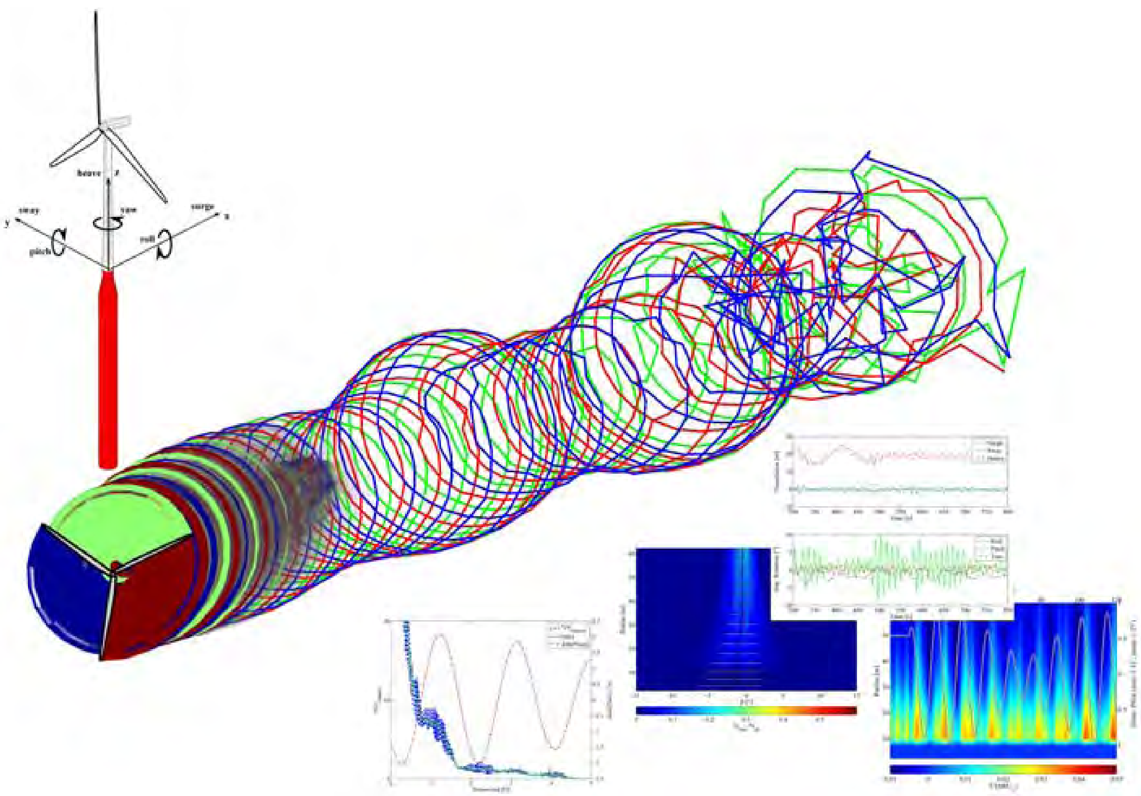
Wake Induced Dynamics Simulator
—WInDS—

VERSION 0.9
THEORY & USER MANUAL



Thomas Sebastian
tommy.sebastian@gmail.com

Wind Energy Center
Department of Mechanical & Industrial Engineering
University of Massachusetts Amherst

December 21, 2011

Contents

1 Introduction

The Wake Induced Dynamics Simulator (WInDS), written by Sebastian and Lackner [1], is a lifting–line theory (LLT) –based free vortex wake method (FVM) code developed at the University of Massachusetts Amherst Wind Energy Center with the express purpose of modeling the offshore floating wind turbine (OFWT) aerodynamics to a higher degree of accuracy than is possible via momentum balance methods. WInDS natively incorporates the multiple DOFs present in offshore floating wind turbines, resulting in a more realistic simulation of the flow field.

Sebastian and Lackner [2] demonstrated that the additional degrees–of–freedom (DOFs) associated with OFWT platform motions (Figure 1) will result in aerodynamic unsteadiness that exceeds the unsteadiness experienced by onshore and conventional offshore systems.

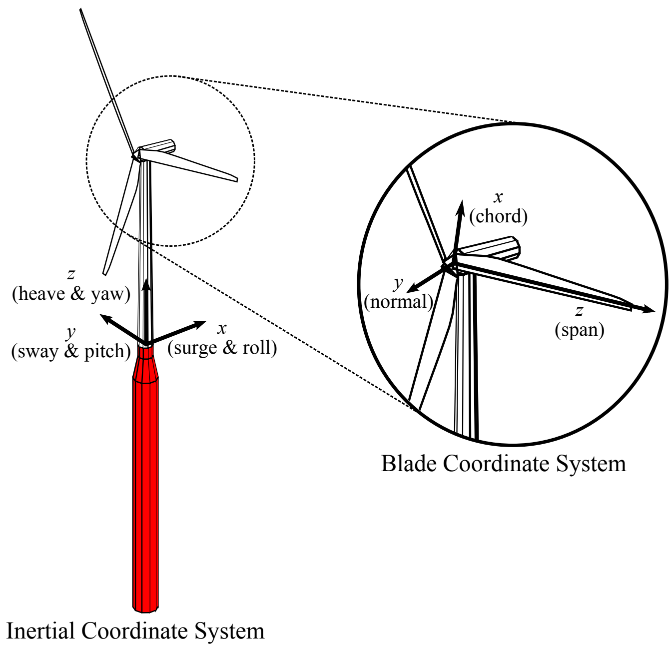


Figure 1: Offshore floating wind turbine platform DOFs and coordinate systems.

These platform DOFs generate an effective flow field velocity perturbation, $\mathbf{U_{platform}}$, given by Equation 1,

$$\begin{aligned}\mathbf{U_{platform}} = &\left(U_{surge} + \dot{\theta}_{pitch} z - \dot{\theta}_{yaw} y\right)\hat{\mathbf{i}} \\ &+ \left(U_{sway} + \dot{\theta}_{yaw} x - \dot{\theta}_{roll} z\right)\hat{\mathbf{j}} \\ &+ \left(U_{heave} + \dot{\theta}_{roll} y - \dot{\theta}_{yaw} x\right)\hat{\mathbf{k}} \qquad (1)\end{aligned}$$

where x, y, and z are the coordinates of a point in the flow field in the rotor reference frame. This additional velocity contribution is what sets the aerodynamic analysis of OFWTs apart from conventional wind turbines.

Commonly–used momentum balance approaches, like blade element momentum (BEM) theory, are conceptually simple, but rely on a number of *ad hoc*, empirically–derived corrections. The nominally inviscid, incompressible, and irrotational external flow of a wind turbine permits the use of potential flow methods. These assumptions are global, physically–consistent descriptions of the flow rather than experimentally limited extrapolations. Free vortex wake methods (FVM) are a subset of potential flow and have been in use for a number of decades.

This document will present a practical, general–purpose description of the theory and implementation of WInDS.

2 Practical Theory Behind WInDS

2.1 Biot–Savart Law

Potential flow theory permits the superposition of elementary flows to construct more complex flows. Vortex filaments, an example of a three–dimensional elementary flow, are material lines of concentrated vorticity. The Helmholtz theorems state that the circulation strength, or vorticity, is constant along the filament, which must either form a closed loop or extend to infinity. Multiple filaments may be combined to form a closed vortex lattice that grows with each time step, thereby modeling the complex and unsteady flow field associated with a wake. The velocity induced at a point of interest $\mathbf{P}$ by a filament segment of strength Γ and of length $\mathbf{L}$ between nodes $\mathbf{x}_1$ and $\mathbf{x}_2$ (Figure 2) may be computed using the Biot–Savart law.

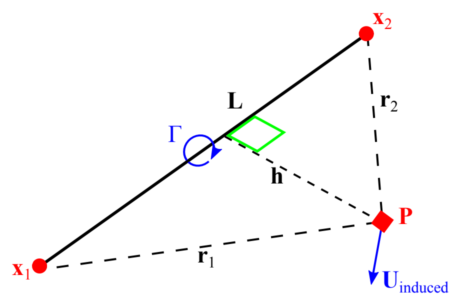


Figure 2: Diagram of relevant vectors for discretized Biot–Savart law formulation.

The induced velocity approaches infinity as the orthogonal distance between the point and filament, h, decreases. Widnall [3] showed via method of matched asymptotic expansion (MAE) that a cutoff radius, δ, may be included in the Biot–Savart equation while maintaining mathematical consistency and asymptotic validity under potential flow restrictions. Equation 2 presents the resulting modified form of the Biot–Savart equation with, referring to the geometry specified in Figure 2.

$$\mathbf{U}_{\text{induced}} = \frac{\Gamma}{4\pi} \frac{(|\mathbf{r}_1| + |\mathbf{r}_2|)\,(\mathbf{r}_1 \times \mathbf{r}_2)}{|\mathbf{r}_1|\,|\mathbf{r}_2|\,(|\mathbf{r}_1|\,|\mathbf{r}_2| + \mathbf{r}_1 \cdot \mathbf{r}_2) + (\delta\,|\mathbf{L}|)^2} \tag{2}$$

2.2 Free Vortex Wake for OFWTs

Under potential flow, vortex filament nodes move as Lagrangian markers with the local fluid flow. The advection equation that describes the motion of the nodes is given by Equation 3,

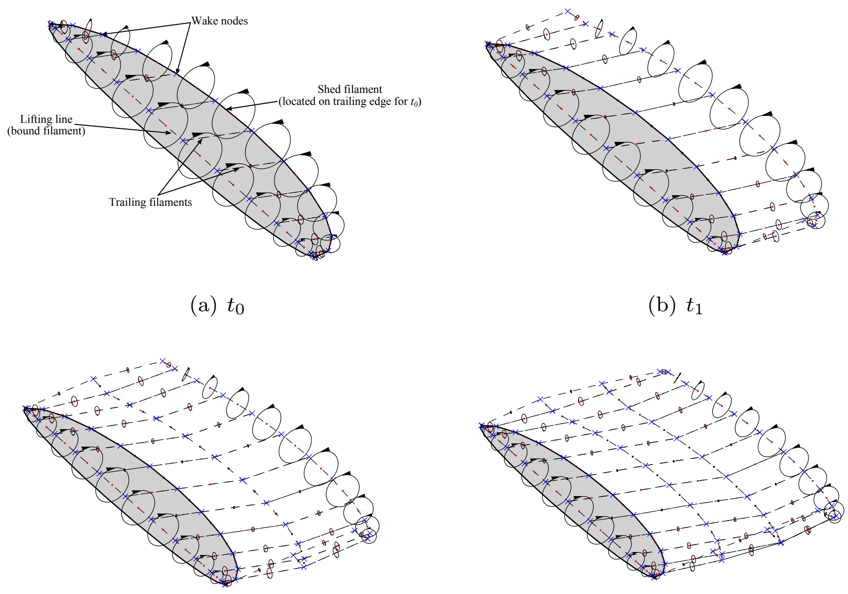
$$\frac{d\mathbf{x}}{dt} = \mathbf{U} \tag{3}$$

where $\mathbf{U}$ is the velocity of the local fluid in the rotor reference frame. In terms of a rotor, the azimuthal rotor position, ψ, and wake age, ζ, may be used to define nodal position (Equation 4),

$$\frac{d\mathbf{x}}{dt} = \frac{\partial\psi}{\partial t}\frac{\partial\mathbf{x}}{\partial\psi} + \frac{\partial\zeta}{\partial t}\frac{\partial\mathbf{x}}{\partial\zeta} = \Omega\left(\frac{\partial\mathbf{x}}{\partial\psi} + \frac{\partial\mathbf{x}}{\partial\zeta}\right) = \mathbf{U} \tag{4}$$

where Ω is the rotor speed and the contributions to $\mathbf{U}$ are given as

$$\mathbf{U} = \mathbf{U}_\infty + \mathbf{U}_{\mathbf{induced}} + \mathbf{U}_{\mathbf{platform}} \tag{5}$$



(a) t_0 (b) t_1

(c) t_2 (d) t_3

Figure 3: Vortex lattice wake structure, illustrating wake evolution between timesteps.

The wake lattice will grow with each time step (Figure 3) as the wake nodes are numerically advected. Bhagwat and Leishman [4] showed that a second-order scheme is a minimum accuracy and stability requirement for a numerical integration approach, but not sufficient. Algorithm 1 outlines how the second–order Runge–Kutta (RK2) integration scheme may be applied to the advection of wake nodes in the inertial reference frame,

Algorithm 1: Second–order Runge–Kutta (RK2)

Data: Positions and velocities at current time step, t
Result: Positions and velocities at next time step, $t + \Delta t$

1 Use forward Euler as predictor: $\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \mathbf{U}_t \Delta t$;
2 Compute velocities at newly-predicted locations via Biot–Savart law: $\mathbf{U}_{t+\Delta t} = f\left(\mathbf{x}_{t+\Delta t}\right)$;
3 Correct prediction: $\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \frac{\Delta t}{2}\left(\mathbf{U}_{t+\Delta t} + \mathbf{U}_t\right)$;

where $\mathbf{U}_t$ and $\mathbf{x}_t$ are the velocity and position vectors of a wake node, respectively, at time t. The function f represents the series of functions associated with updating the wake filament locations and strengths, computing the vortex core sizes (discussed in the following section), and calculating the induced velocity throughout the wake via the Biot–Savart law. Additionally, the overall circulation within the wake lattice domain must be constant, as stated by Kelvin's theorem (Equation 6).

$$\frac{D\Gamma}{Dt} = 0 \tag{6}$$

2.3 Vortex Core Models

More sophisticated internal vortex core models may be used as an extension of MAE and Equation 2. These models have been developed via experimental observations of real vortices, and then mathematically generalized such that they maintain physical validity when extrapolated beyond the testing range of the original experiments [5]. Vortex core models with associated core radius r_c are an engineering solution — extremely useful, despite the minimal hand–waving. The Biot–Savart law may be modified (Equation 7) to incorporate an effective viscous parameter C_ν,

$$\mathbf{U}_{\text{induced}} = \frac{C_\nu \Gamma}{4\pi} \frac{(|\mathbf{r}_1| + |\mathbf{r}_2|)(\mathbf{r}_1 \times \mathbf{r}_2)}{|\mathbf{r}_1| |\mathbf{r}_2| (|\mathbf{r}_1| |\mathbf{r}_2| + \mathbf{r}_1 \cdot \mathbf{r}_2)} \tag{7}$$

where C_ν is derived from the Vatistas core model [5] (Equation 8), reformulated for consistency with Equation 7 and Figure 2.

$$C_\nu = \left[\frac{(|\mathbf{L}| |\mathbf{r}_1|)^2 - (\mathbf{L} \cdot \mathbf{r}_1)^2}{|\mathbf{L}|^2} \right] \left[r_c^{2n} + \left(\frac{(|\mathbf{L}| |\mathbf{r}_1|)^2 - (\mathbf{L} \cdot \mathbf{r}_1)^2}{|\mathbf{L}|^2} \right)^n \right]^{-1/n} \tag{8}$$

The integer n in Equation 8 may be changed to approximate various vortex models, as illustrated in Figure 4.

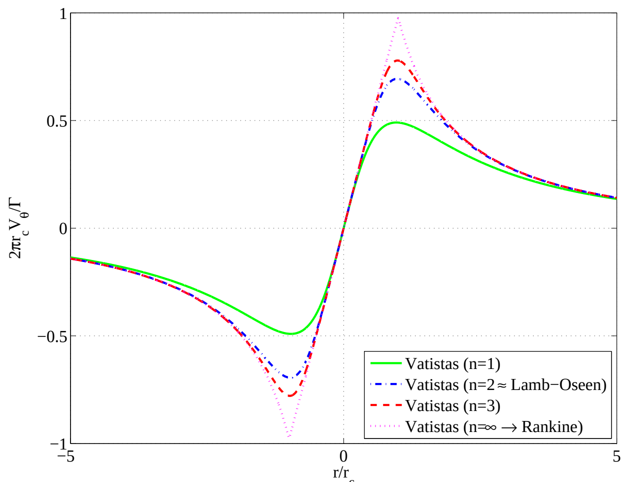


Figure 4: Normalized induced tangential velocity profiles for various Vatistas core model n values.

A freely–advecting filament may stretch. Because the net strength of the filament must remain constant, the core radius r_c must change, yielding an effective vortex core radius, r_{eff}, (Equation 9).

$$r_{\text{eff}} = r_c \left(\frac{|\mathbf{L}| + \Delta |\mathbf{L}|}{|\mathbf{L}|} \right)^{-1/2} \tag{9}$$

This value should be used in place of r_c for FVM, as in Equation 8.

2.4 Kutta–Joukowski Theorem

Lifting–line theory (LLT) concentrates the circulation related to forces generated by a lifting body onto a single bound, or lifting, filament. Thin airfoil theory dictates that this lifting–line be placed on the quarter–chord of a wing or blade (Figure 3(a)). The bound filament generates the trailing (spatial–dependence) and shed (temporal–dependence) vortex filaments that make up the vortex lattice, expressed mathematically by Equation 10.

$$\begin{aligned} \Gamma_{\text{shed}} &= \frac{\partial \Gamma_{\text{bound}}}{\partial t} \Delta t \\ \Gamma_{\text{trail}} &= \frac{\partial \Gamma_{\text{bound}}}{\partial y} \Delta y \end{aligned} \tag{10}$$

Lift and circulation strength are related by the Kutta–Joukowski theorem (Equation 11),

$$C_l = \frac{2\Gamma_{bound}}{Uc\Delta y} \tag{11}$$

where the section lift coefficient C_l is a function of Γ, flow velocity U, chord c, and section length Δy. Computing Γ_{bound} is an iterative process, described by Algorithm 2.

Algorithm 2: Fixed-Point Iteration Used by WInDS

Data: Turbine geometry and wake properties
Result: Updated bound circulation strength

```
while ΔΓ_bound ≥ tol do
    Compute vortex lattice induced velocities on the lifting–line via Equation 7;
    Compute spanwise angles of attack from induced velocities;
    Compute/table look–up C_l;
    Compute new Γ_bound via Equation 11;
    Incorporate relaxation factor in Γ_bound update to prevent overshoot;
    Update lattice to satisfy Equation 6;
```

2.5 Coordinate Systems

The wind turbine blades are defined in the blade coordinate system (BCS), with leading and trailing edges and station points defined relative to spanwise station location and chord. The velocities in this coordinate system are used to compute the angles of attack along the span, and in turn the spanwise lift coefficients via table lookup. The motions of a floating wind turbine and the free stream wind, however, are defined in the inertial coordinate system (ICS). To compute the motion–induced velocities at the spanwise station points, the position vectors of these points are transformed from the BCS to the ICS, and differentiated with respect to time to obtain the motion–induced velocities. To compute the spanwise angles of attack, the ICS-based motion–induced velocities are added to the free stream wind and the wake-induced velocities at the station points, then transformed back into the BCS.

The ICS is defined from the origin located at the water line of the floating system at its nominal position, by the x–axis downwind, the z–axis normal to the sea surface (vertical), and the y–axis as the cross product of the z and x –axes (lateral), as shown in Figure 1. The platform motions, free stream wind, and convecting wake nodes are defined in this system. The BCS is defined from the origin located at the blade or wing root on the quarter–chord line, by the chordwise x–axis (positive toward the trailing edge), the spanwise z–axis (positive toward the blade tip), and the y–axis as the cross product of the z and x –axes. Direction cosine matrices [6] (DCMs) associated with rotations because of platform yaw, pitch, and roll, nacelle yaw, shaft tilt, blade azimuthal angle, cone angle, blade pitch, and spanwise twist about each of the corresponding axes of rotation are used to transform the leading and trailing edges and station points from the BCS to the ICS. Coordinates of the blade stations on the quarter–chord line ($\mathbf{x}_{c/4}$) and trailing edge ($\mathbf{x}_{TE}$) in the ICS are used to compute a transformation matrix, $\mathbf{A}_{ICS\rightarrow BCS}$, mapping velocities computed in the ICS to the BCS for calculation of the spanwise angle of attack. The transformation matrix is given by Equation 12 in terms of $\mathbf{x}_{c/4}$, $\mathbf{x}_{TE}$, and the spanwise differences between blade stations on the quarter–chord line ($\Delta\mathbf{x}_{c/4}$).

$$\mathbf{A}_{ICS\rightarrow BCS} = \begin{bmatrix} \frac{\mathbf{x}_{TE}-\mathbf{x}_{c/4}}{\left|\mathbf{x}_{TE}-\mathbf{x}_{c/4}\right|} \\ \frac{\Delta\mathbf{x}_{c/4}}{\left|\Delta\mathbf{x}_{c/4}\right|} \times \frac{\mathbf{x}_{TE}-\mathbf{x}_{c/4}}{\left|\mathbf{x}_{TE}-\mathbf{x}_{c/4}\right|} \\ \frac{\Delta\mathbf{x}_{c/4}}{\left|\Delta\mathbf{x}_{c/4}\right|} \end{bmatrix} \tag{12}$$

2.6 WInDS Code Execution

Algorithm 3 outlines the execution of the WInDS code in terms of the aforementioned equations:

Algorithm 3: WInDS Algorithm in Terms of Theory

Data: Turbine geometry and load conditions
Result: Turbine loads and wake geometry

Import turbine geometry and load conditions;
Perform coordinate transformations (Equation 12);
Compute velocity of blade nodes;
Determine initial values for spanwise C_l and Γ_{bound} using BEM theory;
for *all time steps* **do**
    Compute the wake lattice Γ_{shed} and Γ_{trail} vorticity (Equation 10);
    Compute vortex core size (Equations 8 and 9);
    Compute induction at all wake nodes (Equation 7);
    Numerically advect wake nodes via Algorithm 1;
    Compute new Γ_{bound} via Algorithm 2;

3 Variable Descriptions

WInDS uses five variable types, corresponding to the properties of a particular variable:

- Scalar
- Span-varying vector/array
- Time series vector/array
- 4D array
- 4D cell array

Structures are used as "data containers" for fields, which represent individual data types. These structures are listed alphabetically. Note that some of the variables are stored as 4D arrays, as illustrated by equation 13.

$$\textbf{variable}\left(\underbrace{\text{ns}}_{\text{Radial index}},\ \underbrace{\text{nd}}_{\text{Dimension index}},\ \underbrace{\text{nt}}_{\text{Time index}},\ \underbrace{\text{nb}}_{\text{Blade index}}\right) \tag{13}$$

For example, the velocity in the y-direction on blade #3 in the blade coordinate system at radial station 5 and at time index 25 would be called as **vel.blade**(5,2,25,3). Note that single dimension variables, like angle of attack, have nd = 1.

Some variables, like wake node positions and vortex filament strengths, are stored as cell arrays, as shown by equation 14.

$$\textbf{variable}\,\{\text{ntau}\}\left(\underbrace{\text{ns}}_{\text{Radial index}},\ \underbrace{\text{nd}}_{\text{Dimension index}},\ \underbrace{\text{nt}}_{\text{Age}},\ \underbrace{\text{nb}}_{\text{Blade index}}\right) \tag{14}$$

In this case, nt refers to the age of the entry, and ntau refers to the time index. For example, the position at time index 6 of a wake node on the starting vortex segment that originated from the tip of blade #2 would be called as **wake.domain**{6}(end,:,end,2).

3.1 airfoils

This structure contains information on the airfoils used in the simulation:

- **airfoils.Names** contains the names of the airfoils used.
- **airfoils.profiles** contains the look–up table for each airfoil with respect to angle of attack.

3.2 blade

This structure contains geometric information on the wind turbine blade (or wing):

- **blade.TipRad** is the blade tip radius.
- **blade.HubRad** is the blade hub radius.
- **blade.RTrail** are the radial locations of the trailing filament origin points, beginning at the hub radius and ending at the tip.
- **blade.ChordTrail** are the spanwise chord lengths corresponding to blade.RTrail.
- **blade.AeroTwstTrail** are the spanwise twist angles corresponding to blade.RTrail.
- **blade.AR** is the computed aspect ratio of the blade.
- **blade.RNodes** are the radial locations of the spanwise stations, located in between blade.RTrail locations.
- **blade.Chord** are the spanwise chord lengths corresponding to blade.RNodes.
- **blade.AeroTwst** are the spanwise twist angles corresponding to blade.RNodes.
- **blade.NFoil** are the number of spanwise stations.
- **blade.DRNodes** are the radial length of each of the spanwise station sections.
- **blade.S** is the computed planform area.

3.3 const

This structure contains constant values used throughout the codes, including physical constants and unit conversion factors:

- **const.alpha** is a constant associated with the Ramasamy–Leishman vortex model.
- **const.nu** is a constant associated with the Ramasamy–Leishman vortex model.
- **const.delta** is a constant associated with the Ramasamy–Leishman vortex model.
- **const.a1** is a constant associated with the Ramasamy–Leishman vortex model.
- **const.rho** is the free stream atmospheric density.
- **const.rpm2rds** is the conversion factor from [rpm] to [radians/second].
- **const.drr** is the conversion factor from degrees to radians.

3.4 fastout

This structure contains all FAST–generated output time series. The number of fields included is dependent on the user. The following fields are explicitly used by WInDS:

- **fastout.Time** are the timestamps.
- **fastout.WindVxi** are the wind time series in the x–direction.
- **fastout.WindVyi** are the wind time series in the y–direction.
- **fastout.WindVzi** are the wind time series in the z–direction.
- **fastout.Azimuth** are the azimuth angle time series of the rotor.
- **fastout.BldPitch1** are the blade pitch time series for blade #1.
- **fastout.BldPitch2** are the blade pitch time series for blade #2.
- **fastout.BldPitch3** are the blade pitch time series for blade #3.
- **fastout.NawYaw** are the nacelle yaw time series.
- **fastout.PtfmSurge** are the platform surge time series.
- **fastout.PtfmSway** are the platform sway time series.
- **fastout.PtfmHeave** are the platform heave time series.
- **fastout.PtfmRoll** are the platform roll time series.
- **fastout.PtfmPitch** are the platform pitch time series.
- **fastout.PtfmYaw** are the platform yaw time series.
- **fastout.TipSpdRat** are the tip speed ratio time series.
- **fastout.RotSpeed** are the rotor speed time series.

3.5 perf

This structure contains the WInDS-generated performance metrics:

- **perf.cl** are the spanwise lift coefficient time series.
- **perf.cd** are the spanwise drag coefficient time series.
- **perf.aoa** are the spanwise angle of attack time series.
- **perf.bem** is a structure containing performance values computed using BEM:
  - **perf.bem.cl** are the spanwise BEM–computed lift coefficient time series.
  - **perf.bem.cd** are the spanwise BEM–computed drag coefficient time series.
  - **perf.bem.phi** are the spanwise BEM–computed inflow angle time series.
  - **perf.bem.aoa** are the spanwise BEM–computed angle of attack time series.
  - **perf.bem.a** are the spanwise BEM–computed axial induction time series.
- **perf.CL** are the total lift coefficient time series for each blade.

3.6 platform

This structure contains the floating platform properties. These are generally not used by WInDS, but included for reference and for file naming purposes:

- **platform.Type** is the type of platform used in the simulation. This variable may be used to define output filenames.
- **platform.TwrDraft** is the downward distance from mean sea level to the tower base platform connection.
- **platform.PtfmCm** is the downward distance from mean sea level to the platform CM.
- **platform.PtfmRef** is the downward distance from mean sea level to the platform reference point.
- **platform.PtfmDraft** is the effective platform draft.
- **platform.PtfmDiam** is the effective platform diameter.

3.7 pos

This structure contains the computed geometric values and station positions:

- **pos.platform** are the locations of the platform reference point in the inertial coordinate system.
- **pos.hub** are the locations of the rotor cone apex in the inertial coordinate system.
- **pos.lead** are the locations of the blade leading edge corresponding to blade.RTrail.
- **pos.bound** are the locations of the blade quarter–chord (lifting–line) corresponding to blade.RNodes.
- **pos.colloc** are the locations of the blade 3/4–chord (collocation points) corresponding to blade.RNodes.
- **pos.quarter** are the locations of the blade quarter–chord (lifting–line) corresponding to blade.RTrail.
- **pos.trail** are the locations of the blade trailing edge corresponding to blade.RTrail.
- **pos.end** are the locations of the blade trailing edge corresponding to blade.RNodes.
- **pos.blade_rotseq** defines the blade-specific rotation sequences.
- **pos.nodes** define the transformation matrix between the inertial and blade coordinate systems.
- **pos.aoag** are the geometric angles of attack.

3.8 turbine

This structure contains basic information on the turbine geometry:

- **turbine.NumBl** is the number of blades.
- **turbine.OverHang** is the distance from the yaw axis to the rotor apex.
- **turbine.TowerHt** is the height of the tower.
- **turbine.Twr2Shft** is the vertical distance from the tower top to the rotor shaft.
- **turbine.ShftTilt** is the rotor shaft tilt angle.
- **turbine.Precone** are the blade cone angles.

3.9 user

This structure contains user–defined conditions WInDS operating conditions:

- **user.t** contains the initial and final times and the frequency of the interpolated time series.
- **user.filename** is the user–defined output filename.
- **user.tol** is the convergence tolerance for the Kutta–Joukowski iteration.
- **user.d** is the cut–off distance for vortex core models.
- **user.co** is the distance at which vortex contributions are assumed to be zero.
- **user.integ** selects the integration method used by WInDS.
- **user.ns** is the number of radial stations along each blade.
- **user.maxiter** is the maximum allowed number Kutta–Joukowski iterations.
- **user.roll** selects whether or not wake self–induction is included.
- **user.anim** selects whether or not a wake evolution animation is generated.
- **user.time** is the initialization time of the simulation, used for file naming purposes.
- **user.kjtype** selects the root-finding method used by Kutta–Joukowski.
- **user.relax** is the relaxation factor used by Kutta–Joukowski.
- **user.ellip** is a structure containing user–defined variables for simulating an elliptical wing:
  - **user.ellip.b** is the wingspan.
  - **user.ellip.AR** is the wing aspect ratio.
  - **user.ellip.wind** is the wind velocity vector.
  - **user.ellip.pitch** is the initial and final pitch angle and the trigger time for pitch change.
  - **user.ellip.pitchrate** is the pitch rate.
  - **user.ellip.yaw** is the yaw angle.
- **user.rotor** is a structure containing user–defined variable for simulating a rotor:
  - **user.rotor.wind** is the wind velocity vector.
  - **user.rotor.tsr** is the tip speed ratio.
  - **user.rotor.casetype** is used for file naming purposes.
  - **user.rotor.pitch** is the blade pitch angle.
  - **user.rotor.yaw** is the yaw angle.
  - **user.rotor.modes** is a cell array used to characterize the platform motions as a bimodal sinusoid.

3.10 vel

This structure contains computed velocities:

- **vel.bound** are the velocites along the blade quarter–chord (lifting–line) in the inertial coordinate system.
- **vel.blade** are the velocites along the blade quarter–chord (lifting–line) in the blade coordinate system.
- **vel.platform** are the velocities of the platform reference point in the inertial coordinate system.
- **vel.hub** are the motion–induced velocities of the rotor cone apex in the inertial coordinate system.
- **vel.relhub** are the total velocities of the rotor cone apex in the inertial coordinate system.
- **vel.domain** are the wake node velocities in the inertial coordinate system.
- **vel.uind** are the wake self–induced velocities in the inertial coordinate system.
- **vel.uindb** are the lifting–line induced velocities in the inertial coordinate system.
- **vel.unid_shed** are the wake self–induced velocities because of shed vorticity in the inertial coordinate system.
- **vel.uind_trail** are the wake self–induced velocities because of trailing vorticity in the inertial coordinate system.
- **vel.rot** are the lifting–line induced velocities in the blade coordinate system.
- **vel.uindb_shed** are the lifting–line induced velocities because of shed vorticity in the inertial coordinate system.

- **vel.uindb_trail** are the lifting–line induced velocities because of trailing vorticity in the inertial coordinate system.
- **vel.tot** are the total lifting–line velocities in the blade coordinate system.

3.11 wake

This structure contains computed wake properties:

- **wake.domain** are the wake node locations.
- **wake.Re** are the vortex Reynolds numbers.
- **wake.rc** are the vortex core radii.
- **wake.length** are the lengths of the vortex filaments.
- **wake.rc_eff** are the effective vortex core radii.
- **wake.gamma** are the filament circulation strengths.
- **wake.r0** are the initial vortex core radii.
- **wake.strain** are the computed filament strain.

3.12 wind

This structure contains free stream wind values:

- **wind.infty** are the free stream wind velocities.
- **wind.time** are the timestamps.
- **wind.inftyM** are the magnitudes of the free stream wind velocities.

4 Core Functions

A generalized, modular approach, illustrated by Algorithm 4, was taken when writing the core functions that make up WInDS. These functions are described in the following sections.

Algorithm 4: WInDS Algorithm in Terms of Functions

Data: Turbine geometry and load conditions
Result: Turbine loads and wake geometry

```
Import turbine geometry and load conditions (Sections 4.7 & 4.10);
Determine position of blade nodes (Sections 4.8 & 4.4);
Compute velocity of blade nodes because of platform, turbine and rotor motions (Section 4.12);
Determine initial values for spanwise lift distribution and bound circulation strength using BEM
theory (Section 4.2);
for all time steps (Section 4.1) do
    Compute circulation strength of trailing and shed filaments;
    Compute vortex core size, including filament strain effects (Section 4.5);
    Compute induction at all wake nodes via Biot–Savart law (Section 4.3);
    Convect wake nodes via user-selected numerical integration scheme;
    Compute new bound circulation strength via iteration on Kutta–Joukowski theorem (Section 4.9);
```

4.1 Main WInDS Driver

This driver code is a script that calls all of the functions in the correct order, allows for user–specified variables to be defined, and saves the completed simulation results.

```
%% WInDS Driver -> Wake Induced Dynamics Simulator
%
% Driver script to compute wind turbine performance via unsteady lifting
% line method.
%
% Uses FAST input and output files to define wind turbine geometry and
% operating conditions. WInDS then predicts wind turbine performance due
% to wake evolution via free vortex wake method and lifting-line theory.
%
%
% ****Function(s)****
% constants          Load constants used by other functions
% elliptical         Generate geometry and variables for elliptical wing
% rotor              Generate geometry and variables for rotor
% input_import       Import FAST-formatted input files
% output_import      Import FAST-formatted output files
% input_mod          Modify inputs, remove discontinuities
% kinematics         Compute positions of blade stations
% velocity           Compute velocity contributions due to kinematics
% initials           Set initial conditions and preallocate memory
% performance        Compute performance and load values
%
%
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited December 16, 2011
%

%% Clear command window and workspace
clear all
close all
clc

%% !!!User-defined variables!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
user.t=[0 5 5]; %Initial t, final t, and frequency in Hz
user.filename='NRELrotor'; %Test case (elliptical, rotor type, or .fst file)
user.tol=1e-8; %Tolerance value for convergence of numerical methods
user.d='visc1'; %Core model for filaments (numerical values are the squared cutoff radius,
                %'viscX' applied viscous model of index X)
user.co=1000; %Distance from wake nodes beyond which influence is negligible
user.integ='pcc'; %Numerical integration scheme
user.ns=20; %Number of spanwise stations
user.maxiter=30; %Maximum number of iterations for Kutta-Joukowski theorem
user.roll='true'; %If 'true', will apply induction to all wake nodes
user.anim='true'; %If 'true', will generate animation of wake evolution
user.time=datestr(now ,'mm-dd-yyyy_HHMM'); %Date and time of code execution
user.kjtype='fixed'; %Use either fixed point or Brent's method for convergence (Brent is
                     %still a bit coarse)
user.relax=0.25; %Relaxation value for fixed-point iteration

%%Variables for user.ellip.* used only if user.filename='elliptical'
user.ellip.b=10; %Elliptical wingspan
user.ellip.AR=6; %Elliptical wing aspect ratio (AR=b^2/S)
user.ellip.wind=[1 0 0]; %Wind velocity vector
user.ellip.pitch=[5 5 0]; %Pitch angle of elliptical wing (in degrees)
user.ellip.pitchrate=0; %Pitch rate of elliptical wing (in degrees)
user.ellip.yaw=0; %Yaw angle of elliptical wing (in degrees)

%%Variables for user.rotor.* used only if user.filename='rotor'
user.rotor.wind=[11.4 0 0]; %Wind velocity vector
user.rotor.tsr=7; %Tip speed ratio
user.rotor.casetype='static_rated';
```

```
user.rotor.pitch=0; %Pitch angle of rotor blade (in degrees)
user.rotor.yaw=0;
user.rotor.modes=[];%{'Surge' 0.72520 0.00740 -1.16256 -0.44205 0.07750 2.60940 13.60156 10};

addpath(genpath(fullfile(cd))); %Add directories to search path

%% Load constants (physical and derived)
[const]=constants;


%% Load test case (elliptical wing, rotor, or FAST-generated)
if strcmp(user.filename,'elliptical')
    [blade,turbine,platform,fastout,airfoils,wind]=elliptical(user);
elseif strcmp(user.filename,'NRELflat')
    [blade,turbine,platform,fastout,airfoils,wind]=NRELflat(user);
elseif strcmp(user.filename,'NRELrotor')
    [blade,turbine,platform,fastout,airfoils,wind]=NRELrotor(user);
elseif strcmp(user.filename,'FAST')
    [airfoils,blade,turbine,platform,wind]=input_import(user.filename);
    [fastout]=output_import(user.filename,user.t);
end

%% Compute positions of blade stations in inertial reference frame
[pos]=kinematics(blade,turbine,platform,fastout);

%% Compute velocities of blade stations due to external motions
[vel,pos]=velocity(pos,blade,turbine,wind,fastout);

%% Define initial values (wake strength, geometry, etc)
[wake,vel,perf]=initials(pos,vel,blade,turbine,wind,airfoils,fastout,const,user);

%% !!!PRIMARY LOOP OVER TIMESERIES!!!
%Determine size of test vectors/arrays
nt=length(fastout.Time); %Number of timesteps
nb=turbine.NumBl; %Number of blades
ns=length(blade.RNodes); %Number of shed nodes (stations)
tm=zeros(nt,1); %Preallocate memory for timer (time for each timestep)

for p=2:nt
    tic; %Begin timing this timestep
%Update shed and trailing filament strength
    %Bound filament for previous timestep becomes new bound filament
    wake.gamma.shed{p}(:,:,1,:)=wake.gamma.shed{p-1}(:,:,1,:);
    %Compute spanwise change in bound filament to compute first set of trailing filaments
    wake.gamma.trail{p}(:,:,1,:)=diff([zeros(1,1,1,nb) ; wake.gamma.shed{p}(:,:,1,:) ; ...
        zeros(1,1,1,nb)],1);
    %Previous set of trailing filaments becomes new set of trailing filaments
    wake.gamma.trail{p}(:,:,2:end,:)=wake.gamma.trail{p-1};
    %Shed filaments computed via spanwise summation of trailing filaments (ensure Kelvin's
    %theorem is satisfied)
    wake.gamma.shed{p}(:,:,2:end,:)=diff(cat(3,cumsum(wake.gamma.trail{p}(1:end-1,:,:,:),1), ...
        zeros(ns,1,1,nb)),1,3);

%Modify vortex core size via Ramasamy-Leishman model and include effect of filament stretching
%from previous timestep
    wake=vcore(wake,const,fastout,user,p);

%Compute induced velocity at all points
    %Velocity induced by shed filaments on all nodes in wake
    if strcmp(user.roll,'true')
        vel.uind_shed=BiotSavart(wake.domain{p}(1:end-1,:,:,:),wake.domain{p}(2:end,:,:,:), ...
            wake.domain{p},wake.gamma.shed{p},wake.rc_eff.shed{p},user.d,user.co,'full');
        %Velocity induced by trailing filaments on all nodes in wake
        vel.uind_trail=BiotSavart(wake.domain{p}(:,:,2:end,:),wake.domain{p}(:,:,1:end-1,:), ...
            wake.domain{p},wake.gamma.trail{p},wake.rc_eff.trail{p},user.d,user.co,'full');
        %Sum the induced velocity contributions due to shed and trailing filaments
        vel.uind{p}=vel.uind_shed+vel.uind_trail;
    end
```

```
    %Add the total induced velocity in the wake to the freestream velocity
    vel.domain{p}=vel.domain{p}+vel.uind{p};

%Numerically convect wake nodes to time+1
    if strcmp(user.integ,'fe') && p~=nt
        wake=fe(wake,vel,user,p); %Foward euler
    elseif strcmp(user.integ,'ab2') && p~=nt
        wake=ab2(wake,vel,user,p); %2nd-order Adams-Bashforth
    elseif strcmp(user.integ,'ab4') && p~=nt
        wake=ab4(wake,vel,user,p); %2nd-order Adams-Bashforth
    elseif strcmp(user.integ,'pcc') && p~=nt
        wake=pcc(wake,vel,const,fastout,user,p); %Predictor-corrector, central-difference
    end

%Compute strength of new bound vortex via Kutta-Joukowski theorem
    [wake,perf,vel,ctj]=KuttaJoukowski(pos,vel,blade,turbine,wake,airfoils,user,perf,p, ...
        user.kjtype);

%Determine time spent on current timeloop and estimate time remaining
    tm(p-1)=toc; %Time spent on current loop
    if p>2
        pt=polyfit([0 ; (2:p)'],cumsum([0 ; tm(1:p-1)]),2);
        tr=polyval(pt,nt)-sum(tm(1:p-1)); %Extrapolate to determine time remaining
        clc; disp([num2str(ctj) ': ' num2str(p/nt*100) ...
            '% complete, estimated time remaining: ' num2str(tr/60) ' minutes'])
    end
end

%% Compute performance metrics
perform;

%% Tidy up the workspace
clear yn j nb nt wb1 vs vt pg nst ns tr
save(['savedsims\' user.time '_' user.filename '_' user.rotor.casetype '.mat'])

%% Generate wake figure
if strcmp(user.anim,'true')
    j=length(fastout.Time);
    wakeplot(pos,vel,turbine,blade,wake,fastout,j);
end
```

4.2 BEM

BEM uses a steady implementation of the blade element momentum theory to generate an initial spanwise lift distribution on the rotor blades, which is then used to compute the initial vortex filament strengths.

```
function [cl,cd,phi,aoa,a,ap]=BEM(airfoils,blade,turbine,fastout,vel)
%% [cl,cd,phi,aoa,a,ap]=BEM(airfoils,blade,turbine,fastout,vel) -> BEM theory.
%
% Function computes spanwise and rotor performance and loads via blade
% element momentum theory. Includes corrections for skewed flow and
% heavily loaded rotors.
%
% ****Input(s)****
% airfoils  Structure containing airfoil performance tables
% blade     Structure containing blade geometry
% turbine   Structure containing turbine geometry
% fastout   Structure containing time-dependent kinematics
% vel       Structure containing velocity components in inertial and blade
%           coordinate systems
%
% ****Output(s)****
% cl        Spanwise lift coefficient
% cd        Spanwise drag coefficient
% phi       Spanwise inflow angle
```

```
% aoa       Spanwise angle of attack
% a         Spanwise axial induction factor
% ap        Spanwise tangential induction factor
%
%
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited January 15, 2011
%

%% Preallocate space for variables within loop
%Determine size of test vectors/arrays
ns=length(blade.RNodes);
nt=length(fastout.Time); %Number of timesteps
na=length(airfoils.Names);
RNodes=blade.RNodes;
Chord=blade.Chord;
NFoil=blade.NFoil;

a0=zeros(ns,nt); %Old (previous iteration) axial induction factor
ap0=zeros(ns,nt); %Old (previous iteration) tangential induction factor
phi=zeros(ns,nt); %Local inflow angle
aoa=zeros(ns,nt); %Local angle of attack
cl=zeros(ns,nt); %Local lift coefficient
cd=zeros(ns,nt); %Local drag coefficient
ct=zeros(ns,nt); %Local thrust coefficient
ftip=zeros(ns,nt); %Tip loss factor
fhub=zeros(ns,nt); %Hub loss factor
f=zeros(ns,nt); %Total loss corection factor
fiter=zeros(ns,nt); %Converence flag for gridpoints ('1' if converged, '9999' if not)

%% Define convergence criteria
tol=1e-6; %Convergence tolerance
da=ones(ns,nt); %Set initial value for axial induction factor residual equal to 1
dap=ones(ns,nt); %Set initial value for tangential induction factor residual equal to 1

ncv=find(da>tol | dap>tol); %Identify all nonconverged points (all initially)
miter=5000; %Maximum number of allowable iterations
wt=0.1; %Weighting factor on corrections to balance speed with stability (faster as you
        %approach 1, but less stable)

%% Compute relevant velocity/angle components
Uinf=sqrt(sum(vel.relhub.^2,2));
Om=fastout.RotSpeed*(2*pi/60);

twst=-blade.AeroTwst*pi/180;
ptch=fastout.BldPitch1*pi/180;

rP=-fastout.PtfmPitch*pi/180; %Rotor pitch (vector, wrt time)
rY=(fastout.PtfmYaw+fastout.NacYaw)*pi/180; %Rotor yaw (vector, wrt time)
if sign(rP)==0
    sp=sign(rY);
elseif sign(rY)==0
    sp=sign(rP);
else
    sp=sign(rP).*sign(rY);
end
gamma=sp.*acos(cos(rP).*cos(rY)); %Total skew angle
psi=pi-atan2(cos(rP).*sin(rY),sin(rP)); %Total azimuthal angle of skew

%% Compute initial guesses of key variables
Om=repmat(Om',ns,1);
```

```
Uinf=repmat(Uinf',ns,1);
ptch=repmat(ptch',ns,1);
gamma=repmat(gamma',ns,1);
psi=repmat(psi',ns,1);
twst=repmat(twst,1,nt);
sigmap=repmat(turbine.NumBl.*Chord./(2.*pi.*RNodes),1,nt); %Local solidity
RNodes=repmat(RNodes,1,nt);
lambdar=Om.*RNodes./Uinf; %Local speed ratio


% Initial values for axial and tangential induction factors
a=real(0.25*(2+pi*lambdar.*sigmap-sqrt(4-4*pi*lambdar.*sigmap+pi*lambdar.^2.*sigmap.* ...
    (8*(twst+ptch)+pi*sigmap))));
ap=zeros(size(a));

%% Primary loop for BEM

for j=1:200

    % Save previous values of axial and tangential induction factors
    a0(ncv)=a(ncv);
    ap0(ncv)=ap(ncv);

    % Compute inflow angle and angle of attack
    phi(ncv)=atan2(Uinf(ncv).*(1-a(ncv)),Om(ncv).*RNodes(ncv).*(1+ap(ncv)));
    aoa(ncv)=(phi(ncv)-(twst(ncv)+ptch(ncv)))*180/pi;

    % Interpolate over airfoil database for lift and drag coefficients
    for k=1:na
        cl(NFoil==k,:)=interp1(airfoils.profiles(k,1).AoA,airfoils.profiles(k,1).Cl, ...
            aoa(NFoil==k,:));
        cd(NFoil==k,:)=interp1(airfoils.profiles(k,1).AoA,airfoils.profiles(k,1).Cd, ...
            aoa(NFoil==k,:));
    end

    % Compute elemental thrust coefficient
    ct(ncv)=sigmap(ncv).*(1-a(ncv)).^2.*(cl(ncv).*cos(phi(ncv))+cd(ncv).*sin(phi(ncv)))./ ...
        sin(phi(ncv)).^2;

    % Compute loss correction factor due to tip and hub losses
    ftip(ncv)=2./pi.*acos(exp(-(turbine.NumBl.*(blade.TipRad-RNodes(ncv))./ ...
        (2.*RNodes(ncv).*sin(phi(ncv)))))); %Tip loss factor
    fhub(ncv)=2./pi.*acos(exp(-(turbine.NumBl.*(RNodes(ncv)-blade.HubRad)./ ...
        (2*blade.HubRad.*sin(phi(ncv)))))); %Hub loss factor
    f(ncv)=fhub(ncv).*ftip(ncv); %Total loss correction factor

    % Compute axial induction factor using conventional BEM theory
    a(ncv)=real((1+4.*f(ncv).*sin(phi(ncv)).^2./(sigmap(ncv).*(cl(ncv).*cos(phi(ncv))+ ...
        cd(ncv).*sin(phi(ncv))))).^-1);

    % Identify highly loaded gridpoints (requires use of modified Glauert correction for
    % axial induction factor)
    ncvf=find(ct>0.96*f & (da>tol | dap>tol));

    % Compute axial induction factor using modified Glauert correction (on identified gridpoints)
    a(ncvf)=real((18.*f(ncvf)-20-3.*sqrt(ct(ncvf).*(50-36.*f(ncvf))+12.*f(ncvf).* ...
        (3.*f(ncvf)-4)))./(36.*f(ncvf)-50));

    % Compute tangential induction factor
    ap(ncv)=(4.*f(ncv).*cos(phi(ncv)).*sin(phi(ncv))./(sigmap(ncv).*(cl(ncv).*sin(phi(ncv)) ...
        -cd(ncv).*cos(phi(ncv))))-1).^-1;

    %  Apply skewed wake correction if flow is non-axial
    if abs(gamma)>1e-8;
        a(ncv)=a(ncv).*(1+15*pi/32.*RNodes(ncv)./blade.TipRad.*tan(0.5.*(0.6.*a(ncv)+1).* ...
            gamma(ncv)).*cos(psi(ncv)));
    end

```

```
    % Compute residuals
    da(ncv)=abs(a0(ncv)-a(ncv));
    dap(ncv)=abs(ap0(ncv)-ap(ncv));

    % Apply corrective weighting for convergence stability
    if wt>0
        a(ncv)=a0(ncv)+wt.*(a(ncv)-a0(ncv));
        ap(ncv)=ap0(ncv)+wt.*(ap(ncv)-ap0(ncv));
    end

    % Clear all gridpoint flags in preparation for next loop
    clear ncv ncvf ncvcl ida idap

    % Identify nonconverged gridpoints
    ncv=find(da>tol | dap>tol);

    % If all points meet convergence criteria, break loop
    if isempty(ncv)
        break
    end

    % If maximum allowable iterations has been reached, flag nonconverged gridpoints
    % with '9999'
    if j==miter
        fiter(ncv)=9999;
    else
        fiter(ncv)=j;
    end
end
```

4.3 BiotSavart

The **BiotSavart** function computes the induced velocity at a point in space because of the influence of defined vortex filaments. Despite being written in a vectorized form (capitalizing on one of MATLAB's strengths), the majority of computational resources spent by WInDS during a simulation is on this function.

```
function [uind,L]=BiotSavart(F1,F2,P,gamma,rc,d,co,type)
%% uind=BiotSavart(F1,F2,P,gamma,rc,d,type) -> Biot-Savart Law
%
% Function computes the velocity contributions due to turbine motion and
% freestream flow in the inertial and blade coordinate systems.
%
% ****Input(s)****
% F1        Array containing first point of each vortex filament
% F2        Array containing second point of each vortex filament
% P         Array containing points of interest (where induction is
%           computed)
% gamma     Array of vortex filament circulation strengths
% rc        Vortex core sizes (actually radius squared for code speed-up)
% d         Squared cut-off distance (if =0, then viscous correction used)
% co        Distance from wake nodes beyond which influence is negligible
% type      If 'length', then will only output filament length (for
%           filament stretching correction), if 'full', will compute
%           induction on all points of interest
%
% ****Output(s)****
% uind      Array of induced velocity at each of the points P due to
%           contributions from filaments defined by F1 and F2
% L         Filament length
%
%
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
```

```
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited May 24, 2011
%

%% Relabel filament endpoint variables, preallocate memory
sp=size(P); %Size of 4D array containing induced velocity points
if length(sp)==2
    sp(3)=1;
end
if length(sp)==3
    sp(4)=1;
end
ns=sp(1);
nt=sp(3);
nb=sp(4);

uind=zeros(sp);

if strfind(d,'visc')
    n=str2double(d(5:end));
end

%Filament start points
x1=F1(:,1,:,:);
y1=F1(:,2,:,:);
z1=F1(:,3,:,:);
clear F1

%Filament end points
x2=F2(:,1,:,:);
y2=F2(:,2,:,:);
z2=F2(:,3,:,:);
clear F2

x2x1=x2-x1;
y2y1=y2-y1;
z2z1=z2-z1;
L=x2x1.^2+y2y1.^2+z2z1.^2; %Length of vortex filament (NOTE: L is L^2, as rc is rc^2)

if strcmp(type,'length') %If true, then only returns filament length
    L=sqrt(L);
    uind=zeros(size(P));
elseif strcmp(type,'full')

%% Begin looping over POIs
    for k=1:nb
        for j=1:nt
            for i=1:ns
                px=P(i,1,j,k);
                py=P(i,2,j,k);
                pz=P(i,3,j,k);

%% Compute vector difference calculations
                pxx1=px-x1;
                pyy1=py-y1;
                pzz1=pz-z1;
                pxx2=px-x2;
                pyy2=py-y2;
                pzz2=pz-z2;

%% Compute distances between points on triangle (filament to POI)
                r1=sqrt(pxx1.^2+pyy1.^2+pzz1.^2);
                r2=sqrt(pxx2.^2+pyy2.^2+pzz2.^2);
                r1dr2=pxx1.*pxx2+pyy1.*pyy2+pzz1.*pzz2;
                r1tr2=r1.*r2;
```

```

                if ~isnan(n)
                    Ldr12=(x2x1.*pxx1+y2y1.*pyy1+z2z1.*pzz1).^2;
                    Cnu=r1.^2-Ldr12./L;
                    Cnu=Cnu.*(rc.^n+Cnu.^n).^(-1/n);
                    ubar=Cnu.*gamma/(4*pi).*(r1+r2)./(r1tr2.*(r1tr2+r1dr2));
                else
                    ubar=gamma/(4*pi).*(r1+r2)./(r1tr2.*(r1tr2+r1dr2)+(d*L));
                end

                ubar(isnan(ubar) | isinf(ubar) | (r1>co & r2>co))=0;

                uind(i,1,j,k)=sum(sum(sum(ubar.*(pyy1.*pzz2-pzz1.*pyy2),1),3),4);
                uind(i,2,j,k)=sum(sum(sum(ubar.*(pzz1.*pxx2-pxx1.*pzz2),1),3),4);
                uind(i,3,j,k)=sum(sum(sum(ubar.*(pxx1.*pyy2-pyy1.*pxx2),1),3),4);
            end
        end
    end
end
```

4.4 DCMRot

Provided any sequence of rotations and corresponding axes, **DCMRot** will generate the associated direction cosine matrix (DCM) and perform the rotations on a given vector.

```
function [y,A]=DCMRot(x,t,A,rotseq,rev)
%% [y,A]=DCMRot(x,t,rotseq) -> Vector Rotation.
%
% Function performs a series of rotations about user-defined axes by
% user-defined angles over a series of vectors.
%
% ****Input(s)****
% x         1x3 (or Nx3) vector (array of vectors) to be rotated
% t         NxM array of rotation angles, where M=1..M corresponds to
%           1st-Mth rotation order (degrees)
% A         Nx9 array representing preceeding rotation matrix
% rotseq    String (length M)indicating order of rotation sequence (Example:
%           'xyzy' indicates a rotation first about the x-axis, then y, then
%           z, then y
% rev       Compute transpose of DCM, then compute reverse sequence (if=1)
%
% ****Output(s)****
% y         Nx3 array of rotated vectors
% A         Nx9 array representing rotation matrix
%
%
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited February 26, 2010
%

%% Generate direction cosine matrix for rotation sequence
if isempty(A)
    A=zeros(size(t,1),9); %Form an identity array
    A(:,1:4:9)=1;
end

%Generate diagonal 1's and off-diagonal 0's
f0=zeros(size(t,1),1);
```

```
f1=ones(size(t,1),1);

%Speed up calculations by computing trig functions once
sint=sind(t);
cost=cosd(t);

for c1=1:length(rotseq) %Loop over number of rotation sequences
    if strcmpi(rotseq(c1),'x')
        R=[f1 f0 f0 f0 cost(:,c1) -sint(:,c1) f0 sint(:,c1) cost(:,c1)];
    elseif strcmpi(rotseq(c1),'y')
        R=[cost(:,c1) f0 sint(:,c1) f0 f1 f0 -sint(:,c1) f0 cost(:,c1)];
    elseif strcmpi(rotseq(c1),'z')
        R=[cost(:,c1) -sint(:,c1) f0 sint(:,c1) cost(:,c1) f0 f0 f0 f1];
    end

    B(:,1)=sum(R(:,1:3).*A(:,1:3:7),2);
    B(:,2)=sum(R(:,1:3).*A(:,2:3:8),2);
    B(:,3)=sum(R(:,1:3).*A(:,3:3:9),2);
    B(:,4)=sum(R(:,4:6).*A(:,1:3:7),2);
    B(:,5)=sum(R(:,4:6).*A(:,2:3:8),2);
    B(:,6)=sum(R(:,4:6).*A(:,3:3:9),2);
    B(:,7)=sum(R(:,7:9).*A(:,1:3:7),2);
    B(:,8)=sum(R(:,7:9).*A(:,2:3:8),2);
    B(:,9)=sum(R(:,7:9).*A(:,3:3:9),2);
    A=B;
end

if rev==1 %Compute transpose of DCM to reverse rotation sequence
    B(:,1)=A(:,1);
    B(:,2)=A(:,4);
    B(:,3)=A(:,7);
    B(:,4)=A(:,2);
    B(:,5)=A(:,5);
    B(:,6)=A(:,8);
    B(:,7)=A(:,3);
    B(:,8)=A(:,6);
    B(:,9)=A(:,9);
    A=B;
end

%% Apply rotation sequence to vector elements
if size(x,1)<size(A,1) %If a single vector undergoing a series of rotation, expand for
                       %index multiplication
    x=repmat(x,size(A,1),1);
elseif size(x,1)>size(A,1) %If a single rotation seq. applied to multiple vectors, expand
                           %for index multiplication
    A=repmat(A,size(x,1),1);
end

y(:,1)=sum(A(:,1:3).*x(:,1:3),2);
y(:,2)=sum(A(:,4:6).*x(:,1:3),2);
y(:,3)=sum(A(:,7:9).*x(:,1:3),2);
```

4.5 FilamentMod

FilamentMod computes the effective vortex core radius because of filament stretching between time steps.

```
function wake=filamentmod(wake,time)
%% wake=filamentmod(wake,time) -> Core size due to filament stretching.
%
% Function computes the effective vortex filament core size due to filament
% stretching between timesteps.
%
% ****Input(s)****
% wake      Structure containing wake node positions, filament strengths,
%           vortex core radii, and vortex Reynolds number
```

```
% time      Index for current timestep
%
% ****Output(s)****
% wake      Structure containing wake node positions, filament strengths,
%           vortex core radii (updated), and vortex Reynolds number
%
%
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited February 20, 2011
%

%% Apply filament stretching if time index > 3
if time>3
    trailnew=sqrt(wake.length.trail{time}(:,:,2:end-1,:));
    trailold=sqrt(wake.length.trail{time-1}(:,:,2:end,:));
    shednew=sqrt(wake.length.shed{time}(:,:,2:end-1,:));
    shedold=sqrt(wake.length.shed{time-1}(:,:,2:end,:));

%% Compute strain of trailing and shed filaments
    wake.strain.trail=(trailnew-trailold)./trailold;
    wake.strain.shed=(shednew-shedold)./shedold;

    %Equations modified as rc and re_eff are squared
    wake.rc_eff.trail{time}(:,:,2:end-1,:)=wake.rc.trail{time}(:,:,2:end-1,:).* ...
        (1./(1+wake.strain.trail));
    wake.rc_eff.shed{time}(:,:,2:end-1,:)=wake.rc.shed{time}(:,:,2:end-1,:).* ...
        (1./(1+wake.strain.shed));
end
```

4.6 Initials

Initials preallocates memory and defines the initial conditions. This includes vortex strengths (via **BEM**) as well as initial rotor position.

```
function [wake,vel,perf]=initials(pos,vel,blade,turbine,wind,airfoils,fastout,const,user)
%% [wake,vel,perf]=initials(pos,vel,blade,turbine,wind,airfoils,fastout,const,user)
%  -> Define initial values.
%
% Function preallocates memory for wake and response structures and
% variables and computes initial results for the first timestep.
%
% ****Input(s)****
% pos       Structure containing relevant positions
% vel       Structure containing velocity components in inertial and blade
%           coordinate systems
% blade     Structure containing blade geometry
% turbine   Structure containing turbine geometry
% wind      Structure containing imported wind data
% airfoils  Structure containing airfoil performance tables
% fastout   Structure containing imported FAST-generated results
% const     Structure containing model and atmospheric constants
% user      Structure containing user-defined variables
%
% ****Output(s)****
% wake      Structure containing wake node positions, filament strengths,
%           vortex core radii, and vortex Reynolds number
% vel       Structure containing velocity components in inertial and blade
%           coordinate systems, now including induced velocity
```

```
% perf      Structure containing performance-related variables
%
%
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited February 18, 2011
%

%% Preallocate for speed
%Determine size of test vectors/arrays
nt=length(fastout.Time); %Number of timesteps
nb=turbine.NumBl; %Number of blades
nst=length(blade.RTrail); %Number of trailing nodes (+1 number of station)
ns=length(blade.RNodes); %Number of shed nodes (stations)

wake.domain=cell(nt,1);
wake.domain(1:nt)={zeros([nst 3 nt+1 nb])};
vel.domain=cell(nt,1);
vel.domain(1:nt)={zeros([nst 3 nt+1 nb])};
vel.uind=cell(nt,1);
vel.uind(1:nt)={zeros([nst 3 nt+1 nb])};
vel.uindb=cell(nt,1);
vel.uindb(1:nt)={zeros([nst 3 nt+1 nb])};

wake.Re.shed=cell(nt,1);
wake.Re.shed(1:nt)={zeros([ns,1,nt+1,nb])};
wake.Re.trail=cell(nt,1);
wake.Re.trail(1:nt)={zeros([nst,1,nt,nb])};

wake.rc.shed=cell(nt,1);
wake.rc.shed(1:nt)={zeros([ns,1,nt+1,nb])};
wake.rc.trail=cell(nt,1);
wake.rc.trail(1:nt)={zeros([nst,1,nt,nb])};

wake.length.shed=cell(nt,1);
wake.length.shed(1:nt)={zeros([ns,1,nt+1,nb])};
wake.length.trail=cell(nt,1);
wake.length.trail(1:nt)={zeros([nst,1,nt+1,nb])};

wake.rc_eff.shed=cell(nt,1);
wake.rc_eff.shed(1:nt)={zeros([ns,1,nt+1,nb])};
wake.rc_eff.trail=cell(nt,1);
wake.rc_eff.trail(1:nt)={zeros([nst,1,nt,nb])};

wake.gamma.shed=cell(nt,1);
wake.gamma.shed(1:nt)={zeros([ns,1,nt+1,nb])};
wake.gamma.trail=cell(nt,1);
wake.gamma.trail(1:nt)={zeros([nst,1,nt+1,nb])};

perf.cl=zeros([ns,1,nt,nb]);
perf.cd=zeros([ns,1,nt,nb]);
perf.aoa=zeros([ns,1,nt,nb]);
perf.beta=zeros([ns,1,nt,nb]);

%% Substitute in initial values and truncate size of variables by timestep
for j=1:nt
    wake.domain{j}(:,:,1,:)=pos.quarter(:,:,j,:);
    wake.domain{j}(:,:,2,:)=pos.trail(:,:,j,:);
    wake.domain{j}(:,:,j+2:end,:)=[];

    vel.domain{j}(:,:,1:j+1,:)=repmat(wind.infty(j,:),[nst 1 j+1 nb]);
    vel.domain{j}(:,:,j+2:end,:)=[];
```

```
    vel.uind{j}(:,:,j+2:end,:)=[];

    wake.Re.shed{j}(:,:,j+2:end,:)=[];
    wake.Re.trail{j}(:,:,j+1:end,:)=[];

    wake.rc.shed{j}(:,:,j+2:end,:)=[];
    wake.rc.trail{j}(:,:,j+1:end,:)=[];

    wake.length.shed{j}(:,:,j+2:end,:)=[];
    wake.length.trail{j}(:,:,j+1:end,:)=[];

    wake.rc_eff.shed{j}(:,:,j+2:end,:)=[];
    wake.rc_eff.trail{j}(:,:,j+1:end,:)=[];

    wake.gamma.shed{j}(:,:,j+2:end,:)=[];
    wake.gamma.trail{j}(:,:,j+1:end,:)=[];
end

%% Define initial induced velocities via 1st-order methods
aoa=pos.aoag(:,1,1);
if strcmp(user.filename,'elliptical')
    cl=2*pi/(1+2/turbine.ellip.AR)*aoa*pi/180;
    perf.cl(:,1,1,1:nb)=repmat(cl,[1 1 1 nb]);
    perf.aoa(:,1,1,1:nb)=repmat(aoa,[1 1 1 nb]);
else
    [perf.bem.cl,perf.bem.cd,perf.bem.phi,perf.bem.aoa,perf.bem.a]=BEM(airfoils, ...
        blade,turbine,fastout,vel);
    perf.cl(:,1,1,1:nb)=repmat(perf.bem.cl(:,1),[1 1 1 nb]);
    perf.aoa(:,1,1,1:nb)=repmat(perf.bem.aoa(:,1),[1 1 1 nb]);
end

%% Define initial vortex strength
%Use Kutta-Joukowski theorem to define bound circulation strength
wake.gamma.shed{1}(:,:,1,:)=0.5*wind.inftyM(1).*repmat(blade.Chord,[1 1 1 nb]).* ...
    perf.cl(:,:,1,:);
%Compute spanwise change in bound filament to compute first set of trailing filaments
wake.gamma.trail{1}=diff([zeros(1,1,1,nb) ; wake.gamma.shed{1}(:,:,1,:) ; zeros(1,1,1,nb)],1);
%Shed filaments computed via spanwise summation of trailing filaments (ensure Kelvin's theorem
%is satisfied)
wake.gamma.shed{1}(:,:,2:end,:)=diff(cat(3,cumsum(wake.gamma.trail{1}(1:end-1,:,:,:),1), ...
    zeros(ns,1,1,nb)),1,3);

%% Define initial vortex core size
T0=2*pi*blade.TipRad./(12*fastout.TipSpdRat.*wind.inftyM);
wake.r0=sqrt(4*const.alpha*const.nu*const.delta*T0);

%% Modify core size using Ramasamy-Leishman model
wake=vcore(wake,const,fastout,user,1);

%% Compute induced velocity at all points in domain and convect points to next timestep
%Velocity induced by shed filaments on all nodes in wake
vel.uind_shed=BiotSavart(wake.domain{1}(1:end-1,:,:,:),wake.domain{1}(2:end,:,:,:), ...
    wake.domain{1},wake.gamma.shed{1},wake.rc_eff.shed{1},user.d,user.co,'full');
%Velocity induced by trailing filaments on all nodes in wake
vel.uind_trail=BiotSavart(wake.domain{1}(:,:,2:end,:),wake.domain{1}(:,:,1:end-1,:), ...
    wake.domain{1},wake.gamma.trail{1},wake.rc_eff.trail{1},user.d,user.co,'full');
%Sum the induced velocity contributions due to shed and trailing filaments
vel.uind{1}=vel.uind_shed+vel.uind_trail;
%Add the total induced velocity in the wake to the freestream velocity
vel.domain{1}=vel.domain{1}+vel.uind{1};
%Numerically convect wake nodes to time+1 via forward Euler
wake=fe(wake,vel,user,1);
```

4.7 InputImport

InputImport imports turbine geometry, operating conditions, and airfoil properties directly from the user-selected FAST input files.

```
function [airfoils,blade,turbine,platform,wind]=input_import(filename)
%% [airfoils,blade,turbine,platform,wind]=input_import(filename) -> FAST input files importer.
%
% Function imports FAST simulation input files
%
% ****Input(s)****
% filename  String containing path to FAST input file (.fst)
%
% ****Output(s)****
% airfoils  Structure containing airfoil performance tables
% blade     Structure containing blade geometry from FAST input file
% turbine   Structure containing turbine geometry from FAST input file
% platform  Structure containing platform geometry from FAST input file
% wind      Structure containing wind data file location
%
%
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited February 23, 2010
%

%% Use FAST input file to ID other relevant files
fn=strread(char(filename),'%s','delimiter','\\');
fstfile=char(fn(end));
fstpath=filename(1:end-length(fstfile));
turbine.filename=[fstpath fstfile];
data=importdata(turbine.filename,'\t'); %Import FAST input file

% Identify platform property file
pf=sscanf(char(data(131)),'%i');
if pf>=2
platform.filename=strread(char(data(132)),'%s','delimiter','"');
platform.filename=[fstpath char(platform.filename(2))];
else
platform.filename='No platform model used.';
end

% Identify AeroDyn file
blade.filename=strread(char(data(161)),'%s','delimiter','"');
blade.filename=[fstpath char(blade.filename(2))];

%% Import turbine and blade properties from FAST input file
blade.TipRad=sscanf(char(data(78)),'%f');
blade.HubRad=sscanf(char(data(79)),'%f');
turbine.NumBl=sscanf(char(data(9)),'%i');
turbine.OverHang=sscanf(char(data(83)),'%f');
turbine.TowerHt=sscanf(char(data(87)),'%f');
turbine.Twr2Shft=sscanf(char(data(88)),'%f');
turbine.ShftTilt=sscanf(char(data(90)),'%f');
turbine.PreCone(1)=sscanf(char(data(92)),'%f');
turbine.PreCone(2)=sscanf(char(data(93)),'%f');
turbine.PreCone(3)=sscanf(char(data(94)),'%f');
clear data

%% Import AeroDyn file and individual airfoil files
% Identify TurbSim-based wind input file
```

```
data=importdata(blade.filename,'\t');
wind.filename=strread(char(data(10)),'%s','delimiter','"');
wind.filename=[fstpath char(wind.filename(2))];

% Count up number of airfoils and blade sections, import airfoil tables,
% then import blade properties as a structure
nblades=sscanf(char(data(18)),'%i');
airfoils.Names=cell(nblades,1);
for c1=1:nblades
    af=strread(char(data(18+c1)),'%s','delimiter','"');
    af=char(af(2));
    adata=importdata([fstpath af],'\t');

    if(isfield(adata,'textdata')==0) %Sometimes will import a cell structure, check for this
        adata1=importdata([fstpath af],' ',14);
        adata2=importdata([fstpath af],'\t');
        adata2(15:end)=[];
        clear adata
        adata.data=adata1.data;
        adata.textdata=adata2;
    end

    id=isnan(adata.data(2,:));
    adata.data(:,id)=[];

    af=strread(char(af),'%s','delimiter','\\');
    af=char(af(end));

    airfoils.Names(c1,1)={genvarname(af(1:end-4))};

    id=find(diff(adata.data(:,1))==0); %ID non-distinct values for AoA
    adata.data(id,:)=[]; %#ok<FNDSB>

    eval(['airfoils.profiles(' num2str(c1) ' ,1).StallAoA=' ...
        'sscanf(char(adata.textdata(5)),''%f'');'])
    eval(['airfoils.profiles(' num2str(c1) ' ,1).Cn0AoA=' ...
        'sscanf(char(adata.textdata(9)),''%f'');'])
    eval(['airfoils.profiles(' num2str(c1) ' ,1).Lift0Cn=' ...
        'sscanf(char(adata.textdata(10)),''%f'');'])
    eval(['airfoils.profiles(' num2str(c1) ' ,1).StallAoACn=' ...
        'sscanf(char(adata.textdata(11)),''%f'');'])
    eval(['airfoils.profiles(' num2str(c1) ' ,1).StallAoANCn=' ...
        'sscanf(char(adata.textdata(12)),''%f'');'])
    eval(['airfoils.profiles(' num2str(c1) ' ,1).CdminAoA=' ...
        'sscanf(char(adata.textdata(13)),''%f'');'])
    eval(['airfoils.profiles(' num2str(c1) ' ,1).Cdmin=' ...
        'sscanf(char(adata.textdata(14)),''%f'');'])

    eval(['airfoils.profiles(' num2str(c1) ' ,1).AoA=adata.data(:,1);'])
    eval(['airfoils.profiles(' num2str(c1) ' ,1).Cl=adata.data(:,2);'])
    eval(['airfoils.profiles(' num2str(c1) ' ,1).Cd=adata.data(:,3);'])
    eval(['airfoils.profiles(' num2str(c1) ' ,1).Cm=adata.data(:,4);'])
    clear af adata adata1 adata2 id
end

dm=19+nblades;
ivnames=textscan(char(data(dm+1,:)),'%s');
ivnames=genvarname(cell(ivnames{1,1}));
data=char(data(dm+1:length(data),:));
ndata=zeros(size(data,1)-1,5);
for c2=2:size(data,1)
    ndata(c2-1,:)=sscanf(data(c2, :)', '%f %f %f %f %d', [1, inf]);
end
for c3=1:5
    eval(['blade.' char(ivnames(c3)) '=ndata(:,c3);'])
end

%% Import platform properties
```

```
if pf==0 || pf==1
    platform.Type='onshore';
    platform.TwrDraft=0;
    platform.PtfmCM=0;
    platform.PtfmRef=0;
    platform.PtfmDraft=0;
    platform.PtfmDiam=0;
elseif pf==2
    data=importdata(platform.filename,'\t');
    platform.Type='fixedoffshore';
    platform.TwrDraft=sscanf(char(data(19)),'%f');
    platform.PtfmCM=sscanf(char(data(20)),'%f');
    platform.PtfmRef=sscanf(char(data(21)),'%f');
    platform.PtfmDraft=sscanf(char(data(36)),'%f'); %Water depth
    platform.PtfmDiam=sscanf(char(data(31)),'%f');
elseif pf==3
    data=importdata(platform.filename,'\t');
    platform.Type=strread(char(data(29)),'%s','delimiter','"');
    platform.Type=strread(char(platform.Type(2)),'%s','delimiter','\\');
    platform.Type=char(platform.Type(end));
    platform.TwrDraft=sscanf(char(data(19)),'%f');
    platform.PtfmCM=sscanf(char(data(20)),'%f');
    platform.PtfmRef=sscanf(char(data(21)),'%f');
    platform.PtfmDraft=sscanf(char(data(32)),'%f');
    platform.PtfmDiam=sscanf(char(data(33)),'%f');
end
```

4.8 Kinematics

Kinematics works with **DCMRot** to compute the locations of spanwise points of interest in the inertial and blade coordinate systems.

```
function [pos]=kinematics(blade,turbine,platform,fastout)
%% [pos]=kinematics(blade,turbine,platform,fastout)
%  -> Inertial position of rotor and blade stations.
%
% Function computes the station locations of each blade in the inertial
% coordinate system
%
% ****Input(s)****
% blade     Structure containing blade geometry from FAST input file
% turbine   Structure containing turbine geometry from FAST input file
% platform  Structure containing platform geometry from FAST input file
% fastout   Structure containing imported FAST-generated results
%
% ****Output(s)****
% pos       Structure containing relevant positions
%
%
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited March 25, 2010
%

%% Position of platform reference point in inertial coordinate system
pos.platform=[fastout.PtfmSurge fastout.PtfmSway fastout.PtfmHeave];

%% Position of rotor cone apex (hub) in inertial coordinate system
hx=turbine.OverHang*cosd(turbine.ShftTilt);
```

```
hy=0;
hz=platform.PtfmRef+turbine.TowerHt+turbine.Twr2Shft+turbine.OverHang*sind(turbine.ShftTilt);
hub_nominal=[hx hy hz]; %Coordinates of hub in ICS

%Rotation sequence for hub in ICS due to platform+nacelle motions
hub_rotseq=[fastout.PtfmYaw fastout.PtfmPitch fastout.PtfmRoll fastout.NacYaw];
hub_rotated=DCMRot(hub_nominal,hub_rotseq,[],'zyxz',0);
pos.hub=pos.platform+hub_rotated;

%% Position of spanwise stations and nodes in inertial coordinate system
nt=length(fastout.Time); %Number of timesteps
nb=turbine.NumBl; %Number of blades
nst=length(blade.RTrail); %Number of trailing nodes (+1 number of station)
ns=length(blade.RNodes); %Number of shed nodes (stations)

%Blade stations defined radially along z-axis
blade_lead=[-0.25*blade.ChordTrail zeros(nst,1) blade.RTrail];
blade_bound=[zeros(ns,1) zeros(ns,1) blade.RNodes];
blade_colloc=[0.25*blade.Chord zeros(ns,1) blade.RNodes];
blade_quarter=[zeros(nst,1) zeros(nst,1) blade.RTrail];
blade_trail=[0.75*blade.ChordTrail zeros(nst,1) blade.RTrail];
blade_end=[0.75*blade.Chord zeros(ns,1) blade.RNodes];

%Rotation sequence from rotor to inertial coordinate system
rotor_rotseq=[fastout.Azimuth turbine.ShftTilt*ones(nt,1) flipdim(hub_rotseq,2)];

%Preallocate for speed
pos.lead=zeros(nst,3,nt,nb);
pos.bound=zeros(ns,3,nt,nb);
pos.colloc=zeros(ns,3,nt,nb);
pos.quarter=zeros(nst,3,nt,nb);
pos.trail=zeros(nst,3,nt,nb);
pos.end=zeros(ns,3,nt,nb);
if strcmp(platform.Type,'EllipticalWing')
    pos.blade_rotseq=zeros(nt,9,nb);
else
    pos.blade_rotseq=zeros(nt,10,nb);
end

%Determine azimuth angle between blades, using # of blades
Azstep=360/nb;
Az=[0 cumsum(Azstep*ones(1,nb-1))];

if turbine.NumBl==2
    fastout.BldPitch(:,1)=fastout.BldPitch1;
    fastout.BldPitch(:,2)=fastout.BldPitch2;
elseif turbine.NumBl==3
    fastout.BldPitch(:,1)=fastout.BldPitch1;
    fastout.BldPitch(:,2)=fastout.BldPitch2;
    fastout.BldPitch(:,3)=fastout.BldPitch3;
end

if strcmp(platform.Type,'EllipticalWing')
    rseq='zzyxxyzxyz';
else
    rseq='zzzyxxyzxyz';
end

for c1=1:nb %Blade-specific rotation sequences
    if strcmp(platform.Type,'EllipticalWing')
        pos.blade_rotseq(:,:,c1)=[fastout.BldPitch(:,c1) turbine.PreCone(c1)*ones(nt,1) ...
            Az(c1)*ones(nt,1) rotor_rotseq];
    else
        pos.blade_rotseq(:,:,c1)=[90*ones(nt,1) fastout.BldPitch(:,c1) ...
            turbine.PreCone(c1)*ones(nt,1) Az(c1)*ones(nt,1) rotor_rotseq];
    end
    for c2=1:ns
        total_rotseq=[blade.AeroTwst(c2)*ones(nt,1) pos.blade_rotseq(:,:,c1)];
```

```
        pos.bound(c2,1:3,:,c1)=DCMRot(blade_bound(c2,:),total_rotseq,[],rseq,0)'+pos.hub';
        pos.end(c2,1:3,:,c1)=DCMRot(blade_end(c2,:),total_rotseq,[],rseq,0)'+pos.hub';
        pos.colloc(c2,1:3,:,c1)=DCMRot(blade_colloc(c2,:),total_rotseq,[],rseq,0)'+pos.hub';
    end
    for c2=1:nst
        total_rotseq=[blade.AeroTwstTrail(c2)*ones(nt,1) pos.blade_rotseq(:,:,c1)];
        pos.lead(c2,1:3,:,c1)=DCMRot(blade_lead(c2,:),total_rotseq,[],rseq,0)'+pos.hub';
        pos.quarter(c2,1:3,:,c1)=DCMRot(blade_quarter(c2,:),total_rotseq,[],rseq,0)'+pos.hub';
        pos.trail(c2,1:3,:,c1)=DCMRot(blade_trail(c2,:),total_rotseq,[],rseq,0)'+pos.hub';
    end
end
```

4.9 KuttaJoukowski

KuttaJoukowski converges to the spanwise circulation distribution because of wake–induced inflow via user–selected root finding approaches.

```
function [wake,perf,vel,j]=KuttaJoukowski(pos,vel,blade,turbine,wake,airfoils, ...
    user,perf,time,type)
%% [wake,perf,vel]=KuttaJoukowski(pos,vel,blade,turbine,wake,airfoils,user,perf,time)
%  -> Kutta-Joukowski solver.
%
% Function computes the bound vortex filament strength via Kutta-Joukowski
% theorem, solving via fixed-point iteration or Brent's method
%
% ****Input(s)****
% pos       Structure containing relevant positions
% vel       Structure containing velocity components in inertial and blade
%           coordinate systems
% blade     Structure containing blade geometry
% turbine   Structure containing turbine geometry
% wake      Structure containing wake node positions, filament strengths,
%           vortex core radii, and vortex Reynolds number
% airfoils  Structure containing airfoil performance tables
% user      Structure containing user-defined variables
% perf      Structure containing performance-related variables
% time      Index for current timestep
% type      If 'fixed', will use fixed-point iteration, if 'brent', will
%           use Brent's method
%
% ****Output(s)****
% wake      Structure containing wake node positions, filament strengths
%           (updated), vortex core radii, and vortex Reynolds number
% perf      Structure containing performance-related variables (updated)
% vel       Structure containing velocity components in inertial and blade
%           coordinate systems
%
%
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited February 20, 2011
%

%% Check condition for fixed-point iteration or Brent's method
if strcmp(type,'fixed')
    j=0;
    dg=1;
    while max(max(abs(dg)))>user.tol & j<user.maxiter %#ok<AND2> %Fixed-point iteration
        gamma=wake.gamma.shed{time}(:,:,1,:);
```

```
        [dg,wake,perf,vel]=kj(gamma,vel,wake,pos,blade,turbine,perf,airfoils,time,user);
        j=j+1;
    end
elseif strcmp(type,'brent')

%% Iteration via Brent's method
    %Preallocate for speed
    flag=zeros(6,1); %Space for logical values from conditional tests
    na=length(airfoils.Names); %Number of airfoils
    nb=turbine.NumBl; %Number of blades
    Vinf=sqrt(sum(vel.blade(:,:,time-1,:).^2,2)); %Magnitude of wind at the blade

    %Loop over airfoils + blades, interpolate wrt AoA to determine Cl and Cd
    %Adjust AoA +/-10-degrees to set upper/lower bounds for Brent's method
    aoa=perf.aoa(:,:,time-1,:);
    cla=perf.cl(:,:,time-1,:);
    clb=perf.cl(:,:,time-1,:);
    dalpha=1;
    for k=1:na
        for m=1:nb
            cla(blade.NFoil==k,1,1,m)=interp1(airfoils.profiles(k,1).AoA, ...
                airfoils.profiles(k,1).Cl,squeeze(aoa(blade.NFoil==k,1,1,m)-dalpha));
            clb(blade.NFoil==k,1,1,m)=interp1(airfoils.profiles(k,1).AoA, ...
                airfoils.profiles(k,1).Cl,squeeze(aoa(blade.NFoil==k,1,1,m)+dalpha));
        end
    end

    a=0.5*Vinf.*repmat(blade.Chord,[1 1 1 turbine.NumBl]).*cla;
    b=0.5*Vinf.*repmat(blade.Chord,[1 1 1 turbine.NumBl]).*clb;

    fa=kj(a,vel,wake,pos,blade,turbine,perf,airfoils,time,user);
    fb=kj(b,vel,wake,pos,blade,turbine,perf,airfoils,time,user);
    fs=ones(size(fb));

    %Check that bounds are opposite signs (soln must be between bounds)
    if any(fa(2:end-1,:,:,:).*fb(2:end-1,:,:,:)>0);
        j=0;
        dg=1;
        while max(max(abs(dg)))>user.tol & j<user.maxiter %#ok<AND2> %Fixed-point iteration
            gamma=wake.gamma.shed{time}(:,:,1,:);
            [dg,wake,perf,vel]=kj(gamma,vel,wake,pos,blade,turbine,perf,airfoils,time,user);
            j=j+1;
        end
        return
    end

    %If any values are zero (Cl=0, for example), then no sign... assign
    %+/-1 depending on the number of +/- values in bound
    if any(fa.*fb==0);
        if numel(fa<0)>numel(fa>0)
            fa(fa==0)=-1;
            fb(fb==0)=1;
        else
            fa(fa==0)=1;
            fb(fb==0)=-1;
        end
    end

    %Set |fb| < |fa|
    if abs(fa(mid(fa)))<abs(fb(mid(fb)));
        [b,a,fb,fa]=deal(a,b,fa,fb);
    end

    %Set initial values and conditions
    c=a;
    fc=fa;
    flag(1)=true;
    j=0;
```

```
%% Iterate until convergence or max. iterations reached
    while max(abs(fs))>user.tol & j<user.maxiter %#ok<AND2>
        flag(2)=all(all(fa~=fc)) && all(all(fb~=fc));
        if flag(2) %Inverse quadratic interpolation
            s=a.*fb.*fc./((fa-fb).*(fa-fc))+b.*fa.*fc./((fb-fa).*(fb-fc))+c.*fa.*fb./ ...
                ((fc-fa).*(fc-fb));
        else %Secant rule
            s=b-fb.*(b-a)./(fb-fa);
        end

        t1=0.25*(3*a+b);
        t2=b;
        if t2(mid(t2))<t1(mid(t1));
            [t2,t1]=deal(t1,t2);
        end

        %Conditional flags for method(s) used
        flag(3)=~(t1(mid(t1))<s(mid(s)) && s(mid(s))<t2(mid(t2)));
        flag(4)=flag(1) && abs(s(mid(s))-b(mid(b)))>=0.5*abs(b(mid(b))-c(mid(c)));
        flag(5)=~flag(1) && abs(s(mid(s))-b(mid(b)))>=0.5*abs(c(mid(c))-d(mid(d)));
        flag(6)=flag(1) && abs(b(mid(b))-c(mid(c)))<user.tol;
        flag(7)=~flag(1) && abs(c(mid(c))-d(mid(d)))<user.tol;

        if any(flag(3:7))
            s=0.5*(a+b); %Bisection method
            flag(1)=true;
        else
            flag(1)=false;
        end

        %Apply Kutta-Joukowski theorem to bound filament strength 's'
        [fs,wake,perf,vel]=kj(s,vel,wake,pos,blade,turbine,perf,airfoils,time,user);
        s=wake.gamma.shed{time}(:,:,1,:);
        d=c;
        c=b;
        fc=fb;

        %Swap to set new bounds
        if any(fa.*fs<0)
            b=s;
            fb=fs;
        else
            a=s;
            fa=fs;
        end

        %Set |fb| < |fa|
        if abs(fa(mid(fa)))<abs(fb(mid(fb)));
            [b,a,fb,fa]=deal(a,b,fa,fb);
        end

        j=j+1;
    end
end
end


function [dg,wake,perf,vel]=kj(gamma,vel,wake,pos,blade,turbine,perf,airfoils,time,user)
%% [dg,wake,perf,vel]=kj(gamma,vel,wake,pos,blade,turbine,perf,airfoils,time,user)
% -> Kutta-Joukowski theorem.
%
% Function computes the bound vortex filament strength via Kutta-Joukowski
% theorem, solving via fixed-point iteration or Brent's method
%
% ****Input(s)****
% pos       Structure containing relevant positions
% vel       Structure containing velocity components in inertial and blade
```

```
%           coordinate systems
% blade     Structure containing blade geometry
% turbine   Structure containing turbine geometry
% wake      Structure containing wake node positions, filament strengths,
%           vortex core radii, and vortex Reynolds number
% airfoils  Structure containing airfoil performance tables
% user      Structure containing user-defined variables
% perf      Structure containing performance-related variables
% time      Index for current timestep
%
% ****Output(s)****
% wake      Structure containing wake node positions, filament strengths
%           (updated), vortex core radii, and vortex Reynolds number
% perf      Structure containing performance-related variables (updated)
% vel       Structure containing velocity components in inertial and blade
%           coordinate systems
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited February 20, 2011
%

%% Preallocate for speed
%Determine size of test vectors/arrays
na=length(airfoils.Names); %Number of airfoils
nb=turbine.NumBl; %Number of blades
ns=length(blade.RNodes); %Number of shed nodes (stations)
cl=perf.cl(:,:,time-1,:);
cd=perf.cd(:,:,time-1,:);
vel.rot=zeros(size(vel.blade(:,:,time,:)));
wake.gamma.shed{time}(:,:,1,:)=gamma;

%% Compute induced velocity on lifting line due to shed and trailing filament induction
vel.uindb_shed=BiotSavart(wake.domain{time}(1:end-1,:,:,:),wake.domain{time}(2:end,:,:,:), ...
    pos.bound(:,:,time,:),wake.gamma.shed{time},wake.rc_eff.shed{time},user.d,user.co,'full');
vel.uindb_trail=BiotSavart(wake.domain{time}(:,:,2:end,:),wake.domain{time}(:,:,1:end-1,:), ...
    pos.bound(:,:,time,:),wake.gamma.trail{time},wake.rc_eff.trail{time},user.d,user.co,'full');
vel.uindb=vel.uindb_shed+vel.uindb_trail;

%% Perform coordinate transformation on induced velocity (inertial to blade)
vel.rot(:,1,:,:)=pos.nodes.bxn(:,1,time,:).*vel.uindb(:,1,:,:)+pos.nodes.bxn(:,2,time,:).* ...
    vel.uindb(:,2,:,:)+pos.nodes.bxn(:,3,time,:).*vel.uindb(:,3,:,:);
vel.rot(:,2,:,:)=pos.nodes.byn(:,1,time,:).*vel.uindb(:,1,:,:)+pos.nodes.byn(:,2,time,:).* ...
    vel.uindb(:,2,:,:)+pos.nodes.byn(:,3,time,:).*vel.uindb(:,3,:,:);
vel.rot(:,3,:,:)=pos.nodes.bzn(:,1,time,:).*vel.uindb(:,1,:,:)+pos.nodes.bzn(:,2,time,:).* ...
vel.uindb(:,2,:,:)+pos.nodes.bzn(:,3,time,:).*vel.uindb(:,3,:,:);

%% Compute effective wind in blade coordinate system
vel.tot=vel.blade(:,:,time,:)+vel.rot;
u=vel.tot(:,1,:,:);
v=vel.tot(:,2,:,:);
w=vel.tot(:,3,:,:);

Vinf=sqrt(sum(vel.blade(:,:,time,:).^2,2));
Vtot=sqrt(sum(vel.tot.^2,2));

%% Compute angle of attack and sideslip angle
aoa=atan2(-v,u)*(180/pi);
beta=asind(w./Vtot);

%% Interpolate over airfoil data tables
for k=1:na
    for m=1:nb
        cl(blade.NFoil==k,1,1,m)=interp1(airfoils.profiles(k,1).AoA, ...
            airfoils.profiles(k,1).Cl,squeeze(aoa(blade.NFoil==k,1,1,m)));
        cd(blade.NFoil==k,1,1,m)=interp1(airfoils.profiles(k,1). ...
            AoA,airfoils.profiles(k,1).Cd,squeeze(aoa(blade.NFoil==k,1,1,m)));
    end
end
```

```
%Check for NaN values of Cl
if any(isnan(cl));
    error('Diverging soln!!!');
end

%% Compute bound vorticity via Kutta—Joukowski theorem
gamma=0.5*Vinf.*repmat(blade.Chord,[1 1 1 turbine.NumBl]).*cl;
dg=gamma—wake.gamma.shed{time}(:,:,1,:); %Change in bound vorticity between iterations

if strcmp(user.kjtype,'fixed')
    wake.gamma.shed{time}(:,:,1,:)=wake.gamma.shed{time}(:,:,1,:)+user.relax*dg;
else
    wake.gamma.shed{time}(:,:,1,:)=gamma;
end
wake.gamma.trail{time}(:,:,1,:)=diff([zeros(1,1,1,nb) ; wake.gamma.shed{time}(:,:,1,:) ; ...
    zeros(1,1,1,nb)],1);
wake.gamma.shed{time}(:,:,2:end,:)=diff(cat(3,cumsum(wake.gamma.trail{time} ...
    (1:end—1,:,:,:),1),zeros(ns,1,1,nb)),1,3);

dg=dg./(abs(gamma)+1);

%% Compute performance variables and coefficients
perf.cl(:,:,time,:)=cl;
perf.cd(:,:,time,:)=cd;
perf.aoa(:,:,time,:)=aoa;
perf.beta(:,:,time,:)=beta;

end
```

4.10 OutputImport

OutputImport imports the FAST–generated platform kinematics and performance results these values.

```
function [fastout]=output_import(filename,t)
%% [fastout]=output_import(filename,t) —> FAST—generated output importer.
%
% Function imports FAST output files and interpolates time—series data to
% user—specifications.
%
% ****Input(s)****
% filename  String containing path to FAST input file (.fst)
% t         1x3 vector containing initial and final times and frequency
%
% ****Output(s)****
% fastout   Structure containing imported FAST—generated results
%
%
% This work is licensed under the Creative Commons Attribution—ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by—sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited February 23, 2010
%

%% Use FAST input file to ID other relevant files
data=importdata(filename,'\t'); %Import FAST input file

% Determine if Simulink—derived results or not
simq=sscanf(char(data(13)),'%i');
if simq==2 % Output file name based on use of Simulink or executable
    fastout.filename=[filename(1:end—4) '_SFunc.out'];
```

```
else
    fastout.filename=[filename(1:end-3) 'out'];
end

dt=sscanf(char(data(11)),'%f'); %Integration time step in FAST

%% Import FAST output
if exist(fastout.filename,'file')
    data=importdata(fastout.filename,'\t',7);
else
    fastout.filename=[fastout.filename(1:end-4) '_Sfunc.out'];
    data=importdata(fastout.filename,'\t',7);
    simq=2;
end
ovnames=genvarname(data(7,:)'); %Identify output variable names
odata=importdata(fastout.filename,'\t',7+1);
if simq~=2
    odata.data(:,1)=(odata.data(1,1):dt:odata.data(end,1))';
end

%% Interpolate to user-defined times
if t(3)==0 %If user-selected freq is zero, then use freq that the data is sampled at
    t(3)=1/(mean(diff(odata.data(1,:))));
end

if t(1)<odata.data(1,1);
    t(1)=odata.data(1,1);
    disp(['User selected initial time out-of-range, reset to ' num2str(t(1)) ' seconds.'])
    disp(' ')
end
if t(2)>odata.data(end,1);
    t(2)=odata.data(end,1);
    disp(['User selected final time out-of-range, reset to ' num2str(t(2)) ' seconds.'])
    disp(' ')
end
odatai=interp1(odata.data(:,1),odata.data(:,2:end),(t(1):1/t(3):t(2))');
odatai=[(t(1):1/t(3):t(2))' odatai]; %#ok<NASGU>
for c1=1:length(ovnames)
    eval(['fastout.' char(ovnames(c1)) '=odatai(:,c1);'])
end
```

4.11 Vcore

Vcore computes the effective vortex filament core size using the Ramasamy–Leishman model and filament stretching.

```
function wake=vcore(wake,const,fastout,user,time)
%% wake=vcore(wake,const,fastout,user,time) -> Vortex filament core size.
%
% Function computes the effective vortex filament core size using the
% Ramasamy-Leishman model and filament stretching.
%
% ****Input(s)****
% wake      Structure containing wake node positions, filament strengths,
%           vortex core radii, and vortex Reynolds number
% const     Structure containing model and atmospheric constants
% fastout   Structure containing time-dependent kinematics
% user      Structure containing user-defined variables
% time      Index for current timestep
%
% ****Output(s)****
% wake      Structure containing wake node positions, filament strengths,
%           vortex core radii (updated), and vortex Reynolds number
%
%
```

```
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited February 20, 2011
%

%% Compute vortex Re #
wake.Re.shed{time}=abs(wake.gamma.shed{time}/const.nu);
wake.Re.trail{time}=abs(wake.gamma.trail{time}/const.nu);

%% Modify coresize using Ramasamy-Leishman model
wake.rc.shed{time}=(wake.r0(time).^2+4*const.alpha*const.nu*(1+const.a1* ...
    wake.Re.shed{time}).*fastout.Time(time));
wake.rc.trail{time}=(wake.r0(time).^2+4*const.alpha*const.nu*(1+const.a1* ...
    wake.Re.trail{time}).*fastout.Time(time));

wake.rc_eff.shed{time}=wake.rc.shed{time};
wake.rc_eff.trail{time}=wake.rc.trail{time};

%% Determine filament lengths, then apply filament stretching
if strcmp(user.roll,'true')
    [vel.uind_shed,wake.length.shed{time}]=BiotSavart(wake.domain{time}(1:end-1,:,:,:), ...
   wake.domain{time}(2:end,:,:,:),wake.domain{time},wake.gamma.shed{time}, ...
   wake.rc_eff.shed{time},user.d,user.co,'length');
    [vel.uind_trail,wake.length.trail{time}]=BiotSavart(wake.domain{time}(:,:,2:end,:), ...
   wake.domain{time}(:,:,1:end-1,:),wake.domain{time},wake.gamma.trail{time}, ...
   wake.rc_eff.trail{time},user.d,user.co,'length');

    %Effective vortex filament core size due to filament stretching between
    %current time and time-1
    wake=filamentmod(wake,time);
end
```

4.12 Velocity

Velocity computes the time derivative of the positions calculated by **Kinematics**.

```
function [vel,pos]=velocity(pos,blade,turbine,wind,fastout)
%% [vel]=velocity(blade,turbine,wind,fastout) -> Turbine motion-derived and freestream
%  velocities.
%
% Function computes the velocity contributions due to turbine and platform
% motions and freestream flow in the inertial and blade coordinate systems.
%
% ****Input(s)****
% pos       Structure containing relevant positions
% blade     Structure containing blade geometry from FAST input file
% turbine   Structure containing turbine geometry from FAST input file
% wind      Structure containing imported wind data
% fastout   Structure containing imported FAST-generated results
%
% ****Output(s)****
% vel       Structure containing velocity components in inertial and blade
%           coordinate systems
% pos       Structure containing relevant positions and angles
%
%
% This work is licensed under the Creative Commons Attribution-ShareAlike
% 3.0 Unported License. To view a copy of this license, visit
% http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
% Creative Commons, 444 Castro Street, Suite 900, Mountain View,
```

```
% California, 94041, USA.
%
%
% Written by Thomas Sebastian (tommy.sebastian@gmail.com)
% Last edited June 7, 2010
%

%% Determine size of test vectors/arrays and preallocate memory
nt=length(fastout.Time); %Number of timesteps
nb=turbine.NumBl; %Number of blades
ns=length(blade.RNodes); %Number of shed nodes (stations)

%Preallocate for speed
vel.bound=zeros(ns,3,nt,nb);
vel.blade=zeros(ns,3,nt,nb);

%% Compute kinematically-derived inertial velocities using central differencing
vel.platform=ctdiff(fastout.Time,pos.platform);
vel.hub=ctdiff(fastout.Time,pos.hub);
vel.relhub=vel.platform+vel.hub+wind.infty;

vel.bound=ctdiff(fastout.Time,pos.bound);
for c1=1:nb
    for c2=1:ns %Loop over number of blades
        vel.bound(c2,:,:,c1)=-squeeze(vel.bound(c2,:,:,c1))+wind.infty';
    end
end

%% Determine velocity in BCS via coordinate transformation (inertial to blade)
pos.nodes.bxt=pos.trail-pos.quarter;
pos.nodes.bxt=pos.nodes.bxt./repmat(sqrt(sum(pos.nodes.bxt.^2,2)),[1 3 1]);
pos.nodes.bzt=diff(pos.quarter,1,1);
pos.nodes.bzt=pos.nodes.bzt./repmat(sqrt(sum(pos.nodes.bzt.^2,2)),[1 3 1]);
pos.nodes.bzt=cat(1,pos.nodes.bzt(1,:,:,:),pos.nodes.bzt);
pos.nodes.byt=cross(pos.nodes.bzt,pos.nodes.bxt,2);

pos.nodes.bxn=pos.end-pos.bound;
pos.nodes.bxn=pos.nodes.bxn./repmat(sqrt(sum(pos.nodes.bxn.^2,2)),[1 3 1]);
pos.nodes.bzn=diff(pos.bound,1,1);
pos.nodes.bzn=pos.nodes.bzn./repmat(sqrt(sum(pos.nodes.bzn.^2,2)),[1 3 1]);
pos.nodes.bzn=cat(1,pos.nodes.bzn(1,:,:,:),pos.nodes.bzn);
pos.nodes.byn=cross(pos.nodes.bzn,pos.nodes.bxn,2);

vel.blade(:,1,:,:)=pos.nodes.bxn(:,1,:,:).*vel.bound(:,1,:,:)+pos.nodes.bxn(:,2,:,:).* ...
    vel.bound(:,2,:,:)+pos.nodes.bxn(:,3,:,:).*vel.bound(:,3,:,:);
vel.blade(:,2,:,:)=pos.nodes.byn(:,1,:,:).*vel.bound(:,1,:,:)+pos.nodes.byn(:,2,:,:).* ...
   vel.bound(:,2,:,:)+pos.nodes.byn(:,3,:,:).*vel.bound(:,3,:,:);
vel.blade(:,3,:,:)=pos.nodes.bzn(:,1,:,:).*vel.bound(:,1,:,:)+pos.nodes.bzn(:,2,:,:).* ...
   vel.bound(:,2,:,:)+pos.nodes.bzn(:,3,:,:).*vel.bound(:,3,:,:);

%% Compute geometric total angle of attack (w/o induced velocity)
pos.aoag=atan2(-vel.blade(:,2,:,:),vel.blade(:,1,:,:))*180/pi; %Geometric Total AoA
pos.aoag(isnan(pos.aoag) | abs(pos.aoag)==180)=0;
```

References

[1] T. Sebastian and M. Lackner. Development of a Free Vortex Wake Model Code for Offshore Floating Wind Turbines. *Renewable Energy*, Online:1–15, 2011.

[2] T. Sebastian and M. Lackner. Unsteady Aerodynamics of Offshore Floating Wind Turbines. *Wind Energy*, Online:1–14, 2011. doi: 10.1002/we.545.

[3] Sheila E. Widnall. The Structure and Dynamics of Vortex Filaments. *Annual Review of Fluid Mechanics*, 7:141–165, 1975.

[4] Mahendra J. Bhagwat and J. Gordon Leishman. Stability, Consistency and Convergence of Time-Marching Free-Vortex Rotor Wake Algorithms. *Journal of the American Helicopter Society*, 46(1):59–71, January 2001.

[5] J. Gordon Leishman. *Principles of Helicopter Aerodynamics (Cambridge Aerospace Series)*. Cambridge University Press, 2006. ISBN 0521858607.

[6] Jack B. Kuipers. *Quaternions and Rotation Sequences*. Princeton University Press, 1998.