NREL
NATIONAL RENEWABLE ENERGY LABORATORY

CCBlade Documentation

Release 0.1.0

S. Andrew Ning

NREL is a national laboratory of the U.S. Department of Energy
Office of Energy Efficiency & Renewable Energy
Operated by the Alliance for Sustainable Energy, LLC

This report is available at no cost from the National Renewable Energy Laboratory (NREL) at www.nrel.gov/publications.

Technical Report
NREL/TP-5000-58819
June 2013

Contract No. DE-AC36-08GO28308

NREL
NATIONAL RENEWABLE ENERGY LABORATORY

CCBlade Documentation

Release 0.1.0

S. Andrew Ning

Prepared under Task No(s). WE11.0341

National Renewable Energy Laboratory
15013 Denver West Parkway
Golden, CO 80401
303-275-3000 • www.nrel.gov

Technical Report
NREL/TP-5000-58819
June 2013

Contract No. DE-AC36-08GO28308

This report is available at no cost from the National Renewable Energy Laboratory (NREL) at www.nrel.gov/publications.

Available electronically at http://www.osti.gov/bridge

Available for a processing fee to U.S. Department of Energy and its contractors, in paper, from:

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
phone: 865.576.8401
fax: 865.576.5728
email: mailto:reports@adonis.osti.gov

Available for sale to the public, in paper, from:

U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
phone: 800.553.6847
fax: 703.605.6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/help/ordermethods.aspx

Cover Photos: (left to right) photo by Pat Corkery, NREL 16416, photo from SunEdison, NREL 17423, photo by Pat Corkery, NREL 16560, photo by Dennis Schroeder, NREL 17613, photo by Dean Armstrong, NREL 17436, photo by Pat Corkery, NREL 17721.

Printed on paper containing at least 50% wastepaper, including 10% post consumer waste.

Table of Contents

List of Figures

List of Tables

1 Introduction

CCBlade predicts aerodynamic loading of wind turbine blades using blade element momentum (BEM) theory. CC stands for continuity and convergence. CCBlade was developed primarily for use in gradient-based optimization applications where C^1 continuity and robust convergence are essential.

Typical BEM implementations use iterative solution methods to converge the induction factors (e.g., fixed-point iteration or Newton's method). Some more complex implementations use numerical optimization to minimize the error in the induction factors. These methods can be fairly robust, but all have at least some regions where the algorithm fails to converge. A new methodology was developed that is provably convergent in every instance (see *Theory*). This robustness is particularly important for gradient-based optimization. To ensure C^1 continuity, lift and drag coefficients are computed using a bivariate cubic spline across angle of attack and Reynolds number.

CCBlade is primarily written in Python, but iteration-heavy sections are written in Fortran in order to improve performance. The Fortran code is called from Python as an extension module using f2py. The module AirfoilPrep.py is also included with the source. Although not directly used by CCBlade, the airfoil preprocessing capabilities are often useful for this application. This is the stand-alone version of CCBlade, but users may also be interested in the RotorAero moduled contained in the TWISTER framework. RotorAero uses CCBlade as an option for evaluting rotor aerodynamics and is able to compute power-regulated performance (e.g., power curves, annual energy production) for any arbitrary aerodynamics code.

2 Installation

Prerequisites

C compiler, Fortran compiler, NumPy, SciPy

Download either CCBlade.py-0.1.0.tar.gz or CCBlade.py-0.1.0.zip and uncompress/unpack it.

Install CCBlade with the following command.

```
$ python setup.py install
```

To check if installation was successful run the unit tests for the NREL 5-MW model

```
$ python test/test_ccblade.py
```

An "OK" signifies that all the tests passed.

To access an HTML version of this documentation that contains further details and links to the source code, open docs/index.html.

Note: The CCBlade installation also installs the module *AirfoilPrep.py*. Although it is not necessary to use AirfoilPrep.py with CCBlade, its inclusion is convenient when working with AeroDyn input files or doing any aerodynamic preprocessing of airfoil data.

---

3 Tutorial

One example of a CCBlade application is to simulate the aerodynamic performance of the NREL 5-MW reference model. First, define the geometry and atmospheric properties.

```
import numpy as np
from math import pi
import matplotlib.pyplot as plt

from ccblade import CCAirfoil, CCBlade


# geometry
Rhub = 1.5
Rtip = 63.0

r = np.array([2.8667, 5.6000, 8.3333, 11.7500, 15.8500, 19.9500, 24.0500,
              28.1500, 32.2500, 36.3500, 40.4500, 44.5500, 48.6500, 52.7500,
              56.1667, 58.9000, 61.6333])
chord = np.array([3.542, 3.854, 4.167, 4.557, 4.652, 4.458, 4.249, 4.007, 3.748,
                  3.502, 3.256, 3.010, 2.764, 2.518, 2.313, 2.086, 1.419])
theta = np.array([13.308, 13.308, 13.308, 13.308, 11.480, 10.162, 9.011, 7.795,
                  6.544, 5.361, 4.188, 3.125, 2.319, 1.526, 0.863, 0.370, 0.106])
B = 3  # number of blades

# atmosphere
rho = 1.225
mu = 1.81206e-5
```

Airfoil aerodynamic data is specified using the `CCAirfoil` class. Rather than use the default constructor, this example uses the special constructor designed to read AeroDyn files directly `CCAirfoil.initFromAerodynFile()`.

```
afinit = CCAirfoil.initFromAerodynFile  # just for shorthand

# load all airfoils
airfoil_types = [0]*8
airfoil_types[0] = afinit('Cylinder1.dat')
airfoil_types[1] = afinit('Cylinder2.dat')
airfoil_types[2] = afinit('DU40_A17.dat')
airfoil_types[3] = afinit('DU35_A17.dat')
airfoil_types[4] = afinit('DU30_A17.dat')
airfoil_types[5] = afinit('DU25_A17.dat')
airfoil_types[6] = afinit('DU21_A17.dat')
airfoil_types[7] = afinit('NACA64_A17.dat')

# place at appropriate radial stations
af_idx = [0, 0, 1, 2, 3, 3, 4, 5, 5, 6, 6, 7, 7, 7, 7, 7, 7]

af = [0]*len(r)
for i in range(len(r)):
    af[i] = airfoil_types[af_idx[i]]
```

Next, construct the CCBlade object.

```
# create CCBlade object
rotor = CCBlade(r, chord, theta, af, Rhub, Rtip, B, rho, mu)
```

Evaluate the distributed loads at a chosen set of operating conditions.

```
# set conditions
Uinf = 10.0
tsr = 7.55
pitch = 0.0
Omega = Uinf*tsr/Rtip * 30.0/pi  # convert to RPM

# evaluate distributed loads
r, theta, Tp, Np = rotor.distributedAeroLoads(Uinf, Omega, pitch)
```

Plot the flatwise and edgewise aerodyanmic loading

```
# plot
rstar = (r - Rhub) / (Rtip - Rhub)
plt.plot(rstar, Tp/1e3, 'k', label='lead-lag')
plt.plot(rstar, Np/1e3, 'r', label='flapwise')
plt.xlabel('blade fraction')
plt.ylabel('distributed aerodynamic loads (kN)')
plt.legend(loc='upper left')
plt.show()
```

as shown in Figure 1.

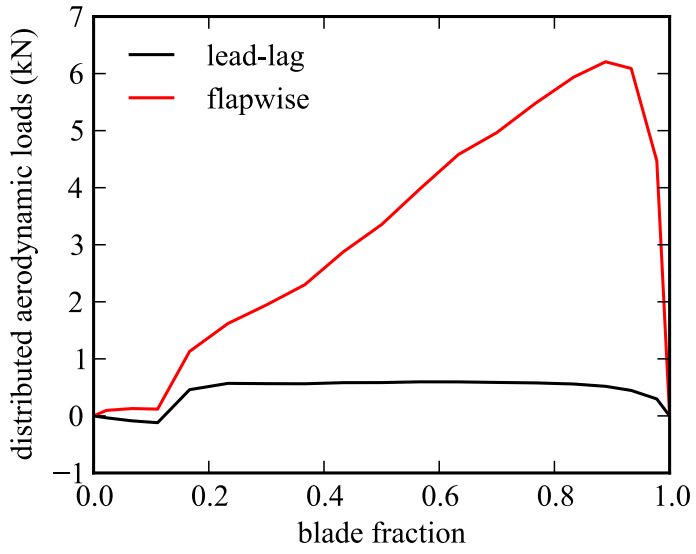

Figure 1. Flatwise and edgewise aerodynamic loads along blade.

To get the power, thrust, and torque at the same conditions (in both absolute and coefficient form), use the `evaluate` method. This is generally used for generating power curves so it expects `array_like` input. For this example a list of size one is used.

```
P, T, Q = rotor.evaluate([Uinf], [Omega], [pitch])

CP, CT, CQ = rotor.evaluate([Uinf], [Omega], [pitch], coefficient=True)

print CP, CT, CQ
```

The result is

```
>>> CP = [ 0.48329808]
>>> CT = [ 0.7772276]
>>> CQ = [ 0.06401299]
```

Note that the outputs are numpy arrays (of length 1 for this example). To generate a nondimensional power curve (λ vs c_p):

```
# velocity has a small amount of Reynolds number dependence
tsr = np.linspace(2, 14, 50)
Omega = 10.0 * np.ones_like(tsr)
Uinf = Omega*pi/30.0 * Rtip/tsr
pitch = np.zeros_like(tsr)

CP, CT, CQ = rotor.evaluate(Uinf, Omega, pitch, coefficient=True)

plt.figure()
plt.plot(tsr, CP, 'k')
plt.xlabel('$\lambda$')
plt.ylabel('$c_p$')
plt.show()
```

Figure 2 shows the resulting plot.

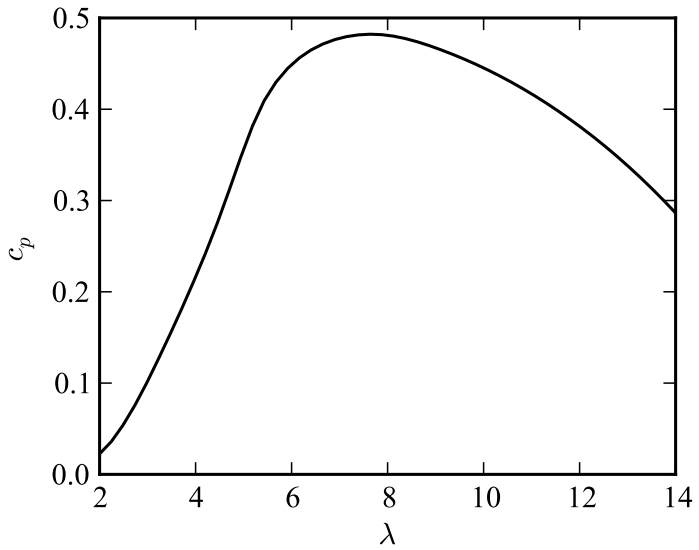

Figure 2. Power coefficient as a function of tip-speed ratio.

CCBlade provides a few additional options in its constructor. The other options are shown in the following example with their default values.

```
# create CCBlade object
rotor = CCBlade(r, chord, theta, bem_airfoil, Rhub, Rtip, B, rho, mu,
                tiploss=True, hubloss=True, wakerotation=True, usecd=True, iterRe=1)
```

The parameters `tiploss` and `hubloss` toggle Prandtl tip and hub losses repsectively. The parameter `wakerotation` toggles wake swirl (i.e., $a' = 0$). The parameter `usecd` can be used to disable the inclusion of drag in the calculation of the induction factors (it is always used in calculations of the distributed loads). However, doing so may cause potential failure in the solution methodology (see Ning (2013)). In practice, it should work fine, but special care for that particular case has not yet been examined, and the default implementation allows for the possibility of convergence failure. All four of these parameters are `True` by default. The parameter `iterRe` is for advanced usage. Referring to Ning (2013), this parameter controls the number of internal iterations on the Reynolds number. One iteration is almost always sufficient, but for high accuracy in the Reynolds number `iterRe` could be set at 2. Anything larger than that is unnecessary.

4 Module Documentation

The main methodology is contained in *CCBlade*. Airfoil data is provided by any object that implements *AirfoilInterface*. The helper class *CCAirfoil* is provided as a useful default implementation for AirfoilInterface. If CCAirfoil is not used, the user must provide an implementation that produces C^1 continuous output (or else accept non-smooth aerodynamic calculations from CCBlade). Some of the underlying implementation for CCBlade is written in Fortran for computational efficiency.

An HTML version of this documentaion is available that is better formatted for reading the code documentation and contains hyperlinks to the source code.

4.1 Airfoil Interface

The airfoil objects used in CCBlade need only implement the following evaluate() method. Although using *CCAirfoil* for the implementation is recommended, any custom class can be used.

Class Summary:

interface `ccblade.`**`AirfoilInterface`**

Interface for airfoil aerodynamic analysis.

`evaluate`(*alpha*, *Re*)

Get lift/drag coefficient at the specified angle of attack and Reynolds number

Parameters

alpha : float (rad)

angle of attack

Re : float

Reynolds number

Returns

cl : float

lift coefficient

cd : float

drag coefficient

Notes

Any implementation can be used, but to keep the smooth properties of CCBlade, the implementation should be C1 continuous.

4.2 CCAirfoil Class

CCAirfoil is a helper class used to evaluate airfoil data with a continuously differentiable bivariate spline across the angle of attack and Reynolds number. The degree of the spline polynomials across the Reynolds number is summarized in the following table (the same applies to the angle of attack although generally, the number of points for the angle of attack is much larger).

Table 1. Degree of spline across Reynolds number.

len(Re)	degree of spline
1	constant
2	linear
3	quadratic
4+	cubic

Class Summary:

class `ccblade.`**`CCAirfoil`**(*alpha*, *Re*, *cl*, *cd*)

Setup CCAirfoil from raw airfoil data on a grid.

Parameters

alpha : array_like (deg)

angles of attack where airfoil data are defined (Should be defined from -180 to +180 degrees)

Re : array_like

Reynolds numbers where airfoil data are defined (can be empty or of length one if not Reynolds number dependent)

cl : array_like

lift coefficient 2-D array with shape (alpha.size, Re.size) cl[i, j] is the lift coefficient at alpha[i] and Re[j]

cd : array_like

drag coefficient 2-D array with shape (alpha.size, Re.size) cd[i, j] is the drag coefficient at alpha[i] and Re[j]

`evaluate`(*alpha*, *Re*)

Get lift/drag coefficient at the specified angle of attack and Reynolds number.

Parameters

alpha : float (rad)

angle of attack

Re : float

Reynolds number

Returns

cl : float

lift coefficient

cd : float

drag coefficient

Notes

This method uses a spline so that the output is continuously differentiable, and also uses a small amount of smoothing to help remove spurious multiple solutions.

classmethod `initFromAerodynFile`(*aerodynFile*)
convenience method for initializing with AeroDyn formatted files

Parameters
aerodynFile : str

location of AeroDyn style airfoiil file

Returns
af : CCAirfoil

a constructed CCAirfoil object

4.3 CCBlade Class

This class provides aerodynamic analysis of wind turbine rotor blades using BEM theory. It can compute distributed aerodynamic loads and integrated quantities such as power, thrust, and torque. An emphasis is placed on convergence robustness and differentiable output so that it can be used with gradient-based optimization.

Class Summary:

class `ccblade.CCBlade`(*r*, *chord*, *theta*, *af*, *Rhub*, *Rtip*, *B=3*, *rho=1.225*, *mu=1.81206e-05*, *tiploss=True*, *hubloss=True*, *wakerotation=True*, *usecd=True*, *iterRe=1*)
Constructor for aerodynamic rotor analysis

Parameters
r : array_like (m)

radial locations where blade is defined (should be increasing)

chord : array_like (m)

corresponding chord length at each section

theta : array_like (deg)

corresponding *twist angle* at each section— positive twist decreases angle of attack.

af : list(AirfoilInterface)

list of AirfoilInterface objects at each section

Rhub : float (m)

radial location of hub

Rtip : float (m)

radial location of tip

B : int, optional

number of blades—3 assumed if not specified

rho : float, optional (kg/m^3)

freestream fluid density—standard atmosphere value at sea level used as default

mu : float, optional (kg/m/s)

dynamic viscosity of fluid—standard atmosphere value at sea level used as default

tiploss : boolean, optional

if True, include Prandtl tip loss model

hubloss : boolean, optional

if True, include Prandtl hub loss model

wakerotation : boolean, optional

if True, include effect of wake rotation (i.e., tangential induction factor is nonzero)

usecd : boolean, optional

If True, use drag coefficient in computing induction factors (always used in evaluating distributed loads from the induction factors). Note that the default implementation may fail at certain points if drag is not included (see Section 4.2 in (Ning, 2013)). This can be worked around, but has not been implemented.

iterRe : int, optional

The number of iterations to use to converge Reynolds number. Generally iterRe=1 is sufficient, but for high accuracy in Reynolds number, iterRe=2 iterations can be used. More than that should not be necessary.

`distributedAeroLoads`(*Uinf*, *Omega*, *pitch*)

Compute distributed aerodynamic loads along blade.

Parameters

Uinf : float (m/s)

freestream velocity

Omega : float (RPM)

rotor rotation speed

pitch : float (deg)

blade pitch in same direction as twist (positive decreases angle of attack)

Returns

r : ndarray (m)

radial stations where force is specified (should go all the way from hub to tip)

theta : ndarray (deg)

corresponding geometric twist angle (not including pitch)— positive twists nose into the wind

Tp : ndarray (N/m)

force per unit length tangential to the section in the direction of rotation

Np : ndarray (N/m)

force per unit length normal to the section on downwind side

evaluate(*Uinf*, *Omega*, *pitch*, *coefficient=False*)

Run the aerodynamic analysis at the specified conditions. Thrust and torque should both be magnitudes.

Parameters

Uinf : array_like (m/s)

freestream velocity

Omega : array_like (RPM)

rotor rotation speed

pitch : array_like (deg)

blade pitch setting

coefficient : bool, optional

if True, results are returned in nondimensional form

Returns

P or CP : ndarray (W)

power or power coefficient

T or CT : ndarray (N)

thrust or thrust coefficient (magnitude)

Q or CQ : ndarray (N*m)

torque or torque coefficient (magnitude)

Notes

Normalization uses Rtip and rho provided in the constructor:

CP = P / (q * Uinf * A)

CT = T / (q * A)

CQ = Q / (q * A * R)

where A = pi*Rtip**2 and q = 0.5*rho*Uinf**2

5 Theory

Note: Only an overview of the theory is included here; details can be found in Ning (2013).

The rotor aerodynamic analysis is based on blade element momentum (BEM) theory. Using BEM theory in a gradient-based rotor optimization problem can be challenging because of occasional convergence difficulties of the BEM equations. The standard approach to solving the BEM equations is to arrange the equations as functions of the axial and tangential induction factors and solve the fixed-point problem:

$$(a, a') = f_{fp}(a, a')$$

using either fixed-point iteration, Newton's method, or a related fixed-point algorithm. An alternative approach is to use nonlinear optimization to minimize the sum of the squares of the residuals of the induction factors (or normal and tangential loads). Although these approaches are generally successful, they suffer from instabilities and failure to converge in some regions of the design space. Thus, they require increased complexity and/or heuristics (but may still not converge).

The new BEM methodology transforms the two-variable, fixed-point problem into an equivalent one-dimensional root-finding problem. This is enormously beneficial as methods exist for one-dimensional root-finding problems that are guaranteed to converge as long as an appropriate bracket can be found. The key insight to this reduction is to use the local inflow angle ϕ and the magnitude of the inflow velocity W as the two unknowns in specifying the inflow conditions, rather than the traditional axial and tangential induction factors (see Figure 3).

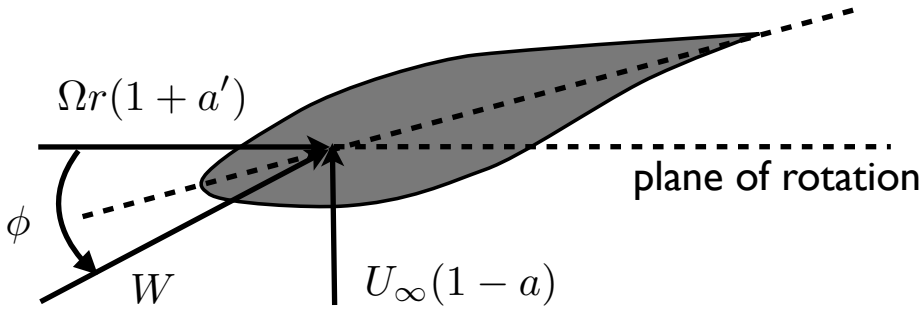

Figure 3. Parameters specifying inflow conditions of a rotating blade section.

This approach allows the BEM equations to be reduced to a one-dimensional residual function as a function of ϕ:

$$f(\phi) = \frac{\sin\phi}{1-a(\phi)} - \frac{\cos\phi}{\lambda_r(1+a'(\phi))} = 0$$

Figure 4 shows the typical behavior of $f(\phi)$ over the range $\phi \in (0, \pi/2]$. Almost all solutions for wind turbines fall within this range (for the provable convergence properties to be true, solutions outside of this range must also be considered). The referenced paper (Ning, 2013) demonstrates through mathematical proof that the methodology will always find a bracket to a zero of $f(\phi)$ without any singularities in the interior. This proof, along with existing proofs for root-finding methods like Brent's method (Brent, 1971), implies that a solution is guaranteed. Furthermore, not only is the solution guaranteed, but it can be found efficiently and in a continuous manner. This behavior allows the

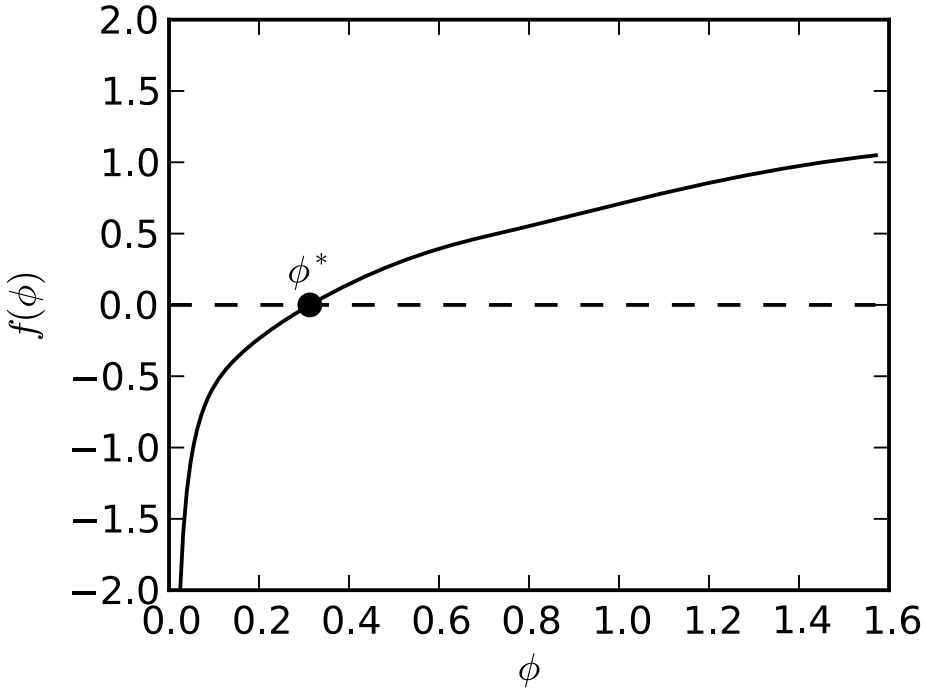

Figure 4. Residual function of BEM equations using new methodology. Solution point is where $f(\phi) = 0$.

use of gradient-based algorithms to solve rotor optimization problems much more effectively than with traditional BEM solution approaches.

Any corrections to the BEM method can be used with this methodology (e.g., finite number of blades and skewed wake) as long as the axial induction factor can be expressed as a function of ϕ (either explicitly or through a numerical solution). CCBlade chooses to include both hub and tip losses using Prandtl's method (Glauert, 1935) and a high-induction factor correction by Buhl (2005). Drag is included in the computation of the induction factors. However, all of these options can be toggled on or off. For a given wind speed, a spline is fit to the normal and tangential forces along the radial discretization of the blade before integrating for thrust and torque. This allows for smoother variation in thrust and torque for improved gradient estimation.

Bibliography

Brent, R.P. (1971). "An Algorithm with Guaranteed Convergence for Finding a Zero of a Function." *The Computer Journal* 14(4); pp. 422–425.

Buhl, M.L. (August 2005). *A New Empirical Relationship between Thrust Coefficient and Induction Factor for the Turbulent Windmill State*. NREL/TP-500-36834, National Renewable Energy Laboratory, Golden, CO.

Glauert, H. (1935). *Airplane Propellers*, Vol. 4. Springer Verlag.

Ning, S.A. (2013). "A Simple Solution Method for the Blade Element Momentum Equations with Guaranteed Convergence." *Wind Energy* (in press).