WT Perf: Programming Guide

for v 3.05.00a-adp

A. Platt

October 2012

Table of Contents

List of Figures

List of Tables

1 Overview

WT_Perf is a wind turbine performance evaluator based on blade element theory (BEM). While this theory is inherently limitted since it cannot take into account turbulent mixing of the wake from a turbine, it has an advantage over other more accurate methods because it is significantly faster. As such it is very usefull when used in conjuction with other tools, such as *FAST*, as it can provide valuable insights into what design parameters more detailed studies should be focused on.

1.1 History

The history of the code can be found in the file *ChangeLog.txt* distributed with the code. Some of the early history is not well known.

1.2 Operation

When *WT_Perf* is run, it will read in the input *.wtp* file. This file specifies the configuration of the turbine and where the airfoil data for the blades can be found. It also contains information on how to set up the algorithm and I/O as well as specifies the parameters for operation.

WT_Perf has two basic methods of operation: case analysis and parametric analysis. The specified case analysis will run a specific set of cases specified in the input file. The Parametric analysis sweeps through a range of values for the wind speed, rotor speed, and pitch angle. These operational methods are handled by the subroutines `CombCase` and `ParmAnal` which act as outer loops that call the BEM code that performs the calculations.

1.2.1 BEM

The core of *WT_Perf* consists of several solvers used together to solve for the axial and tangential induction factors that arise from the BEM equations. In its present form, three methods are used together to solve for the values for the induction factors within the subroutine `InductBEM` (section 3.1). The equations for the axial and tangential induction are dependant upon each other and several other factors including values from the tables for the airfoils used. As such, the induction factors must be solved for by using iterative numerical methods.

Within the `InductBEM` subroutine, a 2-D Newton-Raphson method is first attempted using some initial guesses for the induction values. If this fails, a marching routine is used to try to localize the solution. From there more Netwton-Raphson iterations may be used in conjuction with a bisection method (also known as a binary search method) in an attempt to locate the correct values. However, the equations for the axial and tangential induction may not be well behaved, and may possess several local minima within the solution space. It is possible for one, or all, of the root finding methods to fail to converge. In such cases, no solution can be found (whether or not one really exists). It is also possible for an incorrect solution to be found when the algorithms converge on a local minima.

In order to minimize the possibility of finding an incorrect solution or an incorrect one, several modifications have been made to the `InductBEM` subroutine so as to minimize the probability of that happening. The most recent incarnation includes some modifications to the the equations under certain conditions to minimize regions of instability due to singularities.[1]

1.3 Usage

Some information on the usage of *WT_Perf* can be found in the users guide.

[1] See the AIAA paper by Dave Maniaci, 2011.

Part I

Subroutines and functions

2 Outer routines

This chapter contains information on the outer routines of *WT_Perf*.[1] The first two routines, `CombCase` and `ParmAnal` (sections 2.1 and 2.2, respectively), are wrappers that provide cases for the main computational routine `RotAnal` (section 2.3). This routine then provides the structure and loops for calling the BEM code contained within `InductBEM` (chapter 3), and sums the forces given by the BEM theory. This routine also provides some convergence checking to decide if the results are valid.

2.1 Subroutine `CombCase`

This routine calls the subroutine `RotAnal` with each set of parameters for the combined case analysis. The cases are specified in the input file, and the results are then written out to the output file.

Figure 2.1 Callgraph for the `CombCase` subroutine.

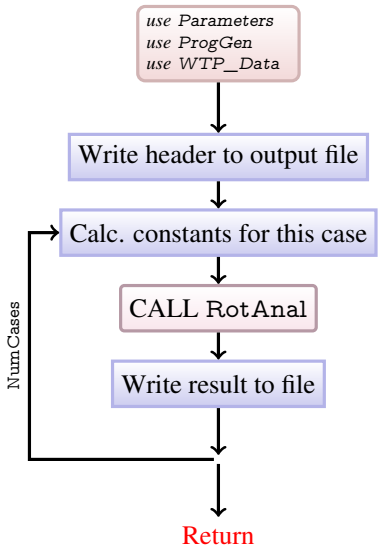

Figure 2.2 Flow diagram for the `CombCase` subroutine.

[1] The information contained here is valid for version 3.05.00a-adp.

2.2 Subroutine ParmAnal

The ParmAnal subroutine runs the parametric analysis of a turbine when it is requested in the input file.

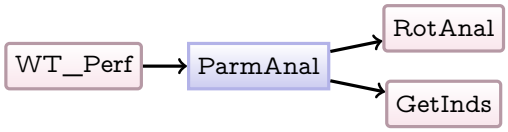

Figure 2.3 Callgraph for the ParmAnal subroutine.

2.2.1 Operation

The ParmAnal subroutine runs a simple parametric analysis of a wind turbine over the parameters of turbine, blade pitch, and wind speeds. The routine consists of three nested loops that increment each of the three parametes and call the RotAnal subroutine to calculate the performance of the turbine. This routine is only used when the number of cases (NumCases) is set to zero in the *.wtp* input file.

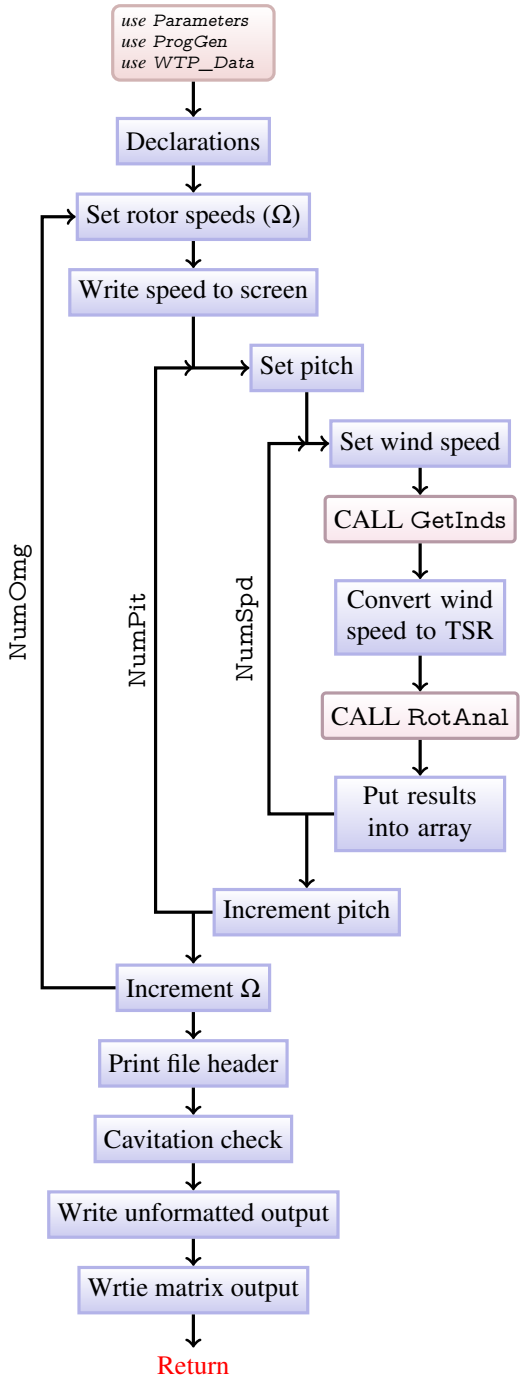

Figure 2.4 Flow diagram for the `ParmAnal` subroutine.

2.3 Subroutine `RotAnal`

This subroutine iteratively performs the analysis of the rotor.

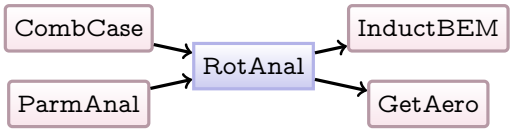

Figure 2.5 Callgraph for the `RotAnal` subroutine.

2.3.1 Operation

This routine steps through the rotor disk starting from the inner most blade section and calls the `InductBEM` routine that finds the induction factors. It then steps around the rotor segments taking into account the wind shear and any angular corrections necessary. Once all the blade sections and rotor sections have been totalled, an outer iteration loop takes the averaged values for the induction factors and adds in the skewed wake correction. When complete, this routine reports the total values for the loads, torques, and power.

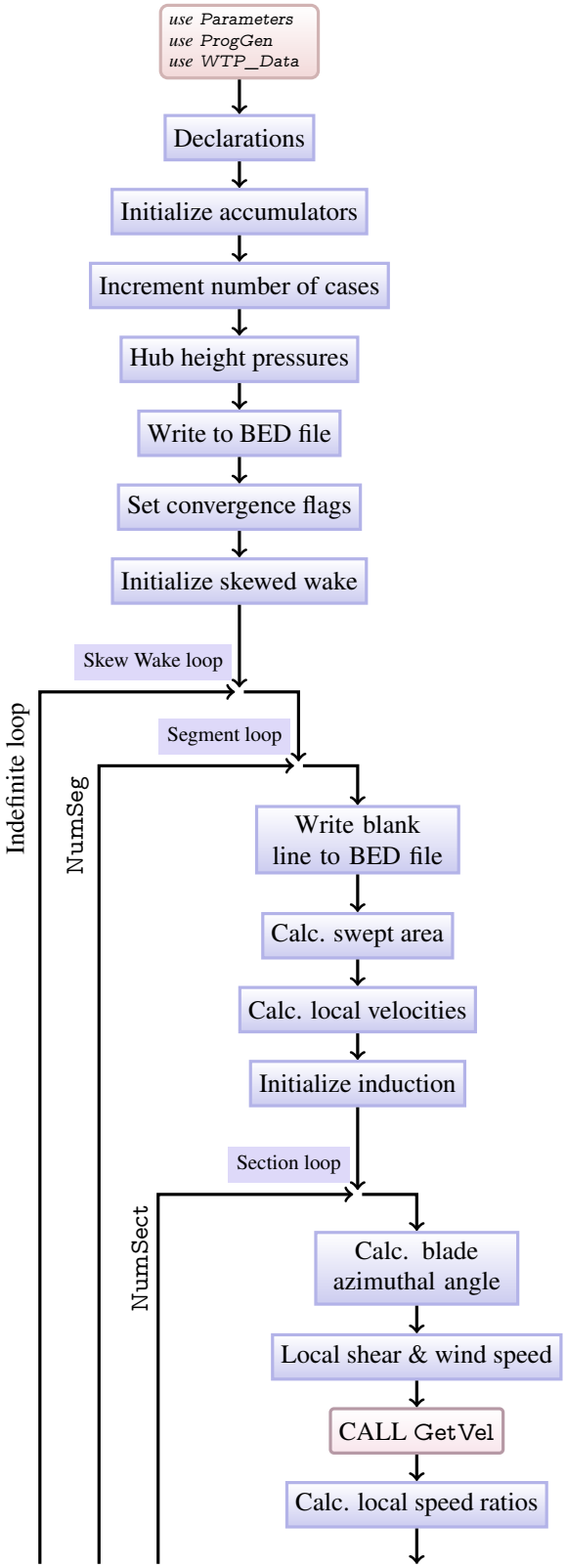

Figure 2.6 Flow diagram for the first part of the `RotAnal` subroutine (the second part is found in Figure 2.7 on page 10).

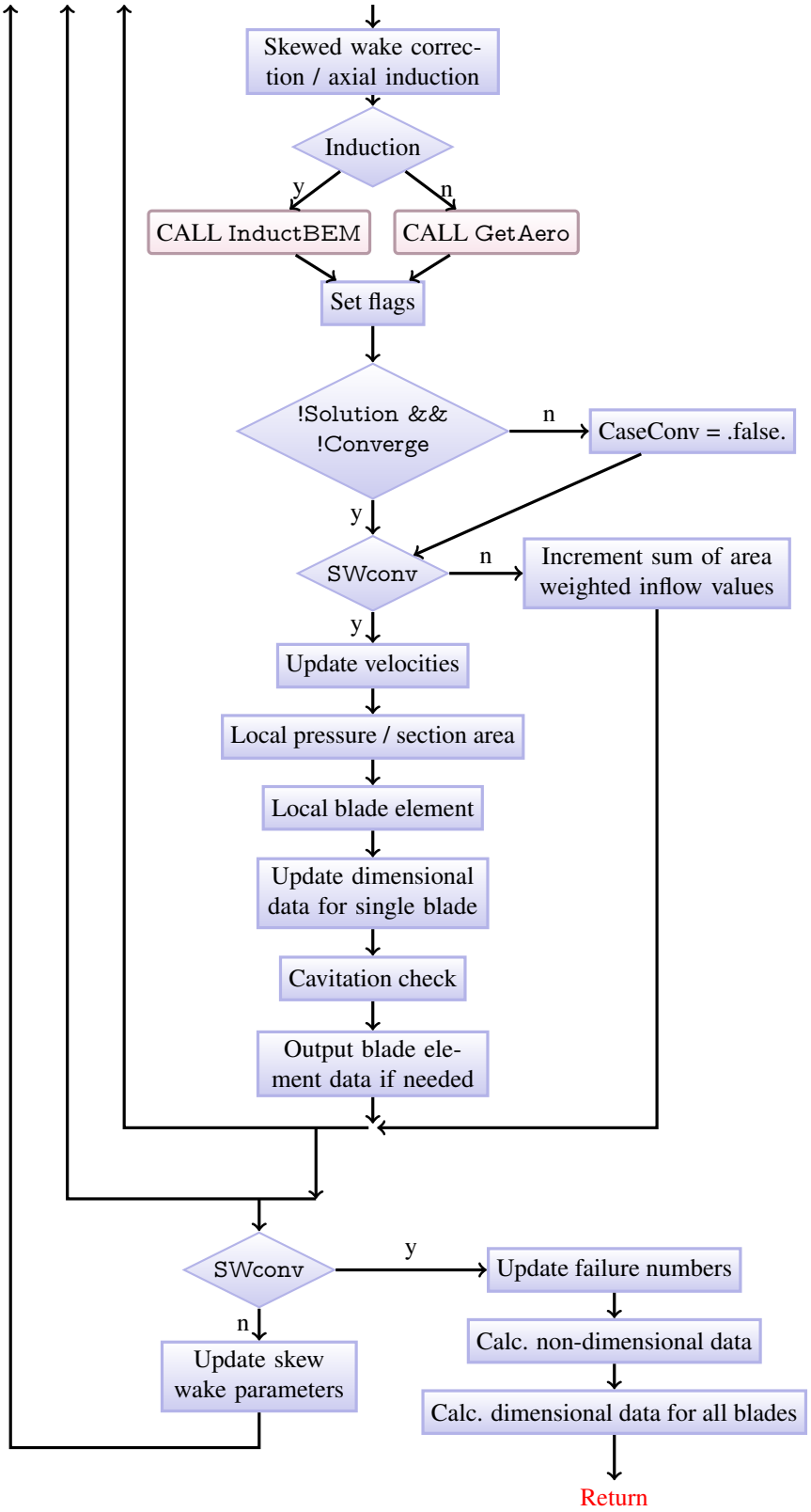

Figure 2.7 Flow diagram for the second part of the `RotAnal` subroutine (the first part is found in Figure 2.6 on page 9).

3 InductBEM

This subroutine, and the subroutines it uses, comprise the core BEM theory that *WT_Perf* uses. The routine makes use of three different solvers to find the induction values for the given case that then allow for the calculation of resulting forces and therefore the overall performance of the wind turbine.

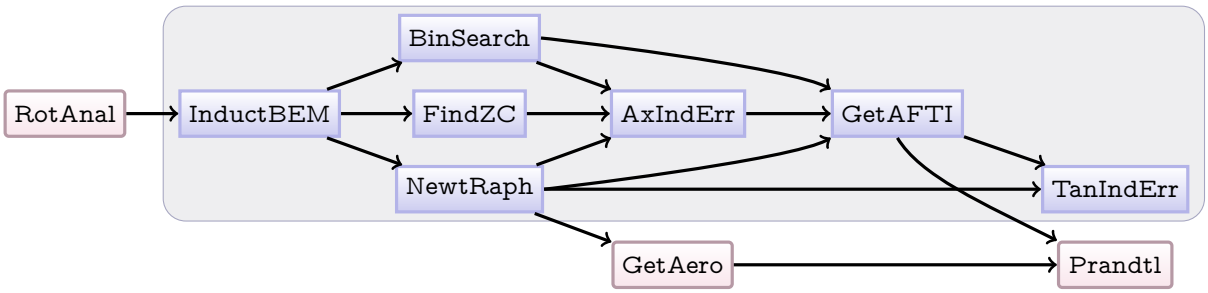

Figure 3.1 Callgraph for the `InductBEM` subroutine and its routines.

3.1 Subroutine `InductBEM`

This routine calculates the induction factors using the old *Prop* method. It will also allow the addition of the drag terms in either or both of the induction factors to (possibly) mimic the *BLADED* algorithm.

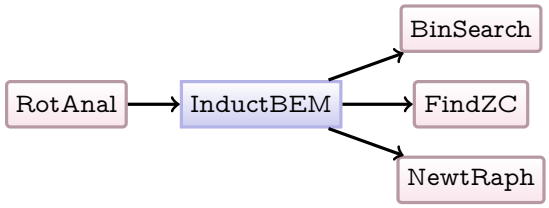

Figure 3.2 Callgraph for the `InductBEM` subroutine.

3.1.1 Operation

The `InductBEM` subroutine contains the BEM solver. In this implimentation, the solver uses several different methods to find the solutions to the induction equations. This subroutine contains several subroutines and functions that are only used here.

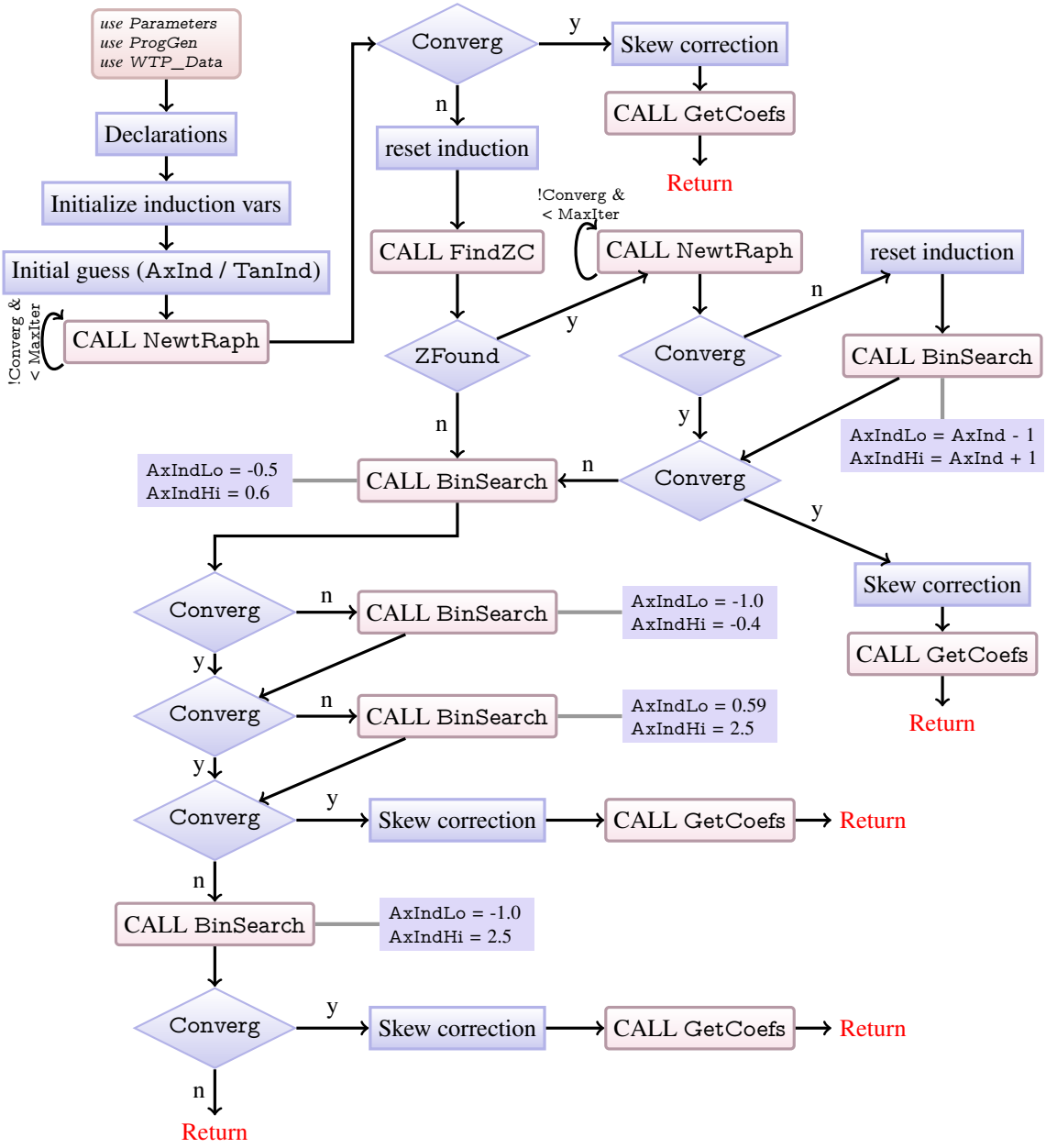

Figure 3.3 Flow diagram for the `InductBEM` subroutine.

3.2 Function BinSearch

This function performs a binary search to find where the function AxIndErr crosses $y = 0$.

Figure 3.4 Callgraph for the BinSearch subroutine.

3.2.1 Operation

This function performs a binary search for the $y = 0$ crossing of the AxIndErr function and returns the value of x for the crossing as outlined in Figure 3.5. To do this, it first looks the endpoints and checks to see if one of this is zero and returns it if so. Second, it checks that we do indeed cross through $y = 0$ within the interval specified, and returns 0.0 if it does not. After this, the algorithm zooms in to the crossing region NSplit times and returns the value for x if the crossing is found within some tolerence, ATol. If a solution has not been found, the solution is now bounded by the values of XLo and XHi. The point midway between these points is then returned.

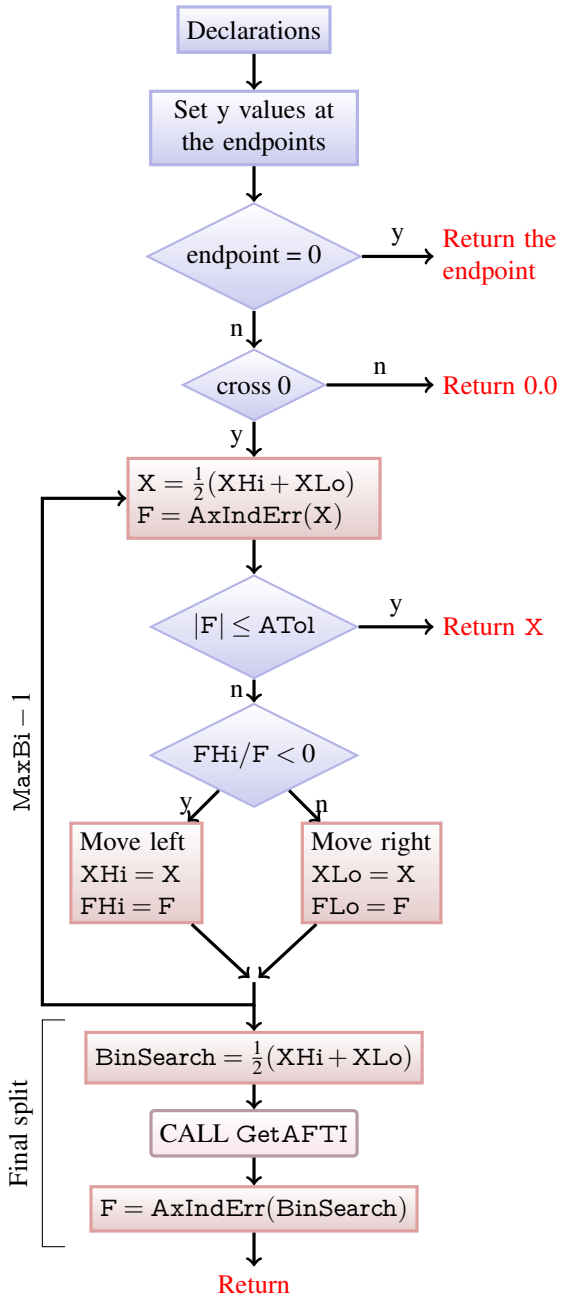

Figure 3.5 Flow diagram for the `BinSearch` function.

3.3 Subroutine FindZC

This is a recursive subroutine that finds the zero crossing of the AxIndErr function using an exhaustive search. It returns a linearly interpolated value between the bounding points of discovered crossing. This routine can call itself once in each iteration (this is controlled by a passed flag).

Figure 3.6 Callgraph for the FindZC subroutine.

3.3.1 Operation

This routine steps through the axial induction factors until it finds where the AxIndErr routine crosses through zero. In certain ambiguous situations it can call itself once to refine the region of the zero crossing (only a single level of recursion is allowed).

3.3.2 Mathematical expressions

In the final step of the algorithm, the returned x-axis value is given by the equation

$$\mathtt{FindZ\_C} = \mathtt{AxErrOld}\frac{\mathtt{AxIndHi} - \mathtt{AxIndLo}}{\mathtt{AxErrOld} - \mathtt{AxErr}} + \mathtt{AxIndLo}. \tag{3.1}$$

This equation is finding the x intercept as a linear interpolation in the bounded region given by (x_1, y_1) and (x_2, y_2) as given by (AxIndLo,AxErrOld) and (AxIndHi,AxErr). This is more obvious when rewritten as

$$x = y_1 \frac{x_2 - x_1}{y_1 - y_2} + x_1, \tag{3.2}$$

where x is our resulting intercept.

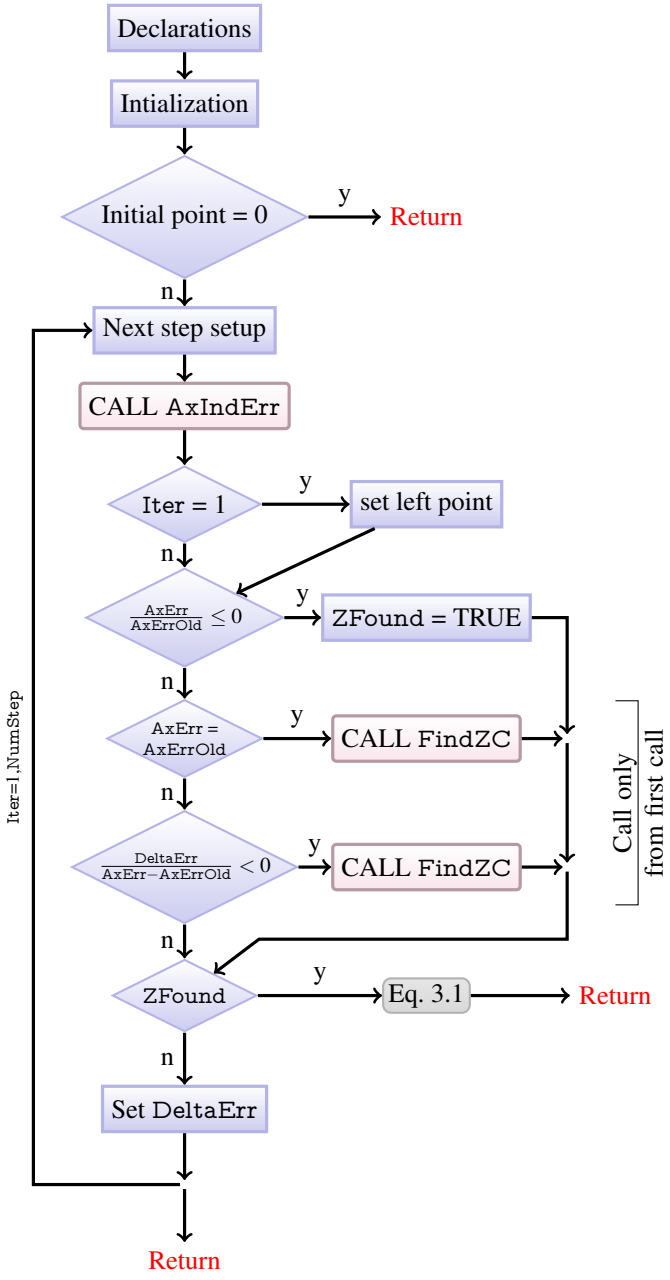

Figure 3.7 Flow diagram for the `FindZC` subroutine.

3.4 Subroutine `NewtRaph`

Figure 3.8 Callgraph for the `NewtRaph` subroutine.

This subroutine uses a Newton Raphson method to try to find the axial and tangential induction. For cases where swirl is ignored, a one-dimensional routine is used whereas a two-dimensional routine is used to try and solve for both simultaneously. While this technique is fast, the Newton Raphson method may converge on the wrong solution (when multiple solutions exist and the function is noisy). In order to attempt to safegaurd against this, several checks have been added to the routine to prevent settling on a solution in the wrong part of solution space. This is particularly problematic when dealing with the two-dimensional solver because this technique can jump over the solution and land on a different solution.

Typically, the two-dimensional solution for the axial and tangential inductions should be within the range

$$\begin{aligned} -0.5 < \quad & \texttt{AxInd} < \quad 1.5 \\ -0.5 < \quad & \texttt{TanInd} < \quad 0.5 \end{aligned} \tag{3.3}$$

for most cases. This is because BEM theory does not work well outside those ranges and because we are interested in extracting energy from wind, not expending energy (hence the offset bounds on `AxInd`). However, the Newton Raphson method as employed here does not ensure such bounds while searching for the solution.[1] Since this is not done, some simple checks are enforced after the routine completes.[2]

Another potential issue with the Newton Raphson method is that it can get stuck in a loop where one solution points towards a different solution which points to the first solution. However this routine is computationally inexpensive and unlikely to get stuck in a perfect loop. Since it is typically not iterated more than twenty times, little performance gain would likely be seen by checking for this (it might even be more expensive to run such checks given the frequency this migh occur).

3.4.1 Operation

At the beginning of this routine, a variable called `AxIndErr_Curr` is set to zero. Unless this value is small, the tangential induction is not calculated (checked within the `TanInd_Err` routine).[3] In previous versions of *WT_Perf*, this was check did not exist and would lead to absurd results for the induction factors. However, when a two-dimensional Newton Raphson method is used, we need to include the tangential induction in the calculations.

[1] This would be best done by reparameterizing the induction factors with arctan functions that would give hard bounds.

[2] Without these checks, it is possible to return values for `TanInd` that are of order 10^2 or 10^3. This causes the routine to localize on induction factors that have no physical meaning.

[3] I consider this a clumsy band-aid fix that should really be done using a reparameterization of the induction factors by bounded functions.

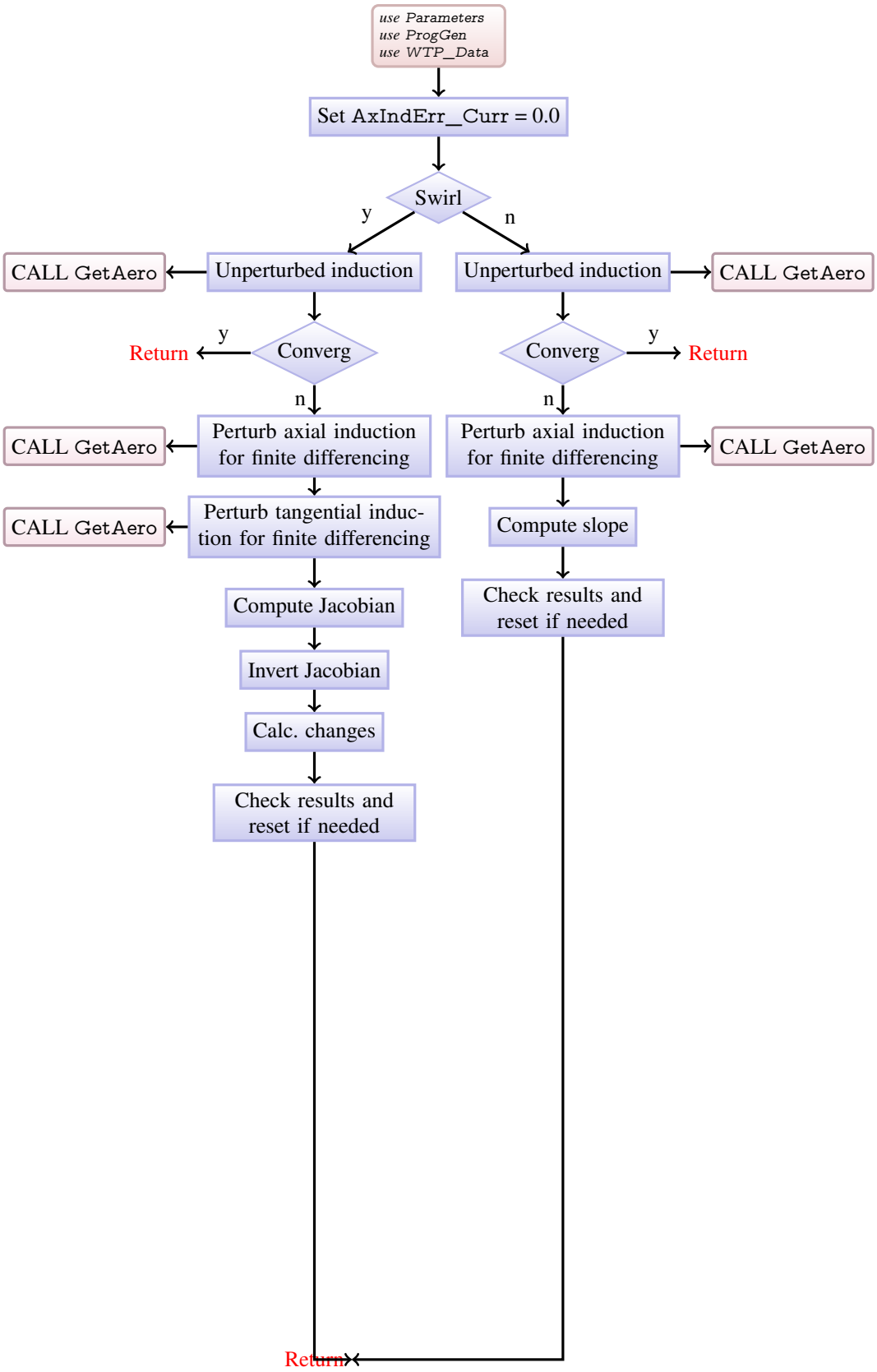

Figure 3.9 Flow diagram for the `NewtRaph` subroutine.

3.5 Function AxIndErr

This routine calculates the error in the axial induction. It is located within the InductBEM subroutine (section 3.1).

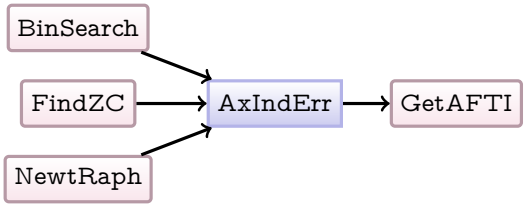

Figure 3.10 Callgraph for the AxIndErr subroutine.

3.5.1 Operation

This routine is passed in the value for the axial induction (AInd) and returns the corrected error.

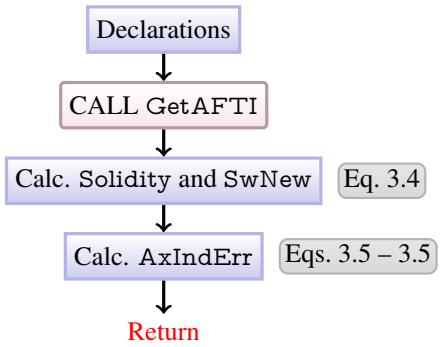

Figure 3.11 Flow diagram for the AxIndErr subroutine.

3.5.2 Mathematical expressions

To calculate the error in the axial induction, we first calculate the head loss coefficient, Dct, and the Dct from momentum theory or Glauert's imperical curve for AInd, SwNew using

$$\left.\begin{aligned} \mathrm{SwNew} &= \frac{\mathrm{VtotTang}^2 \cdot (1+\mathrm{TanInd})^2}{\mathrm{VtotNorm}^2} + (1-\mathrm{Aind})^2 \\ \mathrm{Dct} &= \frac{\mathrm{Solidity} \cdot \mathrm{CnLoc} \cdot \mathrm{SwNew}}{\mathrm{CosCone}} \end{aligned}\right\} \quad \text{for} \quad 0.97 < \mathrm{Aind} < 1.03,$$

$$\left.\begin{aligned} \mathrm{Sw} &= \frac{\mathrm{Solidity} \cdot \mathrm{CnLoc} \cdot \mathrm{CosCone}}{4 \cdot \mathrm{SinAF}^2} \\ \mathrm{Dct} &= 4 \cdot \mathrm{Sw} \cdot (1-\mathrm{AInd})^2 \end{aligned}\right\} \quad \text{otherwise.} \tag{3.4}$$

From this, we calculate the polynomial coefficients for Dct with

$$B_2 = \frac{1}{0.18} - 4 \cdot \mathrm{Loss}$$

$$B_1 = 0.8 \cdot (\mathrm{Loss} - B_2)$$

$$B_0 = 2.0 - B_1 - B_2,$$

where Loss is the total loss as calculated by the Prandtl subroutine (see section 4.1). Using these values, the axial induction error, AxIndErr, is calculated by

$$\begin{aligned} \texttt{AxIndErr} &= \frac{1-\sqrt{1-\texttt{Dct}/\texttt{Loss}}}{2} - \texttt{AInd} && \text{for } \texttt{Dct} < 0.96 \cdot \text{Loss}, \\ \texttt{AxIndErr} &= \frac{-B_1 + \sqrt{B_1^2 - 4 \cdot B_2 \left(B_0 - \texttt{Dct}\right)}}{2 \cdot B_2} - \texttt{AInd} && \text{otherwise,} \end{aligned} \tag{3.5}$$

where AInd is the axial induction factor passed to the function.

3.6 Function TanIndErr

This function calculates the error in the tangential induction.

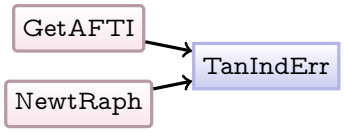

Figure 3.12 Callgraph for the TanIndErr function.

3.6.1 Operation

When the value of AxIndErr_Curr is less than 0.025, this routine will calculate the tangential induction. If AxIndErr_Curr is greater, a value of 0.0 is returned instead. The reason is that the tangential induction is currently included in axial induction calculations as a small correction term. However, when the axial induction is far from a solution, this routine would return unrealistically large values for the tangential induction (on the order of hundreds) which can cause the InductBEM routine to converge in the wrong region of parameter space. At the conclusion of this routine, the calculated error on the tangential induction is bounded to the range of ± 2 to ensure that it does not change too drastically. This and the value of AxIndErr_Curr were chosen through testing.

3.6.2 Mathematical expressions

If CtLoc is not zero, then the value of SingUpper is calculated by

$$\mathrm{SingUpper} = \left[(1.0718 \cdot |\mathrm{CtLoc} \cdot \mathrm{Solidity}| - 0.003) \cdot |\mathrm{LSR}| + 1.01\right] \cdot (\mathrm{SingOffset} + 1.0)\,. \tag{3.6}$$

3.6.2.1 SingTranstion

If the flag TISingularity is set and $2 - \mathrm{SingUpper} \leq \mathrm{AInd} \leq \mathrm{SingUpper}$, then the value of SingTransition is calculated as follows:

$$\begin{aligned} \mathrm{A0Sing} &= 1.0 \\ \mathrm{A1Sing} &= \mathrm{SingUpper} \\ \mathrm{SingTransition} &= \cos\left(\frac{\pi}{2} \cdot \frac{\mathrm{AInd} - \mathrm{A0Sing}}{\mathrm{A1Sing} - \mathrm{A0Sing}} - \frac{\pi}{2}\right) \\ \mathrm{SingTransition} &= \mathrm{SingTransition}^2, \end{aligned} \tag{3.7}$$

which can be rewriten as

$$\mathrm{SingTransition} = \cos^2\left[\frac{\pi}{2}\left(\frac{\mathrm{AInd} - 1}{\mathrm{SingUpper} - 1} - 1\right)\right]. \tag{3.8}$$

However if the flag TISingularity is not set or AInd is not between $2 - \mathrm{SingUpper}$ and SingUpper, then SingTransition is set to 1.

3.6.2.2 TanIndErr

At the conclusion of the routine, TanIndErr is calculated using the following set of equations:

$$\begin{aligned} \mathrm{TanIndCoef\_Cd} &= \frac{\mathrm{Solidity} \cdot \mathrm{CdLoc}}{4 \cdot \mathrm{Loss} \cdot \mathrm{SinAF} \cdot \mathrm{CosCone}} \\ \mathrm{TanIndCoef\_Cl} &= \frac{\mathrm{Solidity} \cdot \mathrm{ClLoc}}{4 \cdot \mathrm{Loss} \cdot \mathrm{CosAF} \cdot \mathrm{CosCone}} \\ \mathrm{TanIndCoef} &= \mathrm{TanIndCoef\_Cl} - \mathrm{SingTransition} \cdot \mathrm{TanIndCoef\_Cd} \\ \mathrm{TanIndErr} &= \frac{\mathrm{TanIndCoef}}{1 - \mathrm{TanIndCoef}} - \mathrm{TInd}. \end{aligned} \tag{3.9}$$

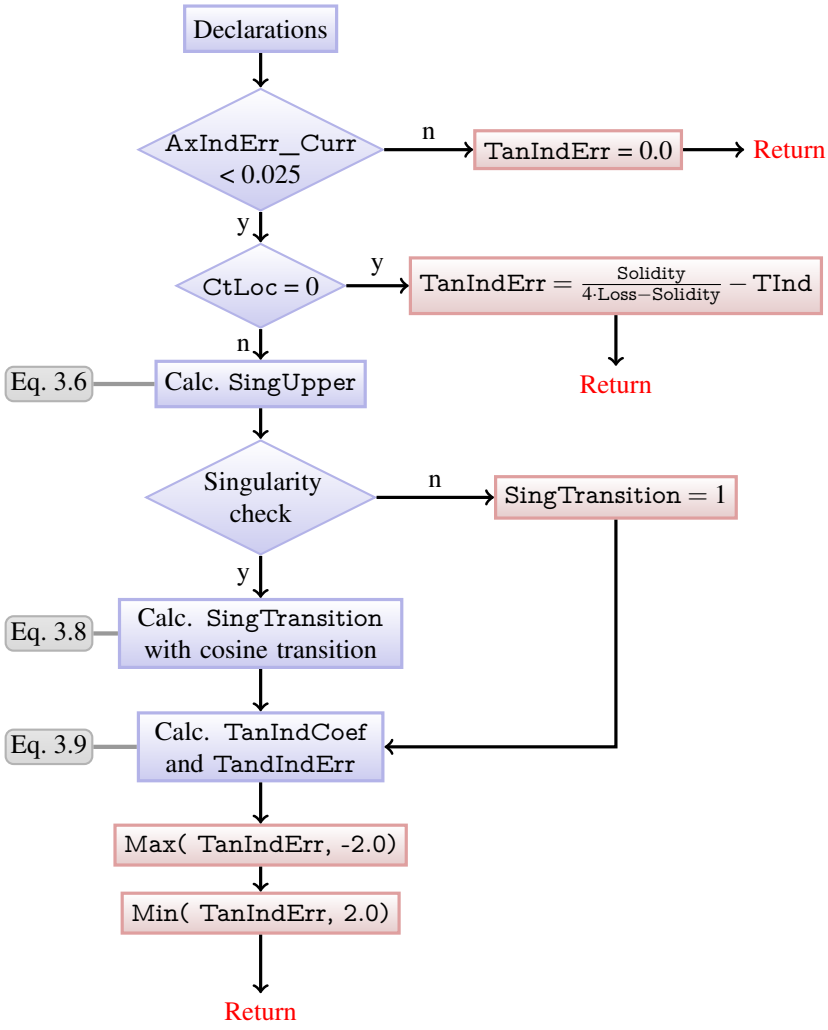

Figure 3.13 Flow diagram for the `TanIndErr` function.

3.7 Subroutine GetAFTI

This routine calculates the airflow angle and makes the call to TanIndErr to calculate the error on the tangential induction.

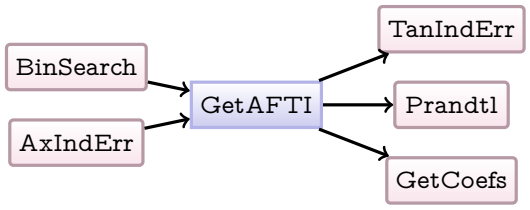

Figure 3.14 Callgraph for the GetAFTI subroutine.

3.7.1 Operation

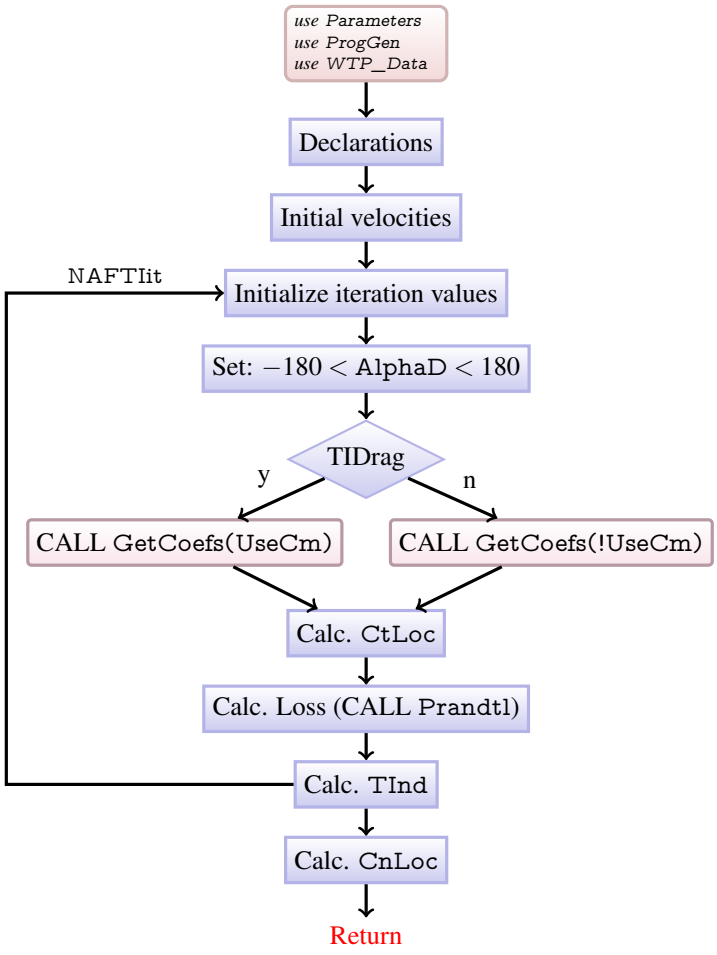

Figure 3.15 Flow diagram for the GetAFTI subroutine.

3.7.2 Mathematical expressions

The initial values of the velocity are calculated as

$$\begin{aligned} \mathtt{VIndTang} &= \mathtt{VTotTang} \\ \mathtt{VIndNrm2} &= \mathtt{VTotNorm}^2 \cdot (1 - \mathtt{AInd} \cdot \mathtt{SWcorr})^2. \end{aligned} \tag{3.10}$$

At the beginning of the loop for iteratively calculating the airflow angle and tangential induciton, the velocities and angles are calculated as

$$\begin{aligned} \mathtt{VIndTang} &= \mathtt{VTotTang} \cdot (1.0 + \mathtt{TInd}) \\ \mathtt{VInd} &= \sqrt{\mathtt{VIndTang}^2 + \mathtt{VIndNrm2}} \\ \mathtt{AF} &= \tan^{-1}(1.0 - \mathtt{AInd} + i(1.0 + \mathtt{TInd} \cdot \mathtt{LSR})) \\ \mathtt{CosAF} &= \cos(\mathtt{AF}) \\ \mathtt{SinAF} &= \sin(\mathtt{AF}) \\ \mathtt{AlfaR} &= \mathtt{AF} - \mathtt{IncidAng} \\ \mathtt{AlfaD} &= \mathtt{AlfaR} \cdot \mathtt{R2D}. \end{aligned} \tag{3.11}$$

For the tangential induction, we add the newly calculated error to our running total with

$$\mathtt{TInd} = \mathtt{TanIndErr}() + \mathtt{TInd} \tag{3.12}$$

where TanIndErr is the function given in section 3.6.

At the end we calculate the local normal coefficient, CnLoc, as

$$\begin{aligned} \mathtt{CnLoc} &= \mathtt{ClLoc} \cdot \mathtt{CosAF} + \mathtt{CdLoc} \cdot \mathtt{SinAF} && \text{with AIDrag,} \\ \mathtt{CnLoc} &= \mathtt{ClLoc} \cdot \mathtt{CosAF} && \text{otherwise.} \end{aligned} \tag{3.13}$$

4 Supporting routines

4.1 Subroutine `Prandtl`

This subroutine calculates the tip and hub losses with the Prandtl model.

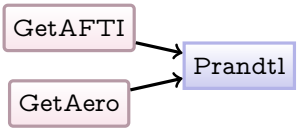

Figure 4.1 Callgraph for the `Prandtl` subroutine.

4.1.1 Operation

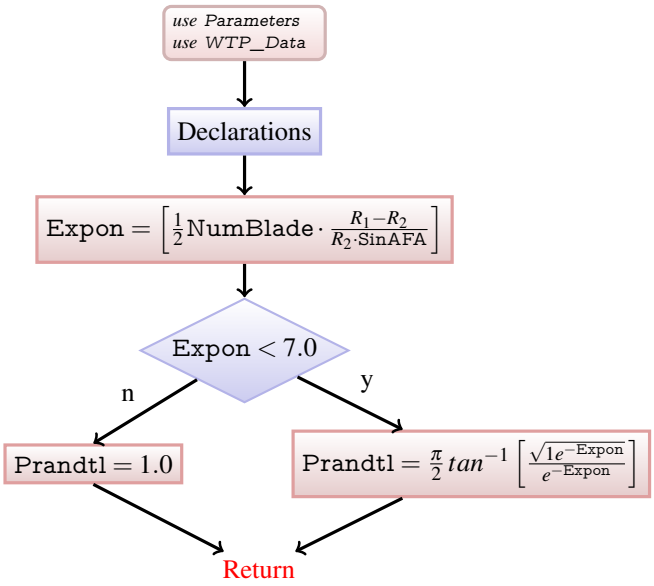

Figure 4.2 Flow diagram for the `Prandtl` subroutine.

4.2 Subroutine GetData

This subroutine reads an input file with all the information needed for a run. There is some *very* limited error checking performed in this subroutine. If the input file is ever changed to a different format[1], this routine should be rewritten to allow for free form input files.

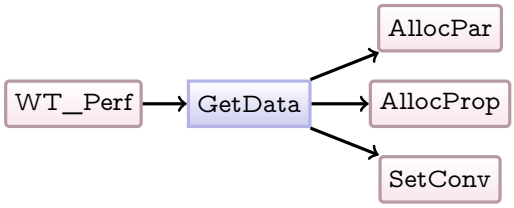

Figure 4.3 Callgraph for the GetData subroutine.

4.2.1 Operation

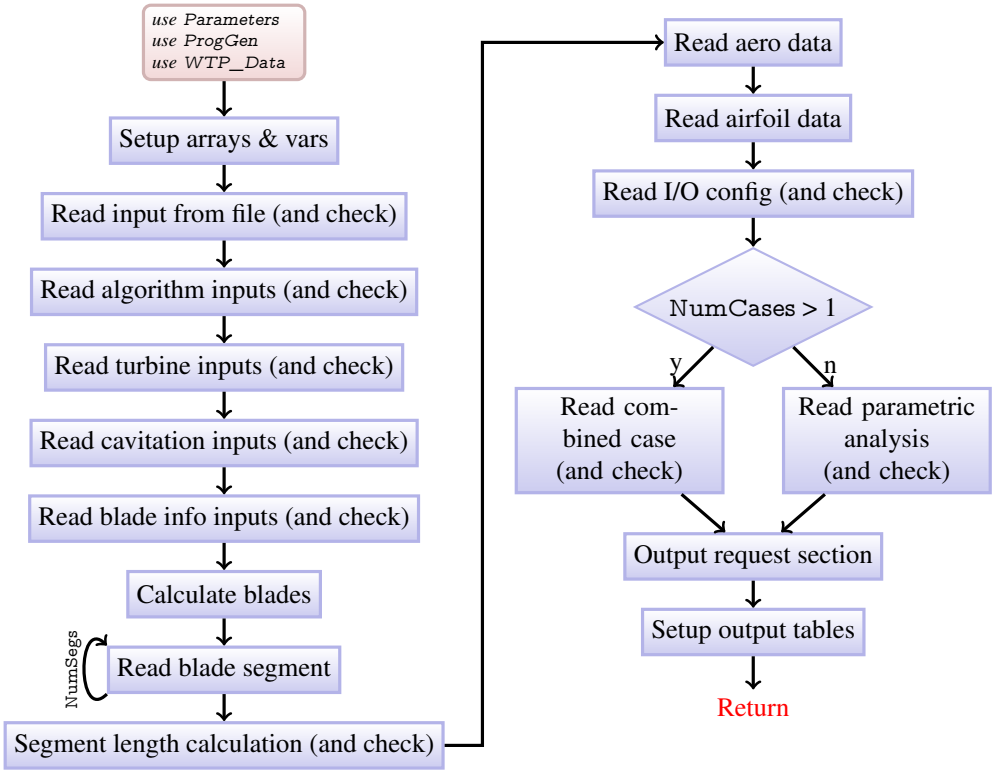

Figure 4.4 Flow diagram for the GetData subroutine.

4.2.2 Variables

There are several book-keeping related variables used locally in this routine during the readin of the file. There are a lot of other global variables that are also used (actually, all the input variables used by the program get processed here). These have not been listed in the table because there are too many of them.

In order to get rid of global variables, this subroutine would need to have structures that contain all the variables passed in and out.

[1] For example, to a *VariableName: VariableValue* style that allows for free form input file structuring.

Table 4.1 Variables used by the subroutine GetData.

Variable Name	Input	Output	Local	Global	Modified	Passed	Desc.
BldLen			x				Blade length
InpCase(3)			x				Array – combined-case input params
OmgSets(3)			x				Array – omega-setting params
PitSets(3)			x				Array – pitch-setting params
SpdSets(3)			x				Array – speed-setting params
IAF			x				Index ofAF table
ICase			x				Index of combined-anal case
IOmg			x				Index of OmgAry
IOS			x				I/O status
IPit			x				Index to PitAry
ISeg			x				Blade segment number
ISpd			x				Index of SpdArray
NumAF			x				Number of AF tables
Sttus			x				Status of allocation
AF_File			x				AF table filename
InpVersn			x				Input version string
Line			x				Input line string
SubTitle			x				Runtitle string

4.2.3 Mathematical expressions

There are several equations used during this subroutine for calculating the values of variables and for checking for input errors. Since these are fairly simple equations, refer to the source code for them and their descriptions.

4.3 Subroutine GetAero

Calculates aerodynamic information including the airflow and attack angles, the induction factors at the speed of the blade segment, the Reynolds number, and the losses.

Figure 4.5 Callgraph for the GetAero subroutine.

4.3.1 Operation

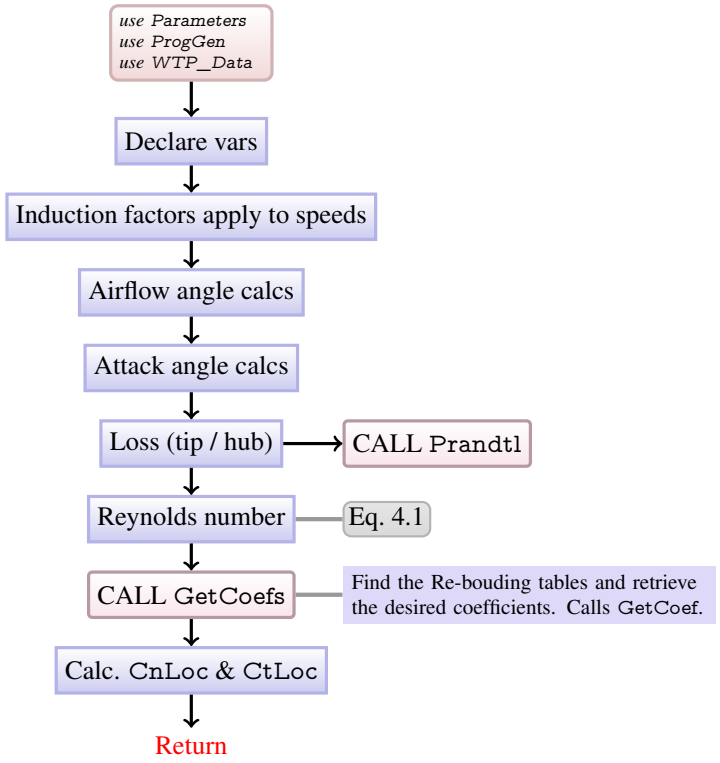

Figure 4.6 Flow diagram for the GetAero subroutine.

4.3.2 Mathematical expressions

To calculate the Reynolds number, the following equation is used:

$$\mathrm{Re} = \mathrm{VInd} \cdot \mathrm{Chord}\,(\mathrm{ISeg}) \cdot \frac{\mathrm{RotorRad}}{\mathrm{KinVisc}}. \tag{4.1}$$

4.4 Subroutine GetInds

This routine gets the indices that are used for the output of the parametric analysis. The returned values include the column, row, and table indices.

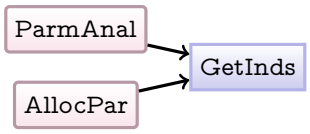

Figure 4.7 Callgraph for the GetInds subroutine.

4.4.1 Operation

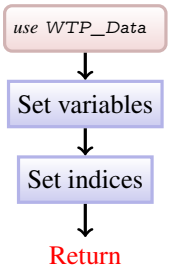

Figure 4.8 Flow diagram for the GetInds subroutine.

5 Known and potential issues

There are a few choices in how this code is implimented that could lead to some issues later on.

5.1 Numerical stability

The derivation of the equations for BEM theory yields a set of equations that cannot be exactly solved for. The equation for axial induction, a, is of the form $a = f(a)$, so this must be solved for iteratively through trial and error. In this case we start with a value for a which is used on the right hand side of the equation to calculate a new value of a on the left hand side of the equation. The difference between the values of a on the right and left hand sides of the equation is then used to select a new value of a to try. A solution is found when the values of a on both sides of the equation agree with each other. In order to decide on a new value of a to try, *WT_Perf* uses several methods including a two dimensional Newton-Raphson (`NewtRaph`, section 3.4), a binary search (`BinSearch`, section 3.2), and another more exhaustive zero crossing search method (`FindZC`, section 3.3). These are called from the `InductBEM` subroutine (section 3.1).

5.1.1 Limits on a and a'

In the derivation of the equations for BEM theory, several simplifying assumptions are made. As a result, the theory is only valid for a small range of mathematically possible solutions for the axial induction, a, and tangential induction, a'. In order to make sure the results from the `InductBEM` routine are realistic, there are places within the induction calculations where limits are enforced. This is primarily done within the `TanIndErr` function (section 3.6) and `InductBEM` subroutine (section 3.1). Since the limits are enforced in an on / off manner, there could be instances where this introduces instability in the calculations. The instability may show up as incorrect solutions for the induction calculations. It is likely, though not proven, that this will have the most effect in cases where little power is produced and high induction is present, or when there is no valid solution for the axial induction. This method of turning the limits on suddenly may however present excessive noise in axial induction calculations and potentially lead to incorrect solutions. *It would be worthwhile to modify the behaviour of the limits imposed within the TanIndErr subroutine so that they are smoothly enforced rather than abruptly.*[1]

At present, the `TanIndErr` function only evaluates when the absolute value of last calculation of the `AxIndErr` is less than 0.025. `TanIndErr` is also bounded to ± 2.0. These limits were chosen by trial and error. The use of these limits also appears to speed up the calculations since the Newt-Raphson routine is more likely to find a valid solution.

5.1.2 Convergence

Another potential issue is with how the convergence of the induction calculations is detected. To evaluate the turbine rotor, the subroutine `RotAnal` (section 2.3) calls the `InductBEM` routine to find the axial and tangential induction factors for a particular blade segment in a specific location. When the values are found, `RotAnal` steps to the next segment and calls `InductBEM` again and totals up the values for the power, thrust, and various other coefficients. After stepping through the entire rotor, an outer loop of the routine steps through the skewed wake correction using the averaged values for the induction. If at any point during the calculations for the rotor disk the `InductBEM` routine does not find a solution, a flag is set indicating that this particular configuration did not converge fully. The `RotAnal` routine continues to step through the rest of the calculations so that the blade element data can be written to the *bed* file.

5.1.2.1 Skewed wake

It is possible that cases will exist where convergence is not achieved during one of the calculations for a specific skewed wake correction, but will actually converge for the next skewed wake correction (the next iteration of the outer loop in `RotAnal`). Whether or not this is possible has not been fully explored. At present such cases will be reported as not converging even though the correct skewed wake correction factor may yield convergence.

[1] For example, an appropriately scaled *atan* function could provide a smooth transition to the limit and still provide a 1:1 correspondence within the region where a valid result would be expected. Since the tangential induction, a' (stored in the variable TanInd), is expected to be of order 0.2 or smaller, the *atan* function with a scaling of 1 would be nearly linear within this range but still provide a hard limit of ± 1.

6 Future Work

Given the current state of the InductBEM subroutine (with its associated pieces) and how many band-aid fixes have been applied, it would be worth rewriting *WT_Perf*. If this is done, the algorithm designed by Andrew Ning[1] that iterates on the inflow angle, Φ, instead of the induction factors might be worth investigating. This iteration coupled with Brent method for solving will likely yield more consistent results and run faster than the methods currently used by *WT_Perf*.

[1] There is a paper that outlines this method that is currently being written. It will likely be available at the start of 2013 on the NWTC codes website.