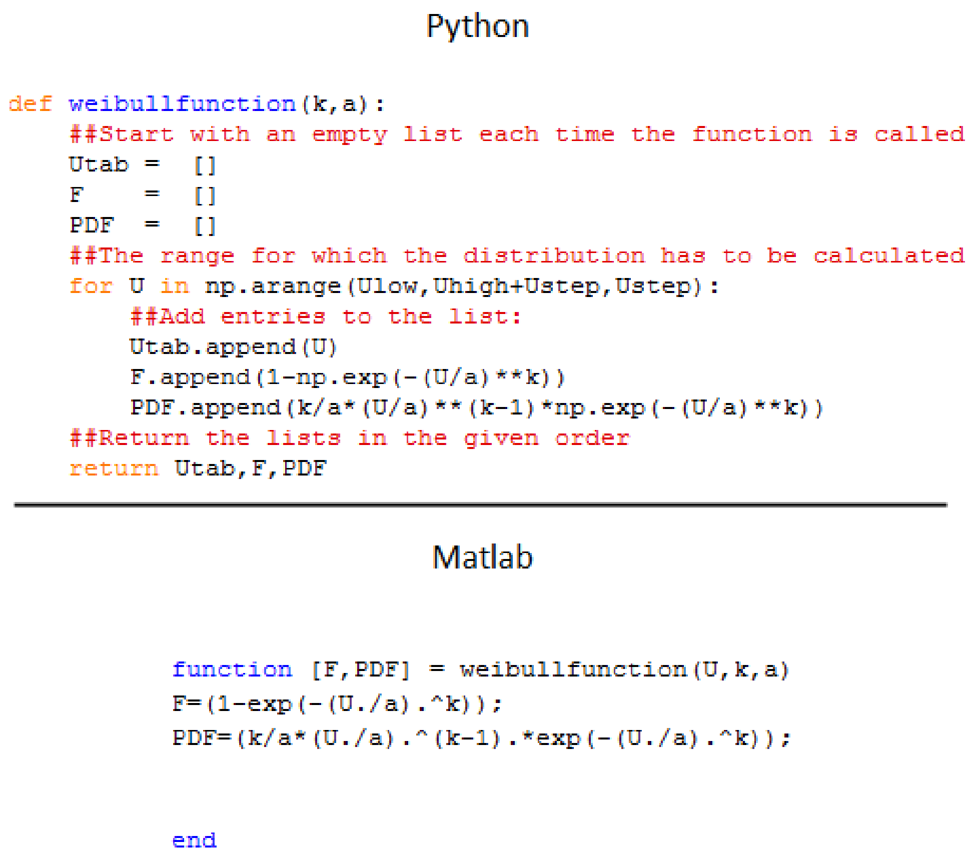
The semicolon is not used in Python to suppress the display of the variable in the 'Python shell'. To display a variable the command 'print' is used, or when using a function the command 'return'.

C.1.5 Packages

In Python there are packages which need to be imported at the beginning of the script. A good example is the Numpy module, where for example the square root function can be called, or the exponent e. In MATLAB these functions are already included. Another example of a package which is used in Python is the control.matlab module. In this module the MATLAB functions for state-space systems, transfer functions and simulations are included.
The benefit of these packages is that you are able to create a complete program or function yourself, this way you are able to tune your Python in the way you want, where in MATLAB you need to pay for each of this package, for Python (most of) these modules are available for free.

C.1.6 Functions and m-files

As mentioned in C.1.5, in Python you are able to create your own functions and import these in another script. If you create multiple functions in one file it becomes a nice module. Of course, in



Python

```python
def weibullfunction(k,a):
    ##Start with an empty list each time the function is called
    Utab =  []
    F    =  []
    PDF  =  []
    ##The range for which the distribution has to be calculated
    for U in np.arange(Ulow,Uhigh+Ustep,Ustep):
        ##Add entries to the list:
        Utab.append(U)
        F.append(1-np.exp(-(U/a)**k))
        PDF.append(k/a*(U/a)**(k-1)*np.exp(-(U/a)**k))
    ##Return the lists in the given order
    return Utab,F,PDF
```

Matlab

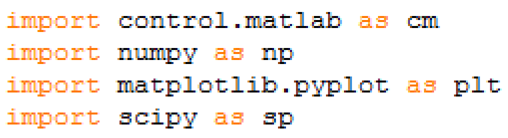
```matlab
function [F,PDF] = weibullfunction(U,k,a)
F=(1-exp(-(U./a).^k));
PDF=(k/a*(U./a).^(k-1).*exp(-(U./a).^k));


end
```

Figure C.1: Python vs Matlab for the creation of a Weibull distribution

MATLAB you are also able to create your own functions by using the m-files. An example of how these packages can be imported in Python is given in figure C.2.

C.1.7 Extiension .pyc

This is a compiled python script. It is automatically created when the original .py file is used in another .py file and this second file is being ran. The python program than uses the data from the .pyc file to run the script.



```python
import control.matlab as cm
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
```

Figure C.2: Importation of packages in Python

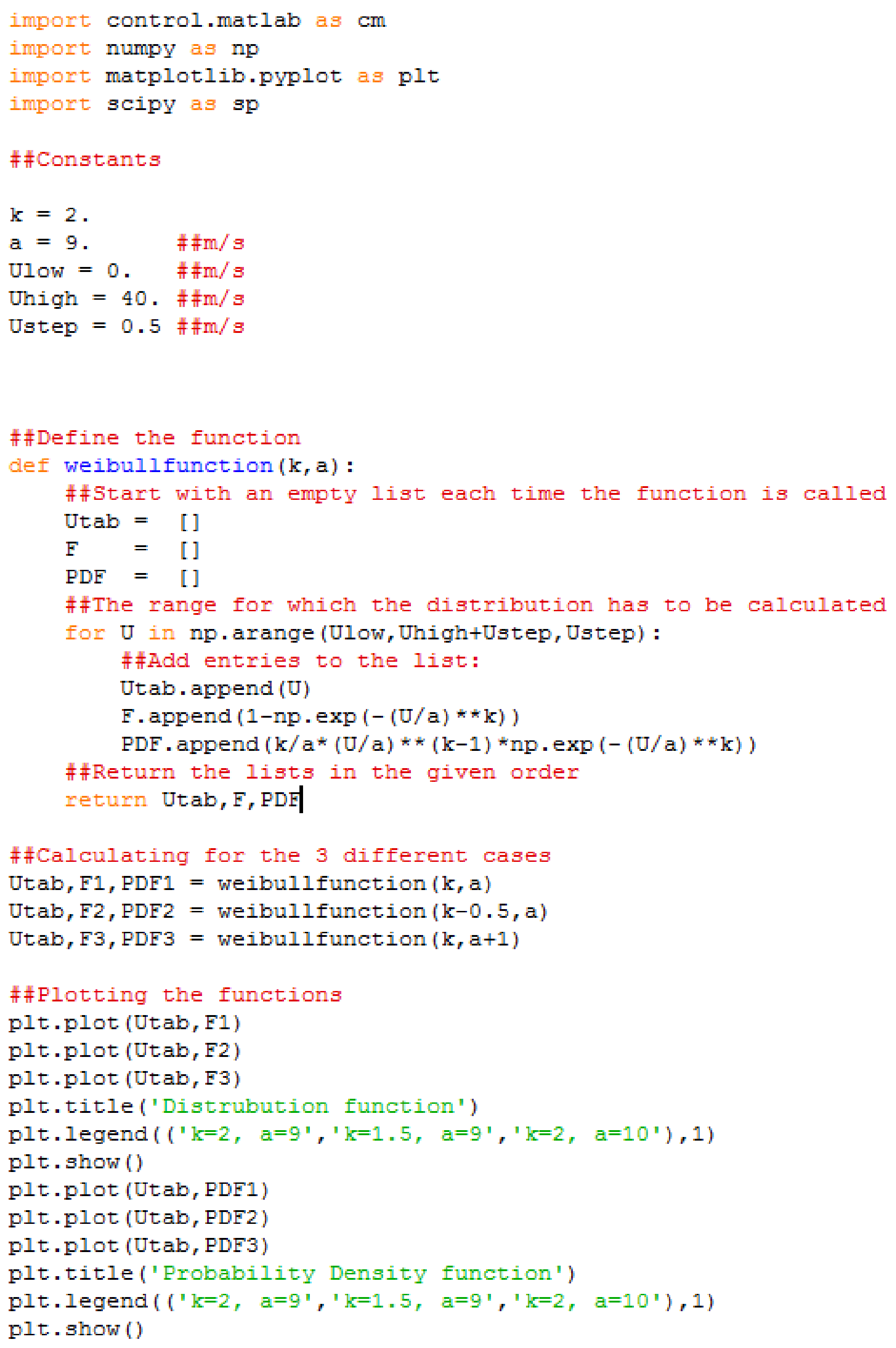
C.2 Difference in usage

Python itself does not have a lot of extensions, it is just the language, but through the packages Python can be made as extensive as you like. MATLAB already includes a lot of these packages. For example with plotting, in Python, you first have to import the matplotlib.pyplot module (see also figure C.2, where MATLAB automatically knows when the plot command is used. The same is valid for all mathematical expressions in Python and MATLAB, MATLAB automatically recognizes the function, where in Python you can import different packages, like Scipy, Numpy or the math module itself.

Another difference in usage is the demo function in MATLAB, Python does not have this function, so it can either be imported or created in for example Pygame. Pygame is an graphical extension of Python, it can be used to do simulations, to create games or for example to create this demo.
When creating your own package, you can use it in another file with the following command:
from FILENAME import function1, function2, function3....
This is not needed in MATLAB, where it automatically uses all m-files in the same directory. When running this script, a FILENAME.pyc file is generated, this is used as the m-file in MATLAB.

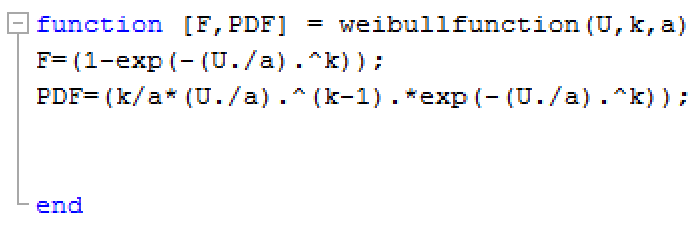
C.3 Weibull distribution

In figure C.1 the Weibull function are compared to each other for both Python and MATLAB. These scripts are given in figure C.3 for the Python version, and C.4 and C.5 for the MATLAB versions.



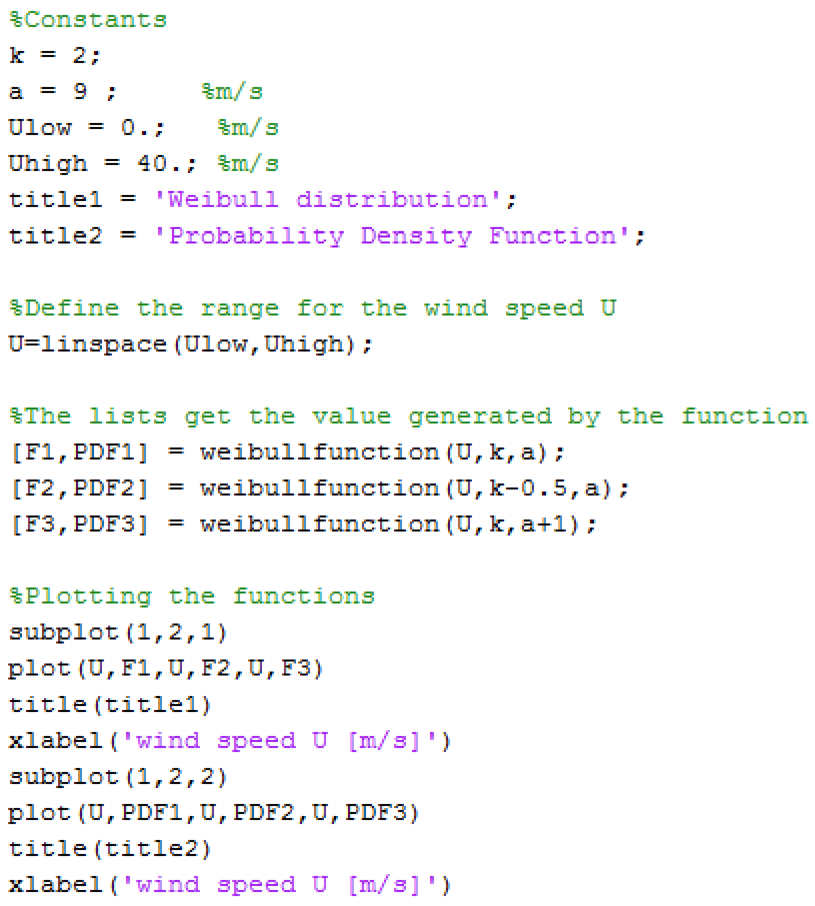
```
import control.matlab as cm
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp

##Constants

k = 2.
a = 9.       ##m/s
Ulow = 0.    ##m/s
Uhigh = 40. ##m/s
Ustep = 0.5 ##m/s



##Define the function
def weibullfunction(k,a):
    ##Start with an empty list each time the function is called
    Utab =  []
    F    =  []
    PDF  =  []
    ##The range for which the distribution has to be calculated
    for U in np.arange(Ulow,Uhigh+Ustep,Ustep):
        ##Add entries to the list:
        Utab.append(U)
        F.append(1-np.exp(-(U/a)**k))
        PDF.append(k/a*(U/a)**(k-1)*np.exp(-(U/a)**k))
    ##Return the lists in the given order
    return Utab,F,PDF

##Calculating for the 3 different cases
Utab,F1,PDF1 = weibullfunction(k,a)
Utab,F2,PDF2 = weibullfunction(k-0.5,a)
Utab,F3,PDF3 = weibullfunction(k,a+1)

##Plotting the functions
plt.plot(Utab,F1)
plt.plot(Utab,F2)
plt.plot(Utab,F3)
plt.title('Distrubution function')
plt.legend(('k=2, a=9','k=1.5, a=9','k=2, a=10'),1)
plt.show()
plt.plot(Utab,PDF1)
plt.plot(Utab,PDF2)
plt.plot(Utab,PDF3)
plt.title('Probability Density function')
plt.legend(('k=2, a=9','k=1.5, a=9','k=2, a=10'),1)
plt.show()
```

Figure C.3: Python Weibull distributions



```
function [F,PDF] = weibullfunction(U,k,a)
F=(1-exp(-(U./a).^k));
PDF=(k/a*(U./a).^(k-1).*exp(-(U./a).^k));



end
```

Figure C.4: The weibull distribution function as m file, weibullfunction.m



```
%Constants
k = 2;
a = 9 ;      %m/s
Ulow = 0.;    %m/s
Uhigh = 40.; %m/s
title1 = 'Weibull distribution';
title2 = 'Probability Density Function';

%Define the range for the wind speed U
U=linspace(Ulow,Uhigh);

%The lists get the value generated by the function
[F1,PDF1] = weibullfunction(U,k,a);
[F2,PDF2] = weibullfunction(U,k-0.5,a);
[F3,PDF3] = weibullfunction(U,k,a+1);

%Plotting the functions
subplot(1,2,1)
plot(U,F1,U,F2,U,F3)
title(title1)
xlabel('wind speed U [m/s]')
subplot(1,2,2)
plot(U,PDF1,U,PDF2,U,PDF3)
title(title2)
xlabel('wind speed U [m/s]')
```

Figure C.5: The MATLAB script which uses the weibullfunction.m file