4-2013

BroncoBlade: An Open Source Wind Turbine Blade Analysis Tool

Alex R. Quinlan
Western Michigan University, alex.r.quinlan@gmail.com

BRONCOBLADE: AN OPEN SOURCE WIND TURBINE BLADE ANALYSIS TOOL

by

Alex R. Quinlan

A thesis submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Master of Science in Engineering (Mechanical)
Mechanical and Aeronautical Engineering
Western Michigan University
April 2013

Thesis Committee:

Peter Gustafson, Ph.D., Chair
Daniel Kujawski, Ph.D.
John Patten, Ph.D.

BRONCOBLADE: AN OPEN SOURCE WIND TURBINE BLADE ANALYSIS TOOL

Alex R. Quinlan, M.S.E.

Western Michigan University, 2013

This thesis reports the development and validation of BroncoBlade, a horizontal axis wind turbine analysis tool. BroncoBlade prepares finite element models and integrates them with an aeroelastic simulator. Analysis results for the SNL100-00 baseline blade are evaluated against reference results published by Sandia National Laboratories. Variations on the SNL100-00 blade incorporating carbon fiber are compared to the baseline blade.

ACKNOWLEDGMENTS

I would first like to thank my family, who have provided unwavering support for me and my education throughout my twenty years of schooling. Along with them, my friends have helped me keep a healthy balance between academia and the rest of the world, and to them I owe many thanks.

I would like to thank Dr. Gustafson for his guidance not only on this thesis, but for the entirety of my graduate studies. I would like to thank Dr. Patten for the opportunity to study wind energy as a research assistant and for his willingness to be on my thesis committee. I would also like to thank Dr. Kujawski for his membership on the committee and for his instruction in the classroom.

Alex R. Quinlan

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As environmental and economic factors urge reform in the energy sector, wind energy has emerged as one of the primary alternatives to fossil fuels. Historically, wind energy has been used for centuries, but its recent development as a utility scale electricity provider can be attributed to improvements in design and available materials. As the wind turbines advance, they are trending towards larger blades, allowing each turbine to produce more power.

In 2011, Griffith and Ashwill of Sandia National Labs released a report on the SNL100-00, a 100m fiberglass turbine blade design [7]. At the time of publication, the largest wind turbine blades in production were about 60m, so Sandia's design is meant as a baseline for development of future blades. The original goal for this thesis was to explore design improvements to the baseline turbine, mainly by modifying the composite materials in the design. During the modeling of the SNL100-00, it was observed that the *easiest* path to creating an accurate model involved several closed source software packages, which required licenses that were not available at Western Michigan University. From this attempted design path, the need for an open-source blade meshing program was discovered.

From that point on, the thesis had two goals. The first was to write a pro-

gram that could model a wind turbine blade using minimal proprietary software packages. The second goal was to use the program to create a duplicate model of the SNL100-00, and make subsequent design iterations based on that model. This developed program was entitled *BroncoBlade*, after the Western Michigan University mascot.

1.1 Wind Turbine Terminology

For the purpose of clarity, the main components of a horizontal axis wind turbine (HAWT) and the accompanying terminology will be briefly reviewed.

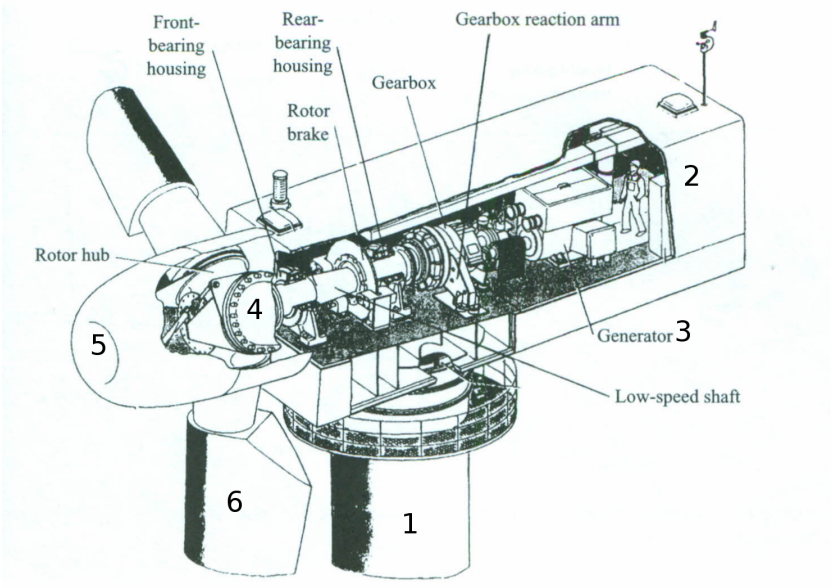


Figure 1.1: *Labeled components of a wind turbine [2]*

The components labeled in Figure 1.1 are described here:

1. *Tower*

   The tower serves to be a stable structure holding the turbine and anchoring it to the earth (or to an offshore platform). This allows for free rotation of the blades without contacting the ground and for access to higher wind speeds, the turbine must be placed well above the ground.

2. *Nacelle*

   The nacelle houses the generator, gearbox, and other electromechanical systems.

3. *Generator*

   The generator turns the mechanical energy of the rotor into electrical energy.

4. *Rotor*

   The rotor is composed of the hub and of the blades.

5. *Hub*

   The hub connects to the generator shaft and acts as the center of rotation for the rotor. The blades are bolted onto the hub

6. *Blade*

   The blades use aerodynamic principals to extract energy from the wind.

The components in Figure 1.2 are labeled here. Descriptions of these items appear throughout the paper.

1. *Skin*
2. *Shear Web*
3. *Chord*
4. *Thickness*
5. *Leading Edge*
6. *Trailing Edge*

1.2 Coordinate Systems

To provide proper orientation, the global and local coordinate systems are described here. Because the design focuses on the blade, the coordinate systems are based on the blade geometry.

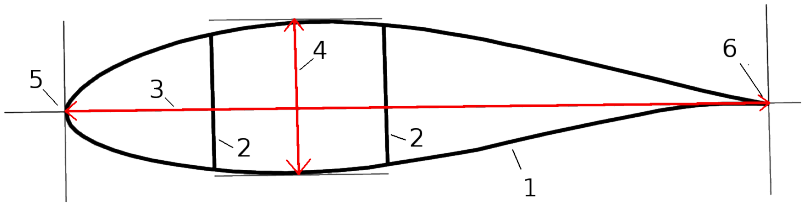


Figure 1.2: *Terminology describing the blade's cross section*

The local coordinates are confined to a 2D plane for a given cross section of the blade. The x-axis lies on the chord of the blade, connecting the leading edge to the trailing edge. This is referred to as the *edgewise* direction. The y-axis lies perpendicular to the x-axis on the cross section plane, pointed towards the low pressure side of the airfoil. Movement on the y-axis is referred to as *flapwise*. The general coordinates on the xy plane are called *chordwise*. The origin for the local coordinate system lies on the pitch axis, or the global z-axis, which is described below. In this paper, there is no sweep to the blade, so the local z-axis is always the same as the global z-axis.

Turbine blades typically have a circular cross section at the root where they are bolted into the rotor hub. The origin for the global coordinates is located at the center of this circle. The z-direction, also referred to as the *spanwise* direction, extends normal to this root connection. The z-axis is also referred to as the pitch axis because the blade rotates about it during pitching maneuvers. Figure 1.3 illustrates the directions used to describe the blade.

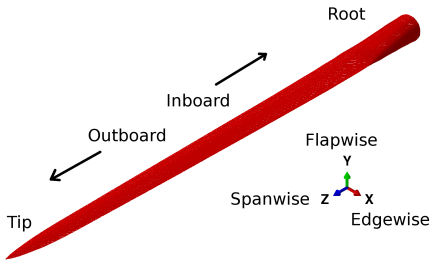


Figure 1.3: *Global coordinate system*

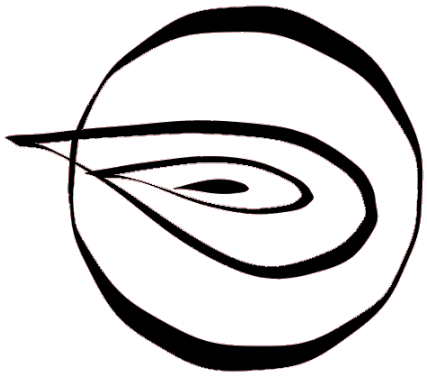


Figure 1.4: *Twist of the blade shown by stations along the global z-axis*

Because each section of the blades has a unique angle of twist as shown in Figure 1.4, the angle between the global x-direction and local x-direction will vary based on the angle of twist at that location. Following the convention used by Sandia in the description of the SNL100-00, the cross section at the tip will have a twist of 0°, meaning the global x and y directions are parallel with the local coordinates at the tip. The degree of twist then increases as the cross section moves closer to the root.

1.3 Overall Turbine Design

When beginning the design of any component, it is critical to define the goals and boundaries that are required. For a horizontal-axis wind turbine (HAWT) blade, the design parameters are outlined as so [2]:

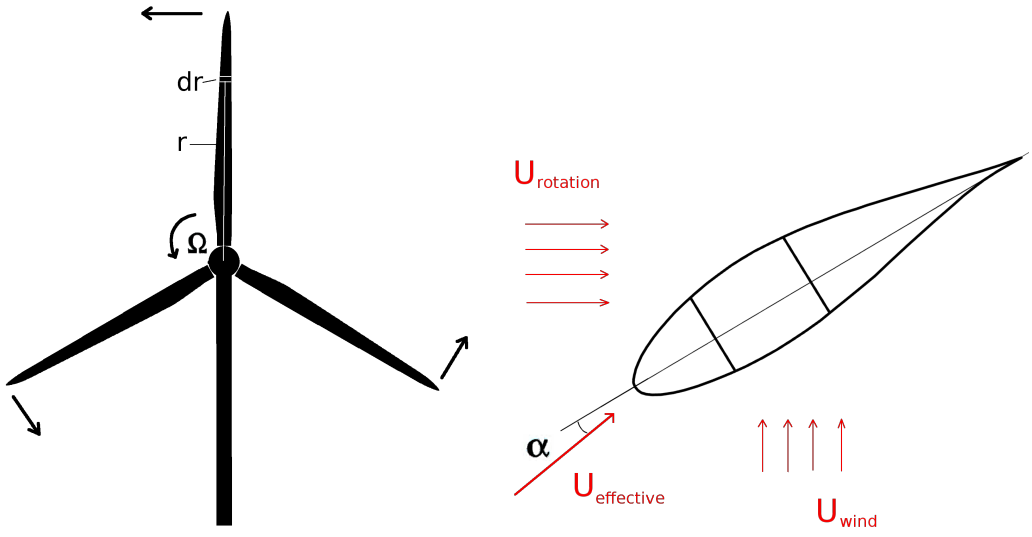
1. Maximize annual energy yield for specific wind speed distributions
2. Limit maximum power output
3. Resist extreme and fatigue loads
4. Restrict tip deflections to avoid tower collision
5. Avoid resonances
6. Minimize weight and cost

The first two parameters are mainly aerodynamic design problems, while 3-6 incorporate structural design. Methods for addressing these two types of problems differ, though a full design will need to use both to fully simulate the behavior of the turbine.

1.3.1 Aerodynamic Design

A Horizontal-axis wind turbine operates on the principle of extracting energy from the passing air by means of lift generated by the blades. An ideal aero-design of a blade will try to maximize the efficiency of the turbine, getting as close to the theoretical maximum limit (Betz limit is 59.3%) as possible.

As the wind blows across the blades, a lift force is generated, causing the turbine rotor to rotate. Once rotation begins, the blade is subjected to a combined airflow that is composed of both the blowing wind and the relative movement of the blade through the air. Thus, the wind direction *seen* by the blade is not the same as the global wind direction, and is dependent on the rotational speed of the blade. Figure 1.5 illustrates the relative wind phenomenon.



(a) Blade rotation (b) Relative wind speed

Figure 1.5: *Effects of relative wind speed*

Because the global wind velocity is not constant, a turbine cannot be designed for a single wind speed. Blades are designed with an operating speed in mind, which is the speed at which the turbine will be most efficient. The best choice for operating speed will be a function of the local wind distribution, which is always studied before a wind farm is built or a turbine is placed.

To predict the behavior of a blade during operation, the lift and drag coefficients are needed as functions of the angle of attack for the different airfoils used in the blade. These are used to approximate the pressure distribution on the blade and thus its movement. Programs, such as FAST [11], use this to simulate the aerodynamic loads and structural deflection during operation.

1.3.2 Structural Design

For an aerodynamic design to be functional, there must be sufficient support to keep the aerodynamic profile from deforming. Like in other aerospace applications, the cost and effectiveness of the turbine is generally negatively affected by increased weight, so excess material is not desirable. To optimize the design, a structural analysis of the blade must be performed. To accurately examine the stresses and strains of the blade, the finite element method should be used.

The blade's skin serves to both define the aerodynamic surfaces and to provide structural support. The outer surface of the skin has its design governed by the aerodynamic requirements, but can have its internal thickness and composition designed for structural performance. Typically, an outer layer of fiber-reinforced polymers will be used for the entire blade. Then different sections will be reinforced with more composites or foam material as is deemed necessary.

The skin has limited ability to resist shear loads because of it's hollow structure, so thin structures called shear webs are added internally [2]. As these connect the top and bottom surfaces, they also form a torque box which aids in resisting

torsional loads. Shown in Figure 1.6, shear webs also reduce the active buckling area of the skin, increasing the buckling stability in that section.

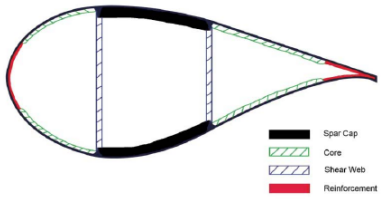


Figure 1.6: *Blade cross section of SNL100-00 [7]*

1.4 HAWT Blade Loading

Throughout its lifetime, a wind turbine will undergo many different loadings based on the different wind conditions. The structure of the wind turbine must be able to withstand these loadings to prevent failure and to ensure the proper operation of the turbine. In the most basic terms, a blade is a cantilever beam rotating about a fixed axis. To understand the requirements of the structure, one must understand the four different sources of loading that are present in turbine's operational lifetime.

1.4.1 Aerodynamic

The aerodynamic loads come from the relative movement of air over the turbine blade. The blade has a cross section in the shape of an airfoil, so lift and drag forces are generated as it passes through the air, using the same principal that allows wings on an airplane to fly. The way these forces act on the blade are shown in Figure 1.7.

The aerodynamic design of the blade ensures that the lift force will cause the blade to rotate about its rotor axis. The lift and drag forces can be calculated using Equations 1.1 and 1.2, where the section lift and drag coefficients (C_L and

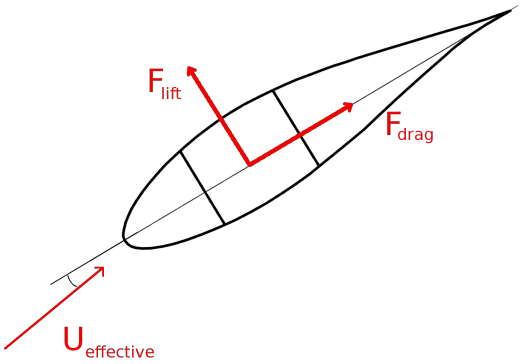


Figure 1.7: *Free body diagram of aerodynamic forces*

C_D, respectively) are determined by the shape of the airfoil, and the area (A) is determined by the chord length.

$$F_{Lift} = \frac{1}{2} C_L \rho A v^2 \tag{1.1}$$

$$F_{Drag} = \frac{1}{2} C_D \rho A v^2 \tag{1.2}$$

As the blade begins to rotate, the relative speed between the air and the blade (v) increases, since the global wind speed becomes augmented with the relative velocity of the blade moving through the air. Because the blade is rotating about a fixed axis, the principals of rotational motion apply, which require that the blade have a constant rotational velocity throughout the structure and the linear speed of the blade increase from the root to the tip. This means that at any given moment during operation, the speed v will be highest at the tip and decrease linearly as the location moves towards the root. Also, since the velocity component coming from rotation changes as a function of location, its ratio to the global wind speed changes too, meaning that the effective direction of the wind experienced by the blade changes with its spanwise position. Because of this fact, blades incorporate twist to allow the ideal angle of attack for the maximum amount of the blade.

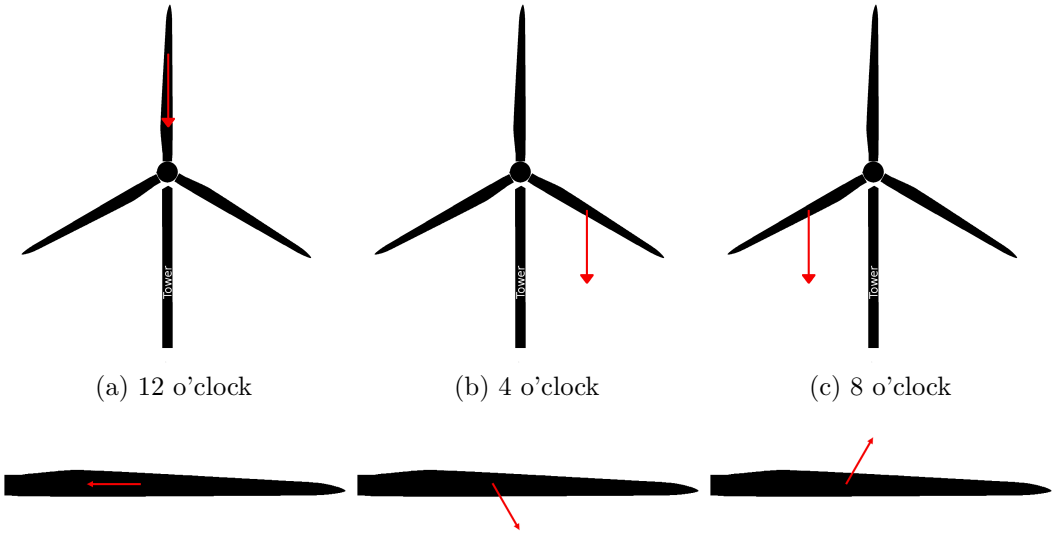
When the lift force, drag force, and rotational moment are applied, the structural reaction results in the blade bending and twisting. Consequentially, the aerodynamic forces change with the new blade configuration. Because of their interdependence and the variability of the global wind speed and direction, the aerodynamic forces cannot be calculated by hand. Fortunately, there are simulation programs that can handle the many calculations that are required to give a prediction of the turbine's performance.

1.4.2 Gravitational

As the turbine rotates, gravity is always present, applying a load on all of the turbine down toward the earth. From the global turbine perspective this load is constant, but since the blades are spinning, their orientation is always changing in relation to gravity. This means that from the local turbine perspective, the gravitational loads are applied to the blade in a cyclically varying direction. Because of the pitching of the blade, the gravitational loads are mainly in the edgewise direction. If the position of the turbine is represented by the hands on a clock as pictured in Figure 1.8, then the positions corresponds with the following example gravitational loadings:

- 12 o'clock = axial compression
- 4 o'clock = axial tension + leading edge compression / trailing edge tension
- 8 o'clock = axial tension + leading edge tension / trail edge compression

The gravitational load, or the blade's weight, is only affected by the mass of the blade, since gravity is assumed constant. However, the moment caused by the gravitational load at the root will increase as the blade length increases, since the moment arm is extending. For small turbines, gravitational loads are usually small in comparison to the aerodynamic loads. But, in the case of large turbines,



(a) 12 o'clock (b) 4 o'clock (c) 8 o'clock

Figure 1.8: *Gravitational load's cyclic nature in the blade coordinate system*

such as the one modeled in this paper, the gravitational loads become increasingly important.

1.4.3 Inertial

As the blade spins, centripetal forces act on the blade in the radial direction. If the blade were perfectly rigid and did not undergo bending from the other load types, these centripetal forces would act in a purely axial fashion, causing tension throughout the blade and effectively stiffening it. When the blade does undergo bending, the stiffening from the centripetal forces acts in a manner that reduces the bending curvature of the blade.

1.4.4 Operational

Operational loads are loads that occur during specific instances of the turbine's life, opposed to the aerodynamic, gravitational, and inertial loads, which are always in effect when a turbine is running. Examples of operational loads include those experienced during braking, pitching (rotation of the blade about the pitch

axis), and yawing (rotation of the nacelle). The turbine also experiences additional loads during start up and shutdown, and in extreme conditions like the failure of the generator. While these loads are very important for a complete turbine design, they are neglected in this paper because the aerodynamic simulation does not include any special operations (although pitching is included in a few cases).

1.4.5 Wind Cases

The IEC (International Electrotechnical Commission) designates different wind conditions that can be experienced by a wind turbine. Each of these has a different code, and its exact wind values depend on the wind classification at the location of the turbine. For example, the harshest wind classification is Class IB [2]. The *Class I* means that the annual average wind speed is 10 m/s, which is the highest of all of the classes. Consequentially, the reference wind speed and 50-year gust speed are also the highest in Class I, at 50 m/s and 70 m/s respectively. The *B* in Class IB signifies a higher level of turbulence than a class A location. A Class IB wind condition is used in this paper, since a design that can survive this class will survive all of the other classes. The load cases mentioned used in this paper are the EWM50 extreme wind 50 year gust condition and the ECD-R extreme coherent gust with direction change load condition. While the ECD-R condition occurs during operation, the EWM50 condition is applied when the high wind speeds have forced the turbine to be shut down and the blades have stopped rotating.

1.5 Analyses for Turbine Blades

To comply with the design parameters outlined in the previous section, a blade design must be subjected to a number of simulations to prove its quality. The three structural simulations that are performed on blades are a static maximum load simulation, a buckling simulation, and a fatigue simulation. The buckling and static simulations are options in BroncoBlade v0.1. Improvements to the static simulation and the creation of the fatigue simulations are planned for addition and discussed in Chapter 6.

By nature of the wind turbine, blades will be exposed to the elements, including unpredictable extreme weather. Though designs do not plan for the destructive forces of hurricanes or tornadoes, they do account for the rarest of heavy winds that are known in the area. The maximum wind condition usually taken into design is called the 50-year gust, meaning the maximum wind speed in a 50 year period. This value is estimated on wind distribution data, rather than recording the actual maximum during a 50 year wind speed study.

Additionally, the position of the turbine will change how damaging the extreme wind is on the blade. Though many small and mid-sized turbine designs use a stall-regulated blade, the largest turbines use a pitching system to control the rotational speed. During extreme weather, turbines shut down to minimize the damage from the extreme wind. Pitch regulated machines will *pitch to feather* during extreme weather, which reduces the angle of attack (minimizes drag). However, Sandia ran the extreme wind speed simulation under the worst case scenario, where the blades could not be pitched out of the wind and were stuck in the maximum drag position [7].

The aerodynamic loading that occurs from this extreme situation is then applied to a static FEA model of the turbine to find the stresses that occur. The stress values must comply with safety factors, such as those set forth by

Germanschier-Lloyd [14].

When the relatively thin skin of the blade is subjected to compressive loads, there is a possibility of localized buckling. This buckling can remain in the elastic region and contribute to fatigue damage, or move into the plastic region and cause localized skin failure. Because of the high cycle lifetime of the blade, the structural design must take into consideration all types of buckling.

To evaluate how a design will perform in terms of buckling, the load conditions are applied and a specific buckling FE analysis is performed. The results of this analysis give the buckling modes and an accompanying eigenvalue. An eigenvalue of 1 or below implies that the critical load for buckling has been exceeded. The eigenvalues greater than 1 can be interpreted as the safety factor on the applied loads. Some eigenvalues are negative, which means the direction of loading would need to be reversed to cause buckling. These negative values are not used for the buckling analysis.

1.6 Current Designs Issues

A HAWT blade is composed of an exterior shell and interior structure. For large scale turbines, the shell is a laminate of either fiberglass or carbon fiber paired with foam or a light wood (balsa). The main two interior support designs used are a 1 piece (being a single box spar) and a 2 piece (a pair of shear webs). The important difference is that loads are transfered through the box spar directly, and less load is transfered to the skin. However, since the pair of shear webs are unconnected, the load between the two is transfered via the skin and adhesive. The consequence of this is that a stronger adhesive must be used for the 2 piece support system.

Both bend-twist and stretch-twist coupling are utilized in wind turbine design.

Stretch-twist relies on centrifugal forces and twists the blade, thus changing the pitch and aero-characteristics [15]. Bend-twist is animated by the thrust force of the wind. Elastic coupling mechanisms can be embedded in the skin or in the box spar.

This coupling increases the level of complexity of the interaction between the aerodynamic performance and the structural response. If properly incorporated into the design, this coupling phenomenon could be advantageous for the performance of the turbine.

Coupling is currently not included in FAST, and so it is not actively being developed in BroncoBlade.

1.7 Larger Blades

The central goal in wind turbine design is an economic one: to minimize the cost of energy produced, which is dependent on the ratio of the power output of the turbine and it's production, installation, and operating costs.

1.7.1 Examining the Power Equation

Compared to the complicated algorithms that go into cost estimation, the turbine's general power output (P) can be summarized in one simple equation [2].

$$P = \frac{1}{2} C_p \rho A U^3 \tag{1.3}$$

The first variable in the equation, C_p, is known as the power coefficient. It has a theoretical maximum of .593, known as the Betz limit, but in practice has a lower value. Turbines are designed with a *rated wind speed*, and the C_p changes depending on how close the wind speed is to the rated speed. Next is ρ, which is the density of the air passing through the wind turbine. This has little

fluctuation compared to the other parameters, ranging from $1.225kg/m^3$ at sea level and reducing to slightly less than $1kg/m^3$ at high altitudes. For a general overview, it is most important to notice that power output is directly related to C_p and ρ, meaning that they are raised to the first power. This is different from the last two variables, which have high order correlations.

Though power output is directly related to the swept rotor area, the area is determined by πr^2. Thus, the power output is related to the square of the blade length. This makes larger blades an attractive option, since a turbine with 20m blades will have the same swept area as 4 turbines with 10m blade.

The final variable, U, is the speed of the wind as it heads towards the wind turbine (the turbine generates electrical energy by extracting kinetic energy from the wind, which means that the speed of the wind after it passes through the rotor is much less than it is before). Because of the cubic relationship between U and P, wind speed is the most important factor in the power production of a turbine. The wind speed is dependent on two things: the geographic location, and the height above the ground (or sea).

Though wind speed is the most important factor, it cannot be engineered by man on a large scale. Thus, engineers are focused on increasing the size of the turbine blade to maximize the swept rotor area. Of course, like all engineering problems, there are limits and trade offs.

$$P \propto r^2 \tag{1.4}$$

$$m \propto r^3 \tag{1.5}$$

As mentioned above, the swept rotor area and turbine power output are related to the square of the blade length. However, as the blade increases in length it must also increase in its cross sectional dimensions to prevent it from failing under

the increased bending moments caused by added material. In general, it is said that the volume, and thus the mass, of the blade will increase with the cube of the blade length. An increase in mass means an increase in material, and thus an increase in cost of the blade, and an increase in cost in other aspects of the turbine [2].

The relations between the rotor radius, r, mass, m, and power, P, in Equations 1.4 and 1.5 show that to keep an optimal power to cost ratio, the design of the blade should adapt as the blade becomes longer.

1.7.2 A Case for Carbon Fiber

Because the weight of the blade becomes a more significant factor in design as the blade length increases, the use of a stronger and lighter material becomes more and more advantageous. The terms often used to quantify the relationship between strength/stiffness and density of a material are called *specific strength* and *specific stiffness*.

Carbon fiber reinforced polymers (CFRP) are known for their high specific strength and stiffness, and are being explored as an alternative to the current fiberglass design. As can be seen in Table 1.1, the AS4 carbon fiber laminate doubles the specific strength of E-glass and more than triples the specific modulus. Using carbon fiber would reduce the amount of material required in the structure of the blade to handle the loads, which would in turn reduce the mass of the blade and reduce the gravitational loads.

While carbon fiber offers structural advantages, its high cost has limited its use in large scale applications; it generally costs about 10 times more than fiberglass. However, recent advances, like the use of a CFRP fuselage in the Boeing 787 Dreamliner [9], suggest that the economic benefits of a carbon fiber design may be starting to outweigh the raw material cost.

Table 1.1: Comparison of Density and Engineering Material Properties [3] [5]

Form	Material	Density ($\frac{kg}{m^3}$)	Elastic Modulus (GPa)	Tensile Strength (MPa)	Specific Modulus ($\frac{MNm}{kg}$)	Specific TS ($\frac{KNm}{kg}$)
Fibers	E-Glass	2540	73	3450	28.7	1350
	S-Glass	2490	86	4500	34.5	1800
	AS4 Carbon	1810	235	3799	129.8	2100
	IM-7 Carbon	1800	290	5170	161.1	2870
Uni-axial laminates (V_f)	E-Glass (0.55)	1970	41	1140	20.8	578
	S-Glass (0.50)	2000	45	1725	22.5	862
	AS4 Carbon (0.63)	1600	147	2280	91.9	1420
Isotropic Engineering Materials	A36 steel	7850	207	500	26.3	63
	440A Steel	7800	200	1790	25.6	229
	Ti-6Al-4V Titanium	4430	114	1172	25.7	264
	7075 Aluminum	2800	71	573	25.3	204

1.8 Outline of the SNL100-00 Baseline Blade

With the rotor size of turbine designs increasing annually, Sandia National Laboratory's research for the future of HAWTs goes past the state of the art 60m blades, and looks to a larger blade of 100m. It is expected that the challenges associated with this larger blade will push designs to be more aerodynamically, structurally, and economically efficient. Sandia's first model is the purposely a basic design, using fiberglass as its primary material. This model, the SNL100-00, is to be the starting point for future studies (such as this thesis) that look to reduce weight and increase performance.

The first step in developing the SNL100-00 was upscaling two 5MW turbine designs to 13.2 MW designs. There was no available composite laminate data from either of the 5MW blades, so a laminate schedule was developed to approximate the stiffnesses predicted by scaling. The upscaled models were then simulated in FAST using extreme loading conditions. The results of the simulation showed that a good 13.2 MW design required more edgewise reinforcement than the directly scaled models provided. The next design iteration added reinforcement, but was shown to be inadequate in buckling in the aft panel. This problem was addressed

in the final design by adding a third shear web.

Both the aerodynamic and structural properties of the Sandia 100m Baseline Blade are outlined in a Sandia report [7]. There was enough information in the report to complete the input files that would eventually be used in BroncoBlade.

1.9 Review of Available Modeling Tools

With the goal of exploring the structural design of a blade, a finite element model was required. A few approaches were used in trying to create a 3D blade model and mesh.

1.9.1 NuMAD

One tool specifically designed for creating a finite element model of a wind turbine blade is Sandia National Laboratories' NuMAD [13]. This program generates a mesh based on airfoil station inputs defining the blade's exterior shape, and on defined shear webs inside the skin, shown in Figure 1.9. NuMAD links with the FEA solver ANSYS to compute the properties of the blade, and requires an ANSYS license for the FE mesh to be exported. Unfortunately, ANSYS educational licenses were not compatible with NuMAD at the time of this thesis, which limited the program's usefulness in a university setting without the full ANSYS license.

Although the program was available for free upon request from Sandia, the source code was not available, and was only actively developed for the Windows platform. Since Linux was the preferred platform for this project, it was decided to forgo the use of NuMAD. However, the creation of BroncoBlade attempted to incorporate the attractive aspects of NuMAD and provide open source code that can be altered for the user's needs.

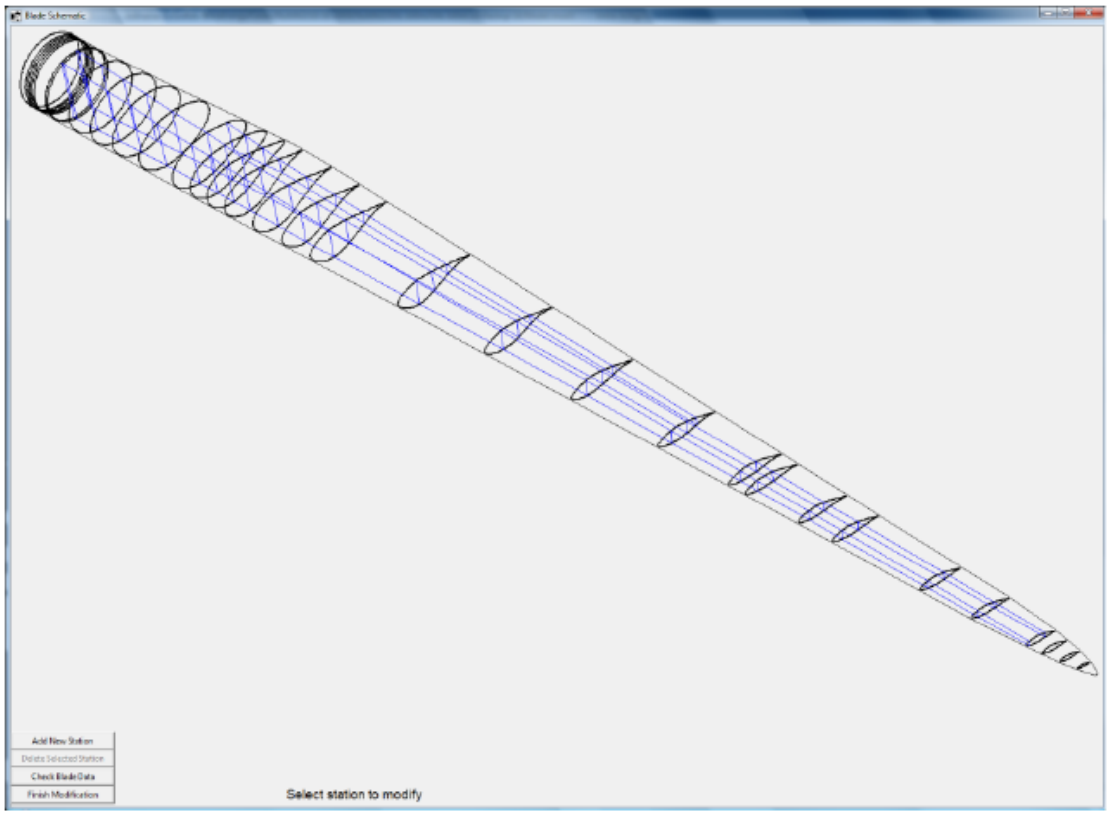


Figure 1.9: *User interface of the NuMAD program [13]*

1.9.2 SolidWorks/Inventor

Once the use of NuMAD had been removed from consideration, an attempt was made to create a solid model of the blade using general commercial 3D CAD packages, SolidWorks and Inventor. The general idea was to imitate the stations of NuMAD by creating a 2D spline for each airfoil and placing them as cross sections along the blade. Once in place, the airfoils could then be lofted into a shell.

However, all aspects of this approach proved cumbersome. Importing the xy coordinates defining the airfoil required the use of Microsoft Excel file for one 3D modeling package, and didn't seem to work at all for the second (possibly an issue with the teaching license). Once the points were import from the Excel file, splines had to be generated and manually altered. At this point, the 3D graphics

display started to bog down. The display problem was exacerbated when the loft was created from the stations, and the response of the user interface to commands became choppy and mostly unusable. Once a loft was finally exported from the 3D modeler and re-opened in the meshing program Hypermesh, the complex shape of the blade had been simplified down to a single airfoil cross section and a line depicting the trailing edge. Despite trying multiple file formats, none could be exported from the modeling package, and imported by the meshing program, and still accurately represent the geometry.

Though these programs are available in most universities and work places, they are closed source, have a fee for use, and force the user to operate within a graphical interface. Rather than continuing to troubleshoot the solid modeling programs, this approach was abandoned in favor of creating BroncoBlade.

1.9.3 FAST

With the overall goal being the analysis of a blade under appropriate loading, generating a blade model is only half of the process. For a realistic simulation, there must also be realistic loading of the blade. This is where FAST comes in. FAST (Fatigue, Aerodynamics, Structures, and Turbulence) is simulation program created by the National Renewable Energy Laboratory (NREL) [11]. It couples with another NREL program, AeroDyn [16], to simulate the Aerodynamic loads and structural response of a wind turbine in the field. NREL makes both of these simulators available on their website, including the source code.

BroncoBlade is written to provide FAST/AeroDyn with the required information for accurate simulation, and to retrieve the results so they can be applied to the finite element model. Because FAST takes very detailed input, some assumptions must be made when there is no specific value given by the Sandia Baseline report (e.g. Tower Properties).

FAST has the capability of preparing an ADAMS model, and interfacing with Simulink. Since this project is leaning away from commercial packages, FAST's ability to link with these programs will initially go un-utilized.

1.10 Objectives

The two objectives for this thesis are to explore the use of carbon fiber in the structure of the wind turbine, and to create a program to aid in this exploration. The idea for the thesis began with wanting to see how composite laminate material and layups can be redesigned to improve the performance of the SNL100-00 blade. In pursuing this goal, it was quickly seen that the available resources fell short of what was needed to complete the task. As described in sections 1.9.1 and 1.9.2, existing tools required expensive licenses unavailable at WMU or were inefficient in creating the model.

The design process failures that occurred using these avenues of modeling showed the need for an open-source design code, powered by and linked with other open-source packages. Having the code available for free and open for modification would be valuable to people outside of private industry who have limited or no access to expensive commercial engineering software.

Though the exploration of materials in the blade structure was the original goal for the project, this objective was pushed back into a secondary roll as the creation of BroncoBlade became the main focus of the project.

1.11 Deliverables

The items delivered by this thesis include:

1. *Release of BroncoBlade v0.1*

This is the alpha release of BroncoBlade. The code has been tested by the author and has been proven to be functional, but it has not yet had any other user. Feedback from this release will help to make BroncoBlade a more user friendly program.

2. *BroncoBlade user manual*

   This user manual is presented here as Chapter 2 of this document. It explains the how BroncoBlade functions, but does not dive into line by line explanations, as the source code has commentary throughout.

3. *Results validating BroncoBlade*

   To prove itself as a useful engineering tool, the results of BroncoBlade are validated against Sandia's results for the SNL100-00 turbine in Chapter 4.

4. *Re-design iterations of SNL100-00*

   Models based on the SNL100-00 are created with improvements to reduce weight and material in Chapter 5.

Chapter 2

BroncoBlade

2.1 Introduction to BroncoBlade v0.1

BroncoBlade is a software package capable of pre-processing both aerodynamic and structural simulations of a wind turbine blade and linking results between the two. It is inspired directly by Sandia National Laboratory's NuMAD [13] and by NSE Composite's BladeMesher [10]. Though these programs have similar capabilities, BroncoBlade is unique in that it is open source. It is written in the object oriented programming language Python [6]. With the source code available to the user, BroncoBlade can be customized to work with the user's preferred analysis tools; for example, though BroncoBlade is currently configured to read and write ABAQUS file formats, it can be freely modified to integrate with ANSYS or any other finite element program. Additionally, BroncoBlade's four modules can be expanded upon or augmented with other modules. The four modules contained in the initial release of BroncoBlade are:

1. The *Blade* module, which builds the mesh for the full turbine blade
2. The *Sections* module, which calculates stiffnesses and densities at each station

3. The *Modes* module, which calculates the mode shapes and prepares the input files for the aerodynamic simulation
4. The *Loads* module, which reads in the loads generated by FAST and writes out FE input files for static and buckling analysis

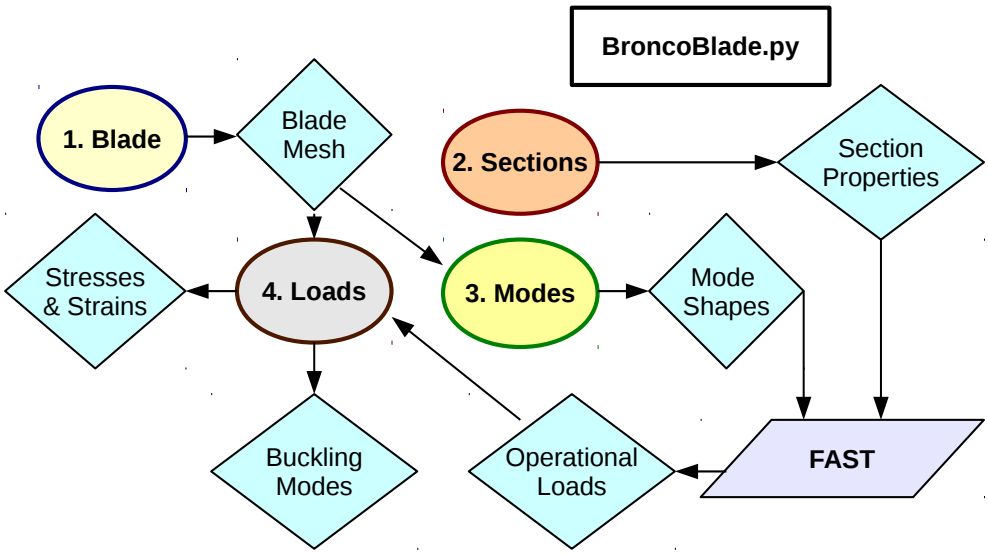


Figure 2.1: *Overview of the four modules in the BroncoBlade program*

Each module can be run individually or multiple can be run together. The *Blade, Sections*, and *Modes* modules are used as pre-processors for the aerodynamic simulation. The *Loads* module is used to post-process the aerodynamic simulation, and is then used in conjunction with the *Blade* module to pre-process the structural analysis. The basic interactions and dependencies between these modules are shown in Figure 2.1.

2.2 Required Inputs

BroncoBlade runs based on three types of input. The first is the arguments given from the command line. Second is the text and data files that are opened and

read by BroncoBlade. The third is a python script that contains the materials properties in a way so they can be written to the FE input file in the proper format. Examples of each of these input files is shown in the Appendix.

2.2.1 Command Line Arguments

The first input read when running BroncoBlade is the series of arguments given from the command line. Each module is activated by supplying an argument in the form of a dash followed by the first letter of the module. Sub-arguments follow without any blank space in between. The arguments available for each module are:

1. -B = Blade module
   s = save station attributes
2. -S = Sections module
   l = load station attributes
   r = run FE analysis
   p = post-process FE analysis
   s = save station attributes
3. -M = Modes module
   l = load station attributes
   r = run FE analysis
   p = post-process FE analysis
4. -L = Loads module

When pre-processing for FAST, the general order of operation is to run the Blade, Sections, and Modes modules in that order. If these three modules are all selected, there is no need to save and load station data between the modules. The command line for this would look like:

- BroncoBlade.py -B -Srp -Mrp *input.in*

The modules can also be run in separate callings of BroncoBlade, which allows the user to inspect results between modules.

- BroncoBlade.py -Bs *input.in*
- BroncoBlade.py -Slrps *input.in*
- BroncoBlade.py -Mlrp *input.in*

The Loads module is slightly different in that it has no sub-arguments, but must be followed by the name of the results file produced by FAST containing the loads chosen for analysis. Since the Loads module is only usable after this results file has been generated by FAST, it typically will be used in a separate calling of BroncoBlade from the other modules. An example of calling the Loads module looks like this:

- BroncoBlade.py -L *WindCondition.elm input.in*

When calling BroncoBlade, the last item in the command line must always be the main input file.

2.2.2 Main Input

As shown in the examples of command line entries in the previous section, the final argument of the command line is the name of the main input file. This file has the extension *.in* and contains both model data and the location of the other required data files. The file is structured with each line containing a keyword followed by the value for the keyword with white space in between. The order of the keywords is not important, as long as they are all contained in the file. Table 2.1 lists the keywords with a brief description.

Table 2.1: Description of Keywords in the Main Input File

Keyword	Description
dir	directory containing airfoil definition files
sched	file listing the airfoil schedule
length	length of the blade from root to tip (m)
seeds	chordwise seeding parameters for sections A,B,C,D,E
spseed	spanwise seeding parameter for sections between stations
num_aerosections	number of aerodynamic sections used in FAST (needed only for Loads module)
Asize	chordwise length of leading edge reinforcement (m)
Esize	chordwise length of trailing edge reinforcement (m)
spar1	chordwise location of Spar 1 relative to the pitch axis (m)
spar2	chordwise location of Spar 2 relative to the pitch axis (m)
SPstrt	starting spanwise location for Spars 1 and 2 (m)
SPend	ending spanwise location for Spars 1 and 2 (m)

2.2.3 Airfoil Definitions

Since BroncoBlade does not have a built-in library of airfoil shapes, the user must provide airfoil definition files. The files must consist of a column of x coordinates and a column of y coordinates. The first coordinate listed should be either (0,0) or (1,0), so the points start at either the leading or trailing edge. Subsequent coordinates should follow in a continuous fashion; there should not be any back-tracking. The color change in Figure 2.2 shows an airfoil defined with the start at the trailing edge.

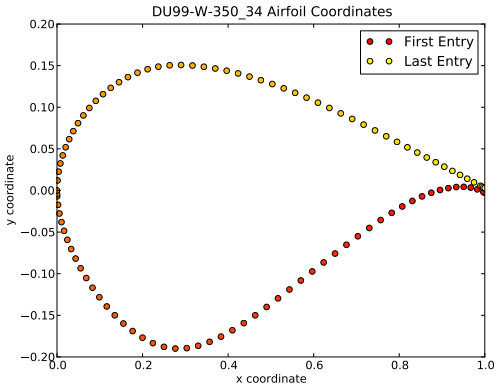


Figure 2.2: *Coordinates for a defined airfoil*

2.2.4 Airfoil Schedule

The airfoil schedule describes the shape and position of each of the stations. It has a file extension of *.bld*. Columns containing the data appear the following order:

1. Station number
2. Fraction of spanwise position
3. Airfoil type
4. Chord (m)
5. Twist (degrees)
6. Pitch axis fraction
7. Thickness-to-chord ratio

An example of an airfoil schedule file format is shown in the Appendix, and the data is shown later on in Table 4.1 on page 53.

2.2.5 Laminate Schedule

The laminate schedule is defined in a file called *layup.txt*. This file tells BroncoBlade where to put reinforcement in the skin, what material to use, and how thick is should be. Currently, this file follows the format of the laminate schedule published for the SNL100-00 [7], which is shown later on in Table 4.5 on page 58.

2.2.6 Material Properties

The python file *write-mats.py* is contained in the source code directory, but should be copied into the working directory and customized for each blade. It is configured so that it will write out all of the material properties in the proper ABAQUS input file format. Because of the high variability in material definitions, such as modeling as isotropic, orthotropic, or fully anisotropic, the material properties are

left in this script rather than being formatted in a general text file. The future work chapter includes a suggested modification to this input: coding a reader that will allow a text file to be used to define the material properties.

2.3 Model Creation: The *Blade* Module

The *Blade* module (Figure 2.3) is core of the BroncoBlade program. It creates the stations that define the blade geometry, meshes the blade, assigns the laminate sections, and then prints out the almost the entire finite element model. The printed FE input file lacks loads and boundary conditions, allowing it to be referenced by multiple jobs with different loadings throughout the BroncoBlade process.

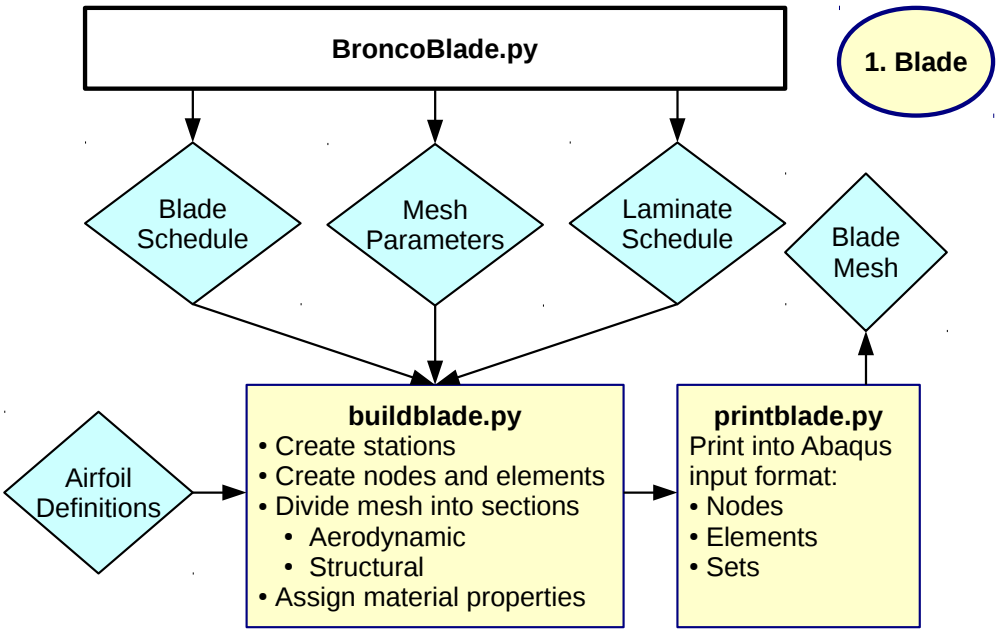


Figure 2.3: *Flow of programs and data in the Blade module*

2.3.1 Meshing

The first product of BroncoBlade is a finite element mesh containing both the skin and the shear webs of the blade. BroncoBlade uses *stations*, which are essentially airfoil cross sections that can be lofted together to define the exterior surface of the blade (see Figure 2.4). On each of the stations, the chordwise seeding is applied, giving each station the same number of nodes (see Figure 2.5). These nodes are created so that all of the nodes with the label "node 1" on each station can be used to generate a spline in the spanwise direction. BroncoBlade generates these *spanwise splines* using a curve fit, giving the blade a general wire-frame form (see Figure 2.6). Since the blade is fully divided in the chordwise direction, the next step is to further divide the blade in the spanwise direction, since it currently only has spanwise divisions at the stations. In the case of the SNL100-00 blade, there are 34 stations, and thus 33 sections between them. So, the spanwise seeding control is a vector of length 33, telling BroncoBlade how to subdivide each of the 33 sections. At each of the subdivision points, a node is created. As an object in the python language, each node is assigned a number, (x,y,z) coordinates, and (chordwise,spanwise) coordinates. Once all of the nodes on the skin have been created, then the elements can be created. The default elements in BroncoBlade are second order 8-noded shell elements, which has 4 corner nodes and 4 side nodes. Because not every node is a corner, it becomes very important to create the nodes such that the corners of the element coincide with the geometric boundaries. This can be done by maintaining the odd or even characteristic of the different parameters in the input file. Avoiding this complication is a part of the future work and is discussed in section 6.3.1. Also, since there are 8 nodes associated with each element, the 9th or center node is ignored and left unattached. The completed skin mesh of a portion of the blade is shown in Figure 2.7.

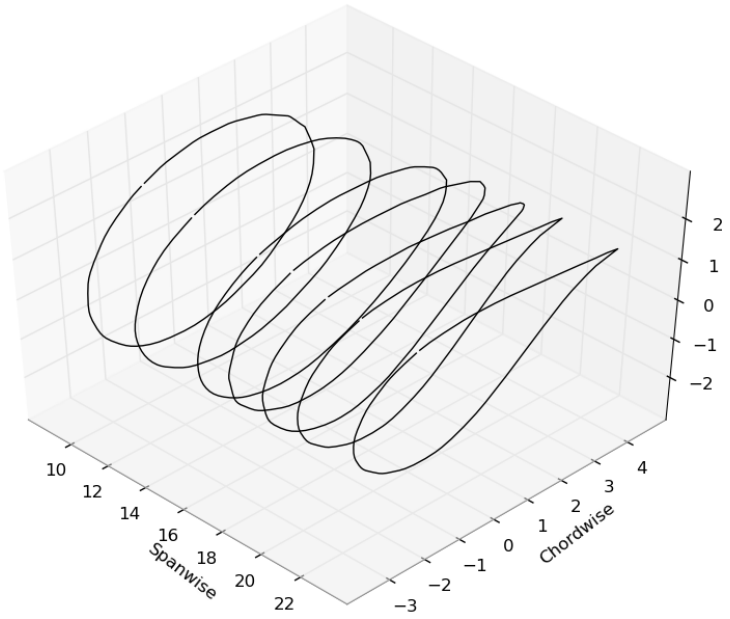


Figure 2.4: *Station shapes defined (stations 10-16)*

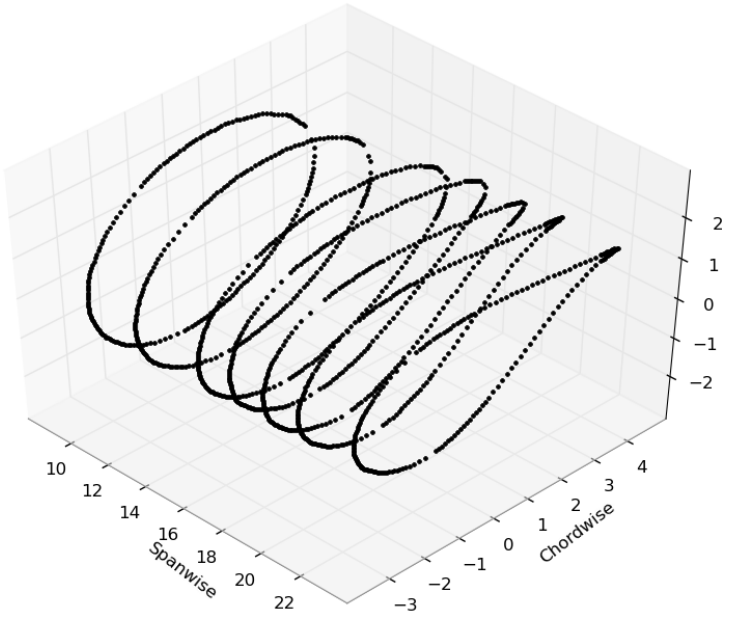


Figure 2.5: *Chordwise seeding applied (stations 10-16)*

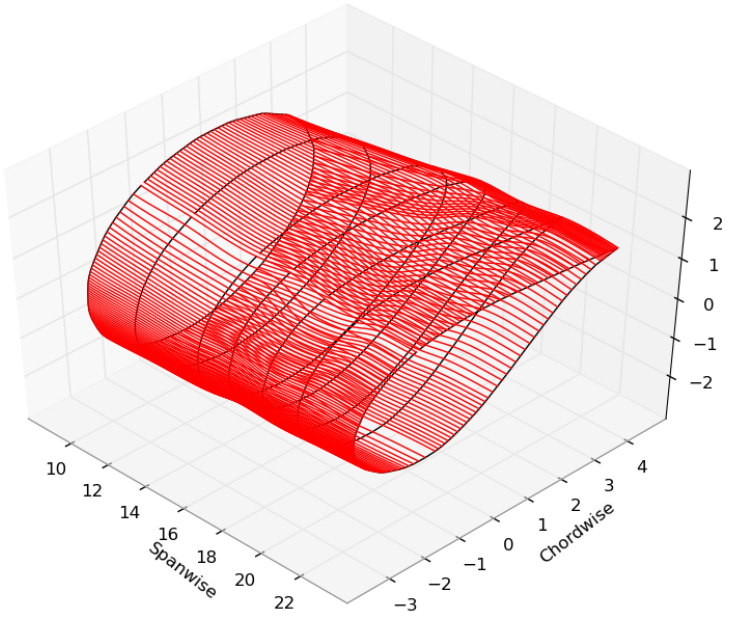


Figure 2.6: *Spanwise splines created from chordwise seeding (stations 10-16)*

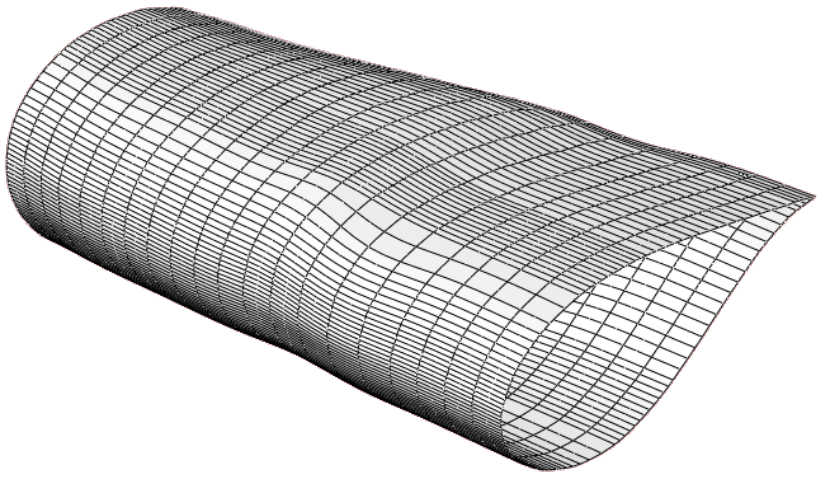


Figure 2.7: *Completed mesh seen in ABAQUS CAE (stations 10-16)*

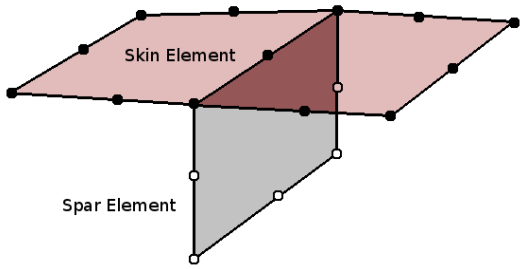


Figure 2.8: *Joining of a shear web element with skin elements*

Since there are two shear webs that run the length of the blade, there are 4 spanwise splines that define the connecting point of the shear webs and the skin. The chordwise seeding subroutine automatically aligns the nodes at the specified spar locations. There is a parameter that determines the number of nodes created between the top and bottom of the shear web. This parameter is used to interpolate between the two splines that define the top and bottom of each shear web. Since the shear webs are physically connected to the skin, they need to be mathematically connected to them also. To do this, the shear webs have direct nodal connectivity to the skin nodes at that location. Figure 2.8 shows the shear web element sharing three nodes with skin elements. In BroncoBlade v0.1, the adhesive layer bonding the shear web and skin is not modeled.

2.3.2 Material Assignment

One of the inputs for BroncoBlade is a skin laminate definition. BroncoBlade divides the skin mesh into four chordwise sections and as many spanwise sections as are created between the stations. BroncoBlade then reads the laminate definition, and assigns each section its own shell composite definition. This is an advantage of BroncoBlade, since in a GUI model the user would have to manually select and specify the elements in each of the 100+ sections and then list the section's layup.

2.3.3 FEA file writing

Once all of these nodes, elements, and materials have been created inside of BroncoBlade, they must be stored in a format that can be used by a finite element program. BroncoBlade v0.1 supports ABAQUS; additional formats may be supported in the future, or can be written by the user without difficulty. The current version writes the FILE_mesh.inp file in this format:

- Part Name
- Nodes
- Elements
- Structural section assignments

  list of elements in section

  composite laminate assigned to section
- Aerodynamic surface creation

  list of elements in section

  surface creation command
- Station node sets creations
- Reference node set creation

2.4 Spanwise Properties: The *Sections* Module

With the complete model from the Blade module, the next step in evaluating a blade is to calculate the appropriate loads. BroncoBlade utilizes the turbine simulator, FAST, to predict the behavior of the blade during operation. Though most of the inputs to FAST are user specified, BroncoBlade is responsible for calculating stiffness and mass properties along the blade, and for computing the blade's mode shapes. These properties are calculated using the Sections module (see Figure 2.9) and are written out to a data file that defines the blade's structure

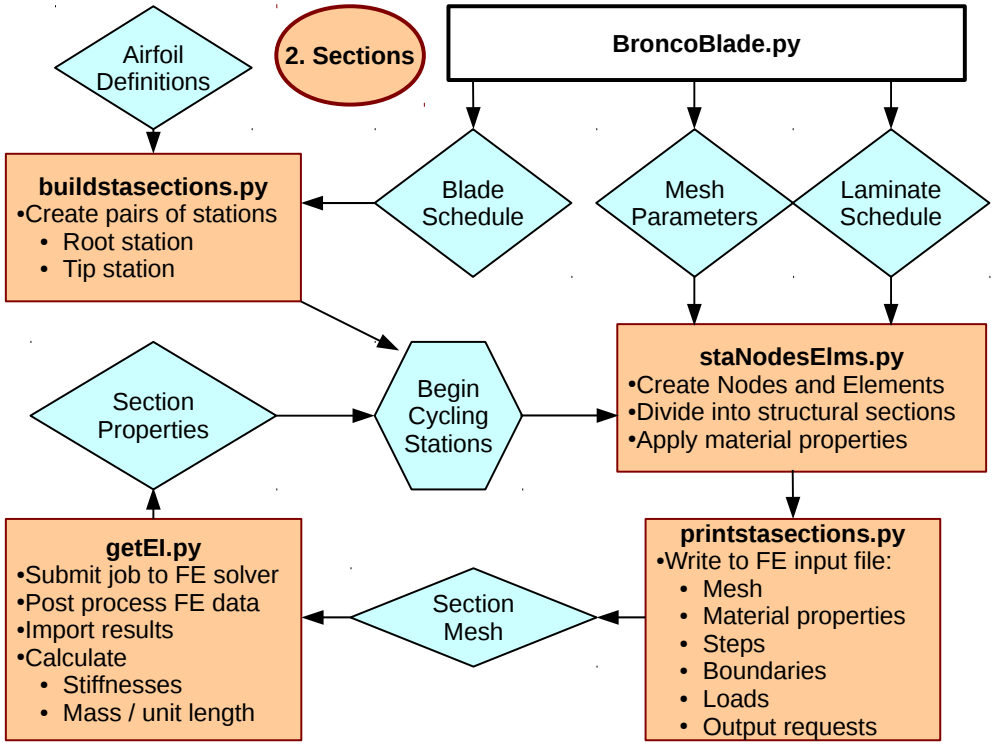
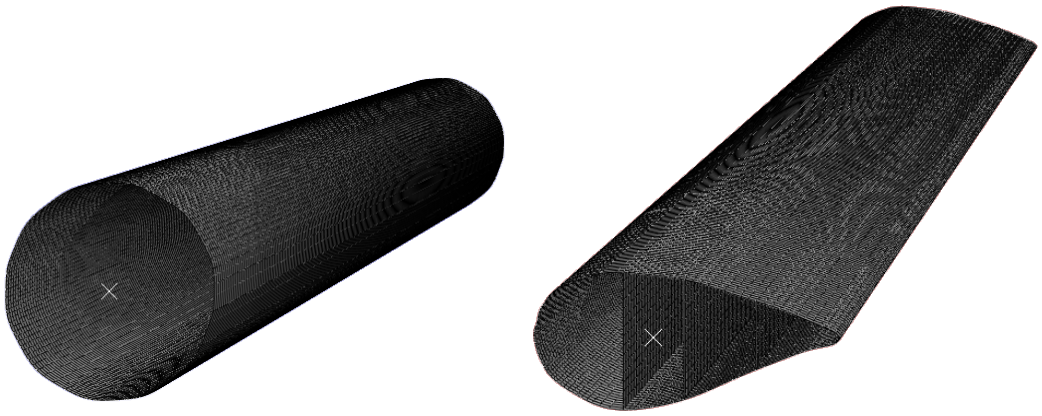


Figure 2.9: *Flow of programs and data in the Sections module*

for use in the FAST simulation.

For each of the stations, FAST requires both the flapwise and edgewise bending stiffnesses and the linearized mass. While some tools available for computing these properties analyze the 2D cross section, BroncoBlade uses its previously created subroutines to create finite element models of 3D beams with a constant cross section equivalent to each of the station cross sections, such as the models in Figure 2.10.

Each beam is created using a similar process as what was used for mesh generation in the Blade module. For example, rather than creating a mesh over many different stations, only one station is used to define the mesh. During the investigation of the properties of Station 14, a station is created at the beam's root with the xy values of station 14 and z-values set to 0. Another station with the same



(a) Station 0 beam (b) Station 14 beam

Figure 2.10: *Extrusions of stations into beams for property calculations*

cross section is created at the tip, identical to the root in x and y coordinates, but having a z-coordinates equal to 20 times the chord length. These z-coordinates are chosen to ensure the model can be idealized as an Euler-Bernoulli beam. These two identical stations are then used to create spanwise splines, nodes, elements, and composite layup assignments. The resulting mesh is of a beam with the extruded cross section of Station 14. This is then printed into an ABAQUS input file, along with the loads and boundary conditions.

In order to calculate the bending stiffness of the beam, three analyses need to be run. The first calculates the center of twist for the beam. This allows for the displacement due to twisting to be isolated from the the bending result. The second and third analyses apply a load and use beam bending theory to calculate the stiffness in the flapwise and edgewise directions. Equation 2.1 is a basic beam bending equation that can be rearranged into Equation 2.2, where it is used to calculate the effective EI value. Further discussion of the methods used for these analyses is in Section 3.4.

$$\delta_{tip} = \frac{PL^3}{3EI} \tag{2.1}$$

$$EI = \frac{PL^3}{3\delta_{tip}} \tag{2.2}$$

For each of the beams, the EI values are calculated, providing stiffness values along the blade as an input for FAST. The linear mass calculation is done by requesting the total mass of the beam from the solver, then dividing by the beam length. The stiffness and mass values are assigned to the appropriate stations; all of the stations and their respective attributes are then saved to file for future use.

While running this simple bending analyses gives the required stiffness, the time required to run the FE models is larger than what is desired. Improving the computational efficiency of the Sections module is discussed in the future work chapter.

2.5 Blade Frequencies: The *Modes* Module

In the Modes module (see Figure 2.11), mode shapes are calculated by using the model generated in the Blade module and performing a frequency analysis on it in an FE program. BroncoBlade generates the input file for an ABAQUS job, submits the job for analysis, then reads the output file, and calculates the first two flapwise mode shapes and first edgewise mode shape using a least-squares fit.

Once the stiffness values and mode shapes have been calculated, BroncoBlade writes them into the ".bld" file that is used by FAST.

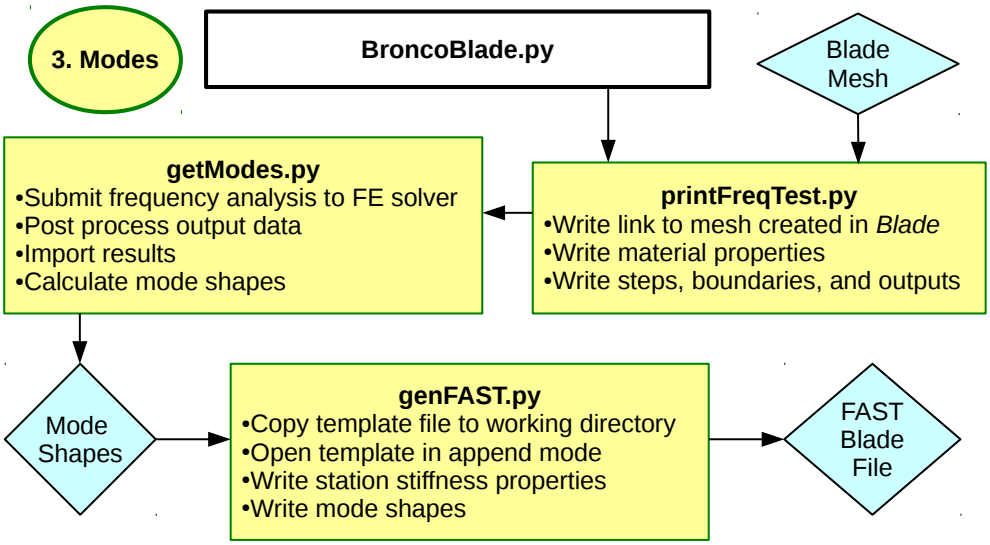


Figure 2.11: *Flow of programs and data in the Modes module*

2.6 Running Operational Simulation: FAST

Because FAST is a detailed software package with ample documentation [11], only the basic requirements for operation with BroncoBlade will be outlined here.

2.6.1 Inputs

The following files are required to run a basic analysis with no turbine operations (yawing, pitching, etc):

- *Primary.fst*

  the main input file outlining the analysis parameters, linking to the other input files, and specifying the data to be written to the output file

- *File_AD.ipt*

  the Aerodyn input file, linking to the wind input file, the airfoil data files, and outlining the aerodynamic profile of the blade. aerodynamic

- *File_Tower.dat*

  tower property definitions

- **File_Blades.dat**

  definition of the mass, stiffness, and modes of the blades; generated by BroncoBlade.

- *wind.wnd*

  specification of the wind condition for the simulation; can be generated by the IECWind program [1].

- *airfoil.dat*

  contains lift and drag coefficients as functions of the angle of attack; there should be one of these files for each of the airfoil shapes referenced in the Aerodyn file.

2.6.2 Outputs

The desired FAST outputs can be requested at the bottom of the Primary.fst file. The selected parameters are printed into a tabular data file called Primary.out. This file is not used directly by BroncoBlade, but is very useful in examining the performance of the turbine. For example, in the validation chapter, the tip deflection and root moments are taken from this file.

FAST uses the program AeroDyn [12] [16] to calculate the aerodynamic loads on the blade. By turning on the PRINT flag in the Aerodyn_AD.ipt file, the aerodynamic data of the simulation will be printed out to a file called Primary.elm. The normal and tangential forces for each aero-section are listed in this file, which are read by BroncoBlade's Loads module.

2.7 Applying FAST Results: The *Loads* Module

The purpose of the Loads module (see Figure 2.12) is to take the results from FAST and use them to prepare a finite element analysis. The Loads module

takes an input of a ".elm" file that contains the aerodynamic loads of the FAST simulation. These loads are given in pairs of flapwise and edgewise forces for each of the aerodynamic sections. BroncoBlade uses the aerosection's twist to rotate the loads into global x and y coordinates.

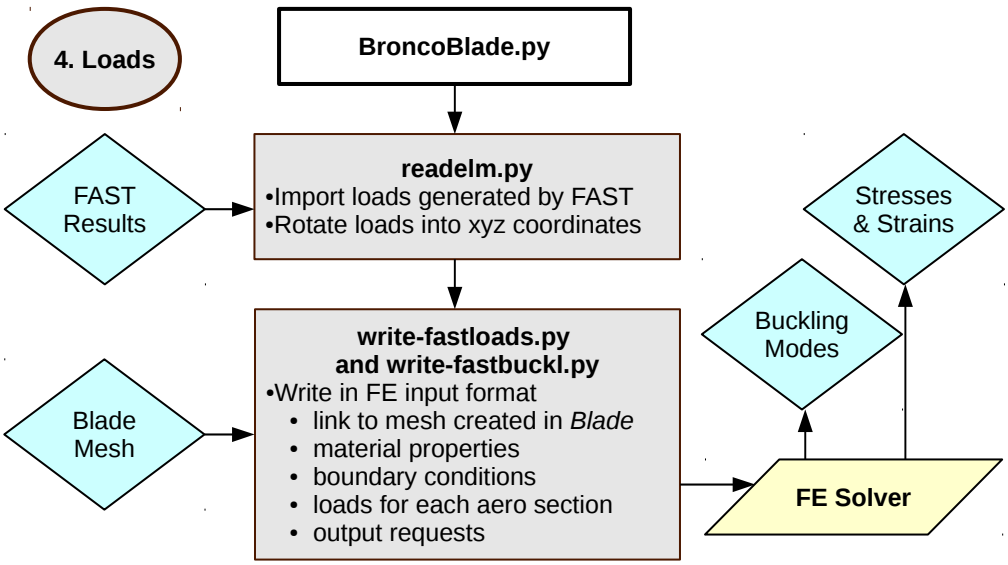


Figure 2.12: *Flow of programs and data in the Loads module*

In BroncoBlade, the elements of each aerosection are stored in a list. Because individual element areas are calculated, the elements in the aerosection can be summed to give the surface area of the section. This allows BroncoBlade to distribute the load over the entire aerosection using a generalized surface traction. This helps avoid the use of artificial concentrated forces.

BroncoBlade then writes out two separate FE files. Both of these use ABAQUS's **include* command [17] to reference the mesh created in the Blade module and apply the calculated section pressures. The only difference is that one specifies a static analysis and the other a buckling analysis.

The writing of the these two files ends the current version of BroncoBlade. The user can then manually run the analyses and interpret the results.

Chapter 3

Methods

BroncoBlade is a pre-processor for both the turbine simulator FAST and the structural finite element analysis program ABAQUS. Both of these external programs allow for highly detailed models that go beyond the current capabilities of BroncoBlade. The choices of inputs for these simulations are important for properly modeling the turbine and obtaining accurate results. This chapter discusses the techniques that are used in these analyses.

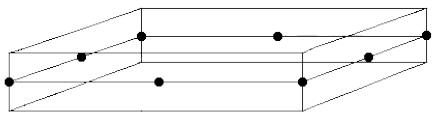
3.1 Element Selection

In structural mechanics, the finite element method is often used to perform analysis on complex structures that can not be solved using elementary engineering equations. The FE method uses a structure discretized into elements where the engineering equations can be directly applied [4]. Though in theory the finite element method can be performed by hand, in practice it requires a computer to perform the thousands of required calculations.

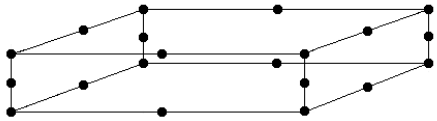
As with any simulation, the quality of results depends directly on the quality of the modeling methods. Element selection is a very important part of the finite element method. Choosing an inappropriate element can cause errors in

the results, failure to capture certain phenomena, or an unnecessary increase to the computational cost of the model. Because of the curves on the blade's skin a second order element has been selected. The addition of side nodes in a second order shell element allows it to have a curved geometry. A first order element mesh would require a many more elements to adequately approximate the curvature.

Most of the available structural elements can be divided into the beam, shell, and solid element types, with each type having its own advantage in specific situations. For modeling a wind turbine blade, qualifications for use of shell elements have been met, and the features of the shell elements have been determined to be the most advantageous. The main requirement for a structure to be modeled using shell elements is that the thickness of the structure be small in relation to the other dimensions. In the SNL100-00, for example, the maximum thickness of the skin is 170.6 mm, occurring at the root. In comparison to the 100m blade length, and the 5.694m chord at the root, the thickness is small enough for the skin and shear webs to be idealized as shells.



(a) Shell element with 8 nodes and 48 DOF



(b) Solid element with 20 nodes and 60 DOF

Figure 3.1: *Two second-order elements*

The main alternative to shell elements (Figure 3.1a) in the situation would be 3D solid elements (Figure 3.1b). In ABAQUS, the general 3D solid element has 20 nodes with 3 translational degrees of freedom on each [17]. These solid elements are lack rotational degrees of freedom on their nodes. Since the turbine is mainly subjected to bending and torsional loads, and the skin undergoes very little out-of-plane compression or tension, solid elements are non-ideal compared to shell elements. Shell elements have three rotational degrees of freedom for each

node, providing an advantage in bending analyses.

Shell elements do not require the discretization of the skin thickness during the creation of the mesh, so it is much easier for BroncoBlade to create a mesh of shell elements than solid elements. The shell element meshing in BroncoBlade can be created by defining the surface of the outer skin and then defining the numerical value for the element thickness later. Similarly, a composite laminate can simply be assigned to the shell element, with every ply represented in the single shell element.

A solid mesh would require the 8 nodes of the top surface of the solid elements to form the exterior of the blade, and another set of 8 nodes to define the interior. Since the thickness is not constant throughout the blade, the interior surface would have to be smoothed out to avoid any discontinuities. Also, applying the composite layup would be more challenging, requiring an element to be created for each ply.

3.2 Mesh Density Convergence

For every finite element model, the mesh must be of adequate quality; a coarse mesh or poorly formed elements can cause errors in the analysis results. Typically, an increasing quality of mesh will approach the "correct" results asymptotically, so after a certain point the mesh refinement does not improve the results much but still increases the computational cost. An ideal mesh is refined to give accurate results without spending extra resources on negligible improvement.

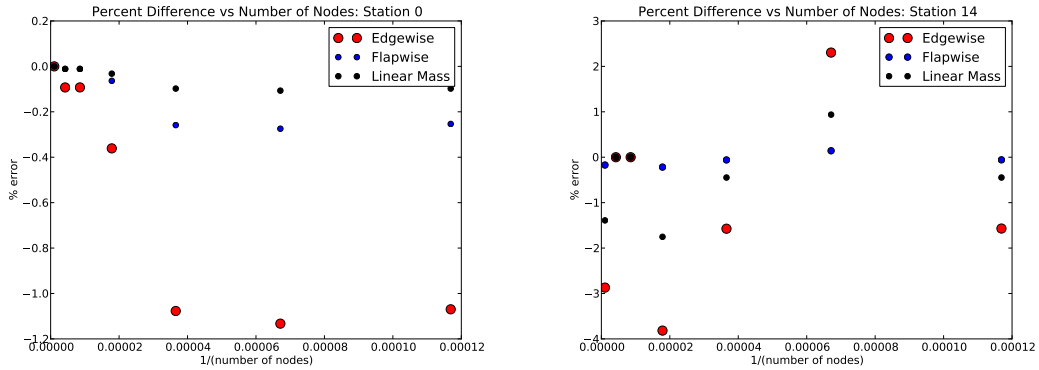
A mesh convergence study was conducted using BroncoBlade to determine an appropriate mesh for the airfoil beams created in the Sections module. Two of these sections were chosen to be studied for convergence of the stiffness calculation. The station 0 beam has a cylindrical cross section and no shear webs, while the

station 14 beam has an DU99-W-405 airfoil cross section and three shear webs. Seven analyses were performed for each of the beams, each with different levels of mesh quality. The spanwise, chordwise, and total node counts are shown in Table 3.1.

Table 3.1: Sensitivity Study Meshes

Mesh Number	Chordwise Seed	Spanwise Seed	Total Nodes	Inverse Nodes
1	94	91	8554	116.90e-06
2	78	191	14898	67.12e-06
3	94	291	27354	36.55e-06
4	114	491	55974	17.86e-06
5	238	491	116858	8.55e-06
6	238	991	235858	4.23e-06
7	476	1991	947716	1.05e-06

The results of the analyses are shown in Figure 3.2 with the stiffness results being shown as a percent difference from the accepted converged value. Using this percent difference allows for the flapwise stiffness, edgewise stiffness, and linearized mass to be compared on one plot. The mesh density is displayed on the x-axis as the inverse of the node count in the model.



(a) Station 0 beam

(b) Station 14 beam

Figure 3.2: *Sensitivity analysis results for two station section beams*

First, Figure 3.2a shows that in the station 0 beam the EI and mass values

converge as the mesh density increases. The edgewise stiffness shows the most sensitivity to the mesh, followed by the flapwise stiffness, and then by the linearized mass. Analysis 5 shows good quality results, with differences less than .01%. Mesh 6 and 7 show little change in results even though they greatly increase the node count, so mesh 5 is chosen.

However, the results for station 14 in Figure 3.2b do not show an expected trend. The flapwise stiffness shows convergence towards meshes 5 and 6, but the more sensitive edgewise stiffness does not converge with the increase node count. Using the converged value of mesh 5 from the station 0 study, the percent difference for the edgewise stiffness ranges up to 3.5%. Because it appears that further refinement of the mesh would not have much result and there is no clear convergence point, and because the 3% difference between the meshes 5, 6, and 7 is deemed not to be catastrophic, mesh 5 has been chosen as the default mesh for the Sections module.

The trouble with the station 14 beam is likely due to the nature of chordwise seeding. In order to apply the the proper materials and composite layups, the beam is divided into five chordwise sections that have hard boundaries. BroncoBlade currently uses a chordwise seeding parameter that specifies the mesh inside of each of these boundaries. The distribution of elements across these boundaries can be uneven, and since the total chordwise seed number does not account for *where* the seeds are on the cross section the effects of seeding can be hidden. This point is revisited in the future work chapter with a suggestion of changing the meshing algorithm and interface.

3.3 Boundary Conditions

Every HAWT blade is fixed at the root to the rotor hub, either with bolts or some other mechanical fixture. Other than this single point of constraint, the blade is free to move. To model this as a boundary condition in the FE model, all of the nodes at the spanwise location of z=0 have all six degrees of freedom constrained. This prevents any translation or rotation from occurring and sufficiently removes any singularity from the models stiffness matrix.

For the analyses performed in the Sections module, the *root* of the beam will not always be the circular shape that it is in the full blade model. Still, the nodes on the z=0 edge of the beam are constrained in all six degrees of freedom, as shown in Figure 3.3, making the model a cantilever beam problem.

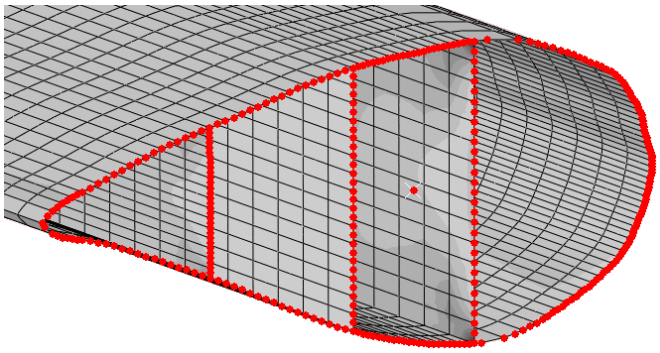


Figure 3.3: *Skin nodes at the edge tied to a central reference node*

3.4 Section Modeling

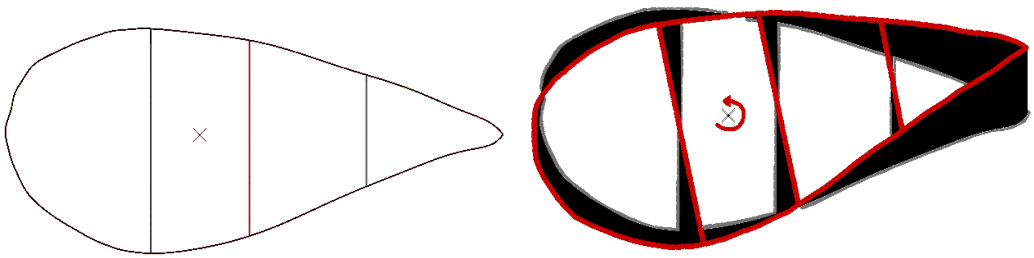
Two different load sets are applied in the BroncoBlade process. In the Sections module, analyses are run to calculate the stiffness properties at each station, so the load is arbitrary. In the Loads module, the resulting aerodynamic forces of FAST are applied to the blade.

In order to calculate the bending stiffness of the beam, three analyses need to

be run. In all three, the nodes on the root edge are constrained in all 6 degrees of freedom, and the nodes on the tip edge are fully tied to a single reference node. This reference node is where loads are applied, such that the whole edge sees equal loading. Each analysis records the displacement of two *measured nodes*, which are located on the leading and trailing edge on the loaded end of the beam. Because of the beam's asymmetric shape, it will undergo twisting during the bending loads, so the displacement of any given node will have both bending and twisting components.

3.4.1 Torsional Analysis

The purpose of the first analysis is to find the center of twist of the beam. This allows the twisting component to be isolated from the calculation of displacement in analyses two and three, leaving only the deflection due to bending. The center of twist is calculated by applying a 1 Nm torque load to the reference node, and thus twisting the beam as shown in Figure 3.4. BroncoBlade assumes that the cross sectional shape of the beam will not change, and so the chord will retain the same length and rotate about some unknown center. This first assumption is made because the load applied is very small, so non-linear deformation and distortion of the beams profile is not expected.



(a) Unloaded (b) Torsional Load

Figure 3.4: *Section beam (scaled deflections)*

BroncoBlade also assumes this center will lie on the chord, which allows the center's location to be easily calculated. This second assumption was made after trying an algorithm that calculated the center as an arbitrary point, not necessarily lying on the chord. This algorithm experienced difficulties because it utilized the angle formed between the leading edge node, the trailing edge node, and the center. Since the center was very close to being on the line chord, this angle was approaching zero, and caused problems with the precision of the mathematics packages being used.

The calculated location of the center is notated as a fraction of the chord, with 0 being at the leading edge, .5 being in the middle, and 1 being at the trailing edge. This center fraction value is added as an attribute of the station by BroncoBlade, and so is saved to a data file when the *save Stations Module* (-Ss) argument is used. This matters because the center fraction value is also required to run the Modes module, so either the Sections and Modes modules must be run sequentially in the same calling of BroncoBlade, or the Modes module can use the *load stations into Modes Module* sub-argument (-Ml) to retrieve station data and thus the center fraction value.

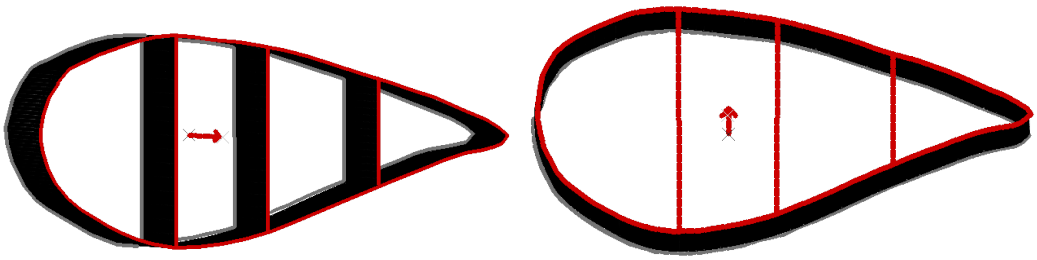
3.4.2 Flapwise and Edgewise Analyses

In the flapwise analysis, a 1 N force is applied in the y-direction. Similarly, the edgewise analysis applies a 1 N force in the x-direction. Both of these use the displacements of the measured nodes (u and v for x and y, respectively) and the center fraction (C_{frac}) found in the torsional analysis to calculate the displacement of the center of twist using Equations 3.1 and 3.2, where u and v are the translations in the x and y directions, respectively, and C_{frac} is the location

of the center of twist as a fraction along the chord.

$$u_{center} = u_{leading} + (u_{trailing} - u_{leading}) * C_{frac} \tag{3.1}$$

$$v_{center} = v_{leading} + (v_{trailing} - v_{leading}) * C_{frac} \tag{3.2}$$



(a) Edgewise (b) Flapwise

Figure 3.5: *Loading of station beam for stiffness calculation (scaled deflections)*

These center values reflect the deflection of the beam due to only bending, and thus can be used in the basic beam bending Equation 2.2 to calculate EI.

Chapter 4

Validation of BroncoBlade Using SNL100-00

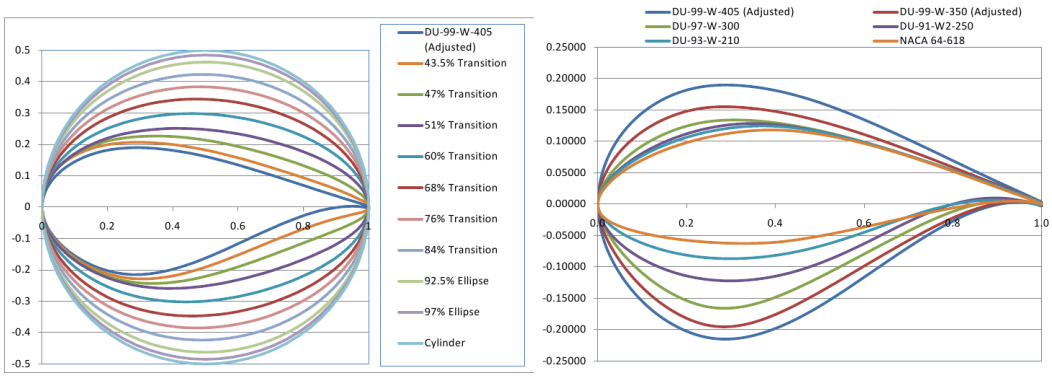
To qualify BroncoBlade as an accurate and useful modeling tool, it needed to be validated against generally accepted results. The chosen model to validate with is the SNL100-00 turbine created by Sandia National Laboratories, which has published model inputs and analysis results [7] [8].

4.1 BroncoBlade Input Data

As import as the analysis process is to the accuracy of results, equally important are the inputs to the analysis. The SNL100-00 turbine was attractive because of the documentation of the turbine's properties in addition to the simulation results. The input parameters primarily come from source [7]. Some of the numerical data, such as the FAST input files, was taken from models acquired from Sandia.

4.1.1 Airfoils

The SNL100-00 design is an based on scaled models of the NREL 5MW turbine and the UpWIND 5MW, which both use the airfoils from the DOWEC turbine. Unlike similar NACA airfoils that have many free profile generation tools available, there appeared to be no coordinate definitions of these DOWEC airfoils publicly available. Fortunately, the profile data files were provided in the requested Sandia model. The airfoils used in the SNL100-00 are shown in Figure 4.1.



(a) Root and Transition (b) Outboard

Figure 4.1: *Geometries of the airfoils used in the SNL100-00 [7]*

4.1.2 Airfoil Schedule

With the normalized airfoil shapes defined, the overall geometry of the blade can be created from positioning, resizing, and rotating the airfoils at positions along the blade. Thus, the exterior of the blade's skin is defined by the airfoil schedule listed in Table 4.1. Each of the 34 airfoils placed along the blade is referred to as a *station*, starting at the root of the blade with Station 0, and ending at the tip with Station 33.

Each station's spanwise position is defined by a fraction of the blade, and is thus normalized to the total blade length. If the spanwise position were to be listed in meters instead, the table would have to specify whether the distance was

Table 4.1: Airfoil Schedule [7]

Station Number	Spanwise Fraction	Airfoil Type	Chord (m)	Twist (degree)	Pitch Axis Fraction	Thickness to Chord Ratio
0	0.000	circle	5.694	13.308	.5	1
1	0.005	circle	5.694	13.308	.5	1
2	0.007	circle	5.694	13.308	.5	.9925
3	0.009	circle	5.694	13.308	.5	.985
4	0.011	circle	5.694	13.308	.5	.9775
5	0.013	circle	5.694	13.308	.5	.97
6	0.024	circle	5.792	13.308	.499	.931
7	0.026	circle	5.811	13.308	.499	.925
8	0.047	transition840	6.058	13.308	.498	.840
9	0.068	transition760	6.304	13.308	.468	.760
10	0.089	transition680	6.551	13.308	.453	.68
11	0.114	transition600	6.835	13.308	.435	.6
12	0.146	transition510	7.215	13.308	.410	.51
13	0.163	transition470	7.404	13.177	.400	.47
14	0.179	transition435	7.552	13.046	.390	.435
15	0.195	DU99W405	7.628	12.915	.380	.405
16	0.222	DU99W405	7.585	12.133	.378	.38
17	0.249	DU99W350	7.488	11.350	.3725	.3
18	0.276	DU99W350	7.347	10.568	.375	.34
19	0.358	DU97W300	6.923	9.166	.375	.30
20	0.439	DU91W250	6.429	7.688	.375	.26
21	0.520	DU93W210	5.915	6.180	.375	.23
22	0.602	DU93W210	5.417	4.743	.375	.21
23	0.667	NACA64618	5.019	3.633	.375	.19
24	0.683	NACA64618	4.920	3.383	.375	.185
25	0.732	NACA64618	4.621	2.735	.375	.18
26	0.764	NACA64618	4.422	2.348	.375	.18
27	0.846	NACA64618	3.925	1.380	.375	.18
28	0.894	NACA64618	3.619	0.799	.375	.18
29	0.943	NACA64618	2.824	0.280	.375	.18
30	0.957	NACA64618	2.375	0.210	.375	.18
31	0.972	NACA64618	1.836	0.140	.375	.18
32	0.986	NACA64618	1.208	0.070	.375	.18
33	1.000	NACA64618	0.100	0.000	.375	.18

from the root of the blade, or from the center of rotation, which would add the 2.5m hub radius to each of the positions.

The normalized airfoil shape is listed, specifying the profile that will be used as the starting point for the creation of the station. This profile, which by default has a chord length of 1, is scaled to match the station's listed chord length. The airfoil is rotated about its specified pitch axis by its twist value. In the SNL100-00, the tip is listed as being at 0°, and the root is twisted to 13.308°.

The final parameter in the schedule is the thickness to chord ratio. In the model distributed by Sandia, there is an airfoil profile for each individual shape of airfoil, including those that only differ in the thickness to chord ratio. For example, there is a data file with the coordinates for both of the DU93W210 variations: *DU93-W-210_23.txt and DU93-W-210.txt*. Though this is perfectly acceptable, only one is required in BroncoBlade, since part of the station creation code will scale the airfoil in the y-direction so it has the appropriate thickness to chord ratio.

BroncoBlade is a unit-less program, so the user is responsible for keeping units consistent. The SNL100-00 uses kilograms, meters, and seconds as it's unit system, so this convention will be used for the remainder of the chapter.

4.1.3 Shear Webs

With the airfoil schedule defining the external surface of the blade, the next step is to define the internal support structure. The NREL and UpWind turbines both have two shear webs running almost the length of the blade. After finding insufficient buckling performance in the first design iteration, a third shear web was added to the SNL100-00 that runs for a shorter section of the blade. The position of the shear webs are shown in Figure 4.2. The two main shear webs are connected by a spar cap, so the combined *box spar* is represented by the blue

rectangle. The third shear web is represented by the red line.

Figure 4.2: *Location of shear webs in the SNL100-00 [7]*

Shear web 1 is located at a fixed position of -0.75m along the chord from the pitch axis, and shear web 2 is located at 0.75m on the other side of the pitch axis. Both of these begin at 2.4m from the root of the blade and end at 94.4m from the root. The orientation of these two webs match with the standard shear web creation algorithm in BroncoBlade, which assumes that the shear webs will have the same location in reference to the local pitch axis and would essentially be straight beams if not for the global twist of the blade.

Shear web 3, however, does not follow the same conventions as webs 1 and 2, and must be manually created and added to the mesh that BroncoBlade generates. Shear web 3 begins at 14.6m from the root, at a chordwise location of 68% of the chord. It expires at 60.2m from the root, with a chordwise location of 78% of the chord. ABAQUS CAE was used to create a shell at this location between the top and bottom surfaces of the blade. The shear web shell was meshed and constrained to the skin mesh using ABAQUS utilities.

4.1.4 Materials

For all structural engineering problems, material properties are a necessity. Modern HAWT blades are constructed mainly from composite and foam or wood core materials. Unlike metals, which have well documented material properties for hundreds of variations, the composite materials do not have accepted standard values because of the great varieties of fibers, matrices, and configurations that go into making a composite laminate. Because of this, the validation must use the

material properties defined by Sandia, rather than using another reference. Table 4.2 contains the properties used in BroncoBlade for the SNL100-00 model. The first three materials in the table are fiberglass laminates, listed with the lamina properties required for an FE model. The last three are considered to be isotropic and are defined in the FE model by only their Elastic Modulus and Poisson's Ratio. A shear modulus is listed in the table for the isotropic materials, but it is not included in any analysis.

Table 4.2: Material Properties [7]

Material	Stacking Sequence	Density $^{kg}/_{m^3}$	E_1 GPa	E_2 GPa	ν_{12}	G_{12} GPa	G_{13} GPa	G_{23} GPa
uniaxial	$[0]_2$	1920	41.8	14.0	0.28	2.63	2.63	2.63
biaxial	$[\pm 45]_4$	1780	13.6	13.3	0.5	11.8	11.8	11.8
triaxial	$[\pm 45]_2[0]_2$	1850	27.7	13.65	0.395	7.22	7.22	7.22
foam	-	200	0.256	*	0.3	2.2*	*	*
gelcoat	-	1235	3.44	*	0.3	1.38*	*	*
resin	-	1100	3.5	*	0.3	1.4*	*	*

4.1.5 Composite Layup

The SNL100-00 is divided into the two main structural sections: the skin and the shear webs. The shear webs have a laminate consisting of a 80mm layer of foam between two layers of 5mm biaxial fiberglass (Table 4.3). This layup includes the extra shear web 3. This layup is both balanced and symmetric.

Table 4.3: Composite Layup: Shear Webs [7]

Material Type	Thickness (mm)
Biaxial	5
Foam	80
Biaxial	5

The skin of the turbine blade can be characterized as having a basic composite layup with extra reinforcement added in different sections of the blade. The basic

layup is shown in in Table 4.4, with the first entry being the exterior gelcoat and the subsequent layers moving towards the interior of the skin. The inner most layer is 5mm of resin, which is not physically present, but represents the parasitic mass from excess resin throughout the other lamina of the skin. Inside of this layup, there is a layer of *reinforcement*, which can change for each element based on its spanwise and chordwise location. The reinforcement for each of these sections is listed in Table 4.5.

Table 4.4: Composite Layup: Skin [7]

Material Type	Thickness (mm)
Gelcoat	0.6
Triaxial	5
Reinforcement	*see Table 4.5*
Triaxial	5
Resin	5

The skin of the blade is divided in both the chordwise and spanwise directions. The four chordwise divisions are the leading edge, the spar, the aft panel, and the trailing edge. The spanwise divisions are based on the stations, which is 34. This would yield a total of 136 sections, but the tip sections (29-33, after the shear webs have ended) are not divided chordwise. Each of these sections can have a different composite layup.

4.2 Comparison of Calculated Blade Properties

Because of the complexity of these models it is advantageous to compare the results of the different steps in the process, rather than only comparing final results. This helps to catch mistakes early and avoid wasting time running analyses on models with fundamental errors. Comparing against the Sandia results step by step also shows the strengths and weaknesses of BroncoBlade from start to finish.

Table 4.5: Laminate Reinforcement Positioning [7]

Units in (mm)

Station Number	Spanwise Fraction	Root Build-up	Spar Cap	Trailing Edge	Trailing Edge	Leading Edge	Aft Panel
Material		triax	uniax	uniax	foam	foam	foam
0	0.0	160					
1	0.005	140	1	1			
2	0.007	120	2	2			
3	0.009	100	3	3			
4	0.011	80	4	5			
5	0.013	70	10	7		1	1
6	0.024	63	13	8		3.5	3.5
7	0.026	55	13	9		13	13
8	0.047	40	20	13		30	100
9	0.068	25	30	18		50	100
10	0.089	15	51	25	60	60	100
11	0.114	5	68	33	60	60	100
12	0.146	0	94	40	60	60	100
13	0.163		111	50	60	60	60
14	0.179		119	60	60	60	60
15	0.195		136	60	60	60	60
16	0.222		136	60	60	60	60
17	0.249		136	60	60	60	60
18	0.276		128	30	40	60	60
19	0.358		119	30	40	60	60
20	0.439		111	15	20	60	60
21	0.520		102	8	10	60	60
22	0.602		85	4	10	60	60
23	0.667		68	4	10	60	60
24	0.683		64	4	10	55	55
25	0.732		47	4	10	45	45
26	0.764		34	4	10	30	30
27	0.846		17	4	10	15	15
28	0.894		9	4	10	10	10
29	0.943		5	4	10	5	0
30	0.957		5	4	10	5	
31	0.972		5	4	10	5	
32	0.986		5	4	10	5	
33	1.000		0	0	0	0	

Because it does not require running any analysis, checking the material composition of the model is the first criterion for model validation. Major errors in geometry, material properties, or laminate assignments will cause a noticeable difference in the the overall mass of the model or in the mass of individual materials. Opening the model in the ABAQUS CAE GUI allows the user to inquire on the mass content of the entire blade, specific sections, or specific materials. Table 4.6 compares the acquired mass from ABAQUS CAE with the listed material usage by Sandia. In addition to this method, FAST also calculates an approximate blade mass during analysis, which gave a mass of 114,679 kg. Sandia reported a value of 115,684 kg calculated using PreComp, and 118634 kg using ANSYS. This shows that there is a bit of variation in the mass calculation depending on the method used, which allows for the conclusion that the results in Table 4.6 are acceptable.

Table 4.6: Material Usage in Blade

Material Laminate	Density ($^{kg}/_{m^3}$)	SNL Mass kg	SNL[7] Percentage of total	Mass kg	Percentage of total	Difference vs SNL kg	Percent Difference vs SNL of total
Biaxial	1,780	4,112	3.6	3,996	3.3	-115.6	-2.81
Foam	200	15,333	13.3	15,059	12.5	-273.29	-1.78
Gelcoat	1,235	920	0.8	927	0.77	7.19	0.78
Resin	1,100	6,863	5.9	6,873	5.7	10.38	0.15
Triax	1,850	38,908	33.6	41,483	34.5	2,575	6.618
Uniax	1,920	49,527	42.8	52,054	43.2	2,527	5.104
Total		115,663	100	120,394	100	4,731	4.09

Once the total mass of the model has been checked, the next level of validation is with its structural properties. The sections module of BroncoBlade analyzes the blade properties at each of the stations. These properties are used in the FAST analysis, and thus are very important for the aerodynamic simulation. The linear mass density performs all of the checks done previously in the material mass validation, but additionally checks for proper distribution of materials.

The flapwise and edgewise stiffness values are more sensitive to errors in the model than the mass values are. The stiffness is defined as EI, which is the Elastic

Modulus (E) multiplied by the area moment of inertia (I). First, accurate EI values require that the composite layup be properly oriented, otherwise the anisotropic nature of the fiberglass will yield erroneous results, since the expected E_{11} stiffness will not be acting in the desired direction. Secondly, the area moment inertia is sensitive to small changes in cross sectional geometry. These two factors combine to make the stiffness values the most stringent of pre-analysis validation checks.

Figures 4.3, 4.4, and 4.5 show the correlation between the results calculated in BroncoBlade and the values released by Sandia [8] for the linear mass, flapwise, and edgewise stiffness, respectively. Though there are some minor departures, the results provide an acceptable level of correlation.

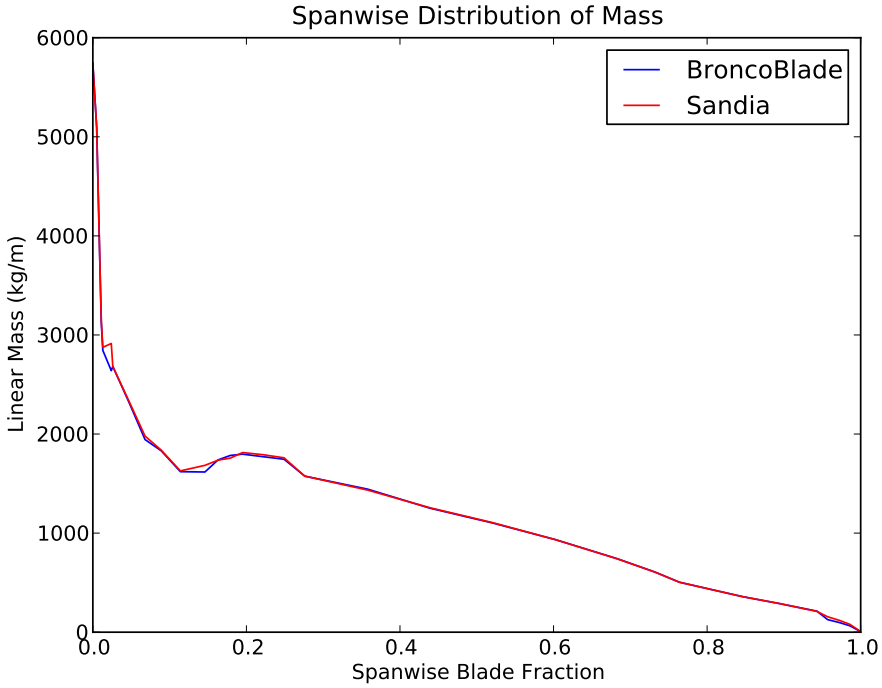


Figure 4.3: *Comparison of linear mass: Sandia and BroncoBlade*

The results from the Modes module are also used to validate the model. FAST requires a minimum of the first flapwise and edgewise mode shapes, and allows for the optional inclusion of the second flapwise mode. Each mode shape is defined by listing the coefficients for the Equation 4.1.

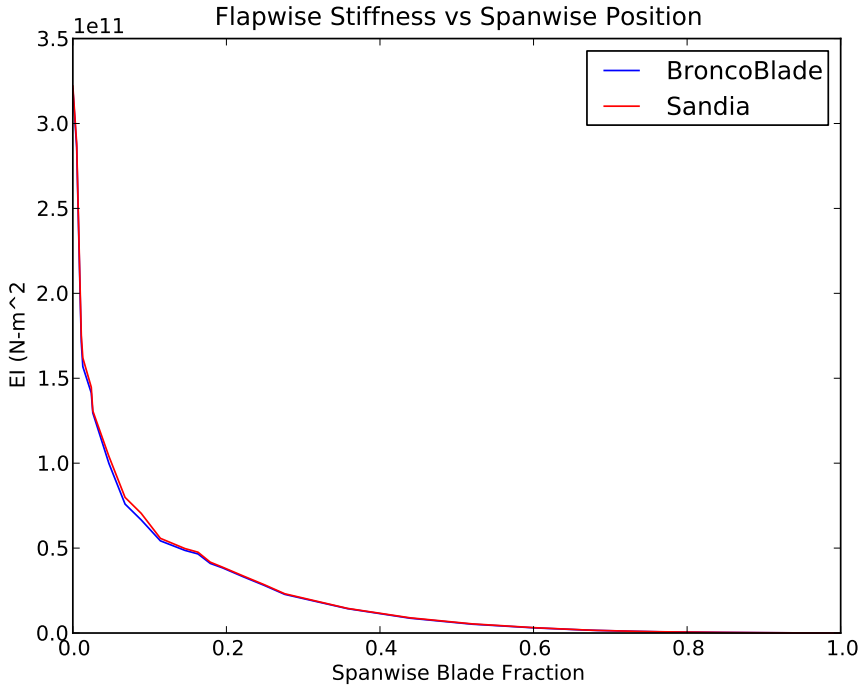


Figure 4.4: *Comparison of flapwise stiffness: Sandia and BroncoBlade*

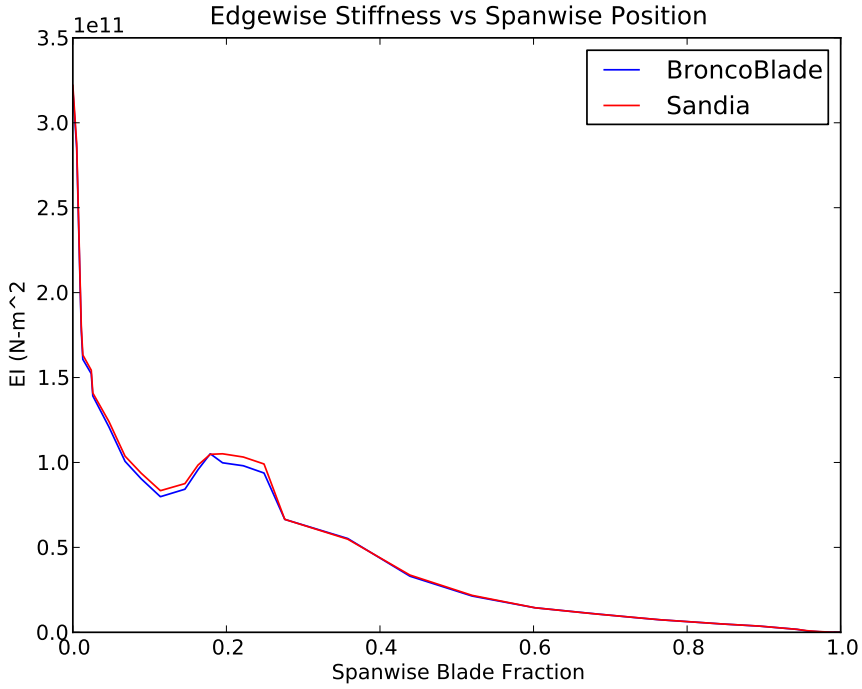


Figure 4.5: *Comparison of edgewise stiffness: Sandia and BroncoBlade*

$$0 + 0 \cdot x + C_2x^2 + C_3x^3 + C_4x^4 + C_5x^5 + C_6x^6 = 1 \tag{4.1}$$

Because of the cantilever beam nature of the blade, the coefficients C_0 and C_1 are forced to 0, and thus are omitted from Equation 4.1. A least-squares fit is used to determine the remaining four coefficients. Because different combinations of coefficients can produce similar polynomial shapes, only the shapes are shown here in Figures 4.6 and 4.7, and the coefficients are listed in blade data file in the Appendix.

The wind condition used in the validation simulates the turbine's performance in extreme winds where normal operating conditions have been exceeded and the turbine has shut down. Since the rotational velocity of the turbine affects the mode shapes, the parked blade's mode shapes must be evaluated separately from the operating mode shapes.

To property calculate the mode shapes of a rotating turbine blade, the rotational loads must be included in the analysis. Unfortunately, an elementary attempt to incorporate these loads into the frequency analysis model failed. Further work on modeling these forces was deemed to be outside of the scope of the project. Because of the similarity between the 0 RPM and 7.44 RPM mode shapes from the Sandia FAST models, shown in Figures 4.8 and 4.9, the parked blade's modes shapes will be used temporarily for any rotating simulations.

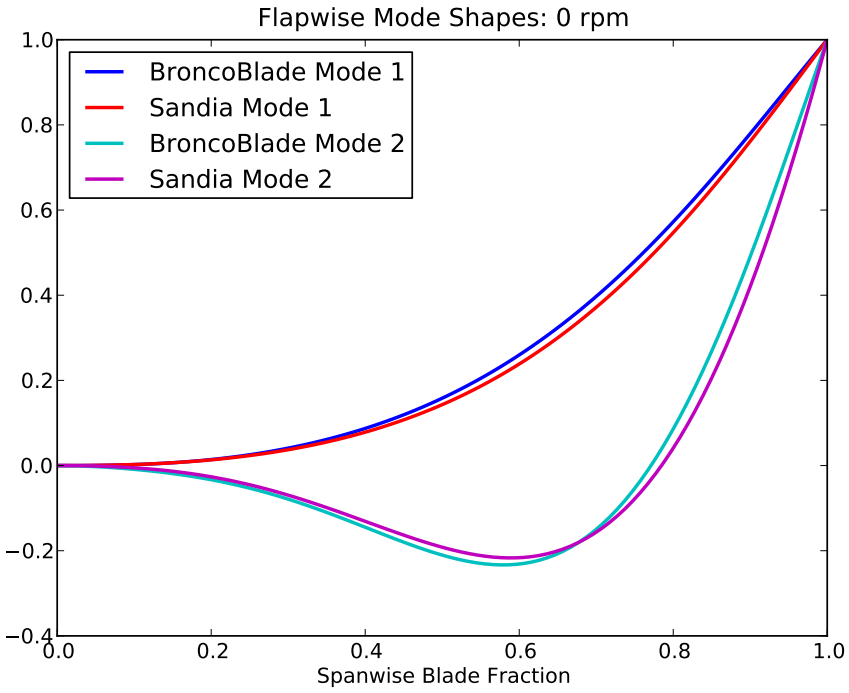


Figure 4.6: *Comparison of flapwise mode shapes: Sandia and BroncoBlade*

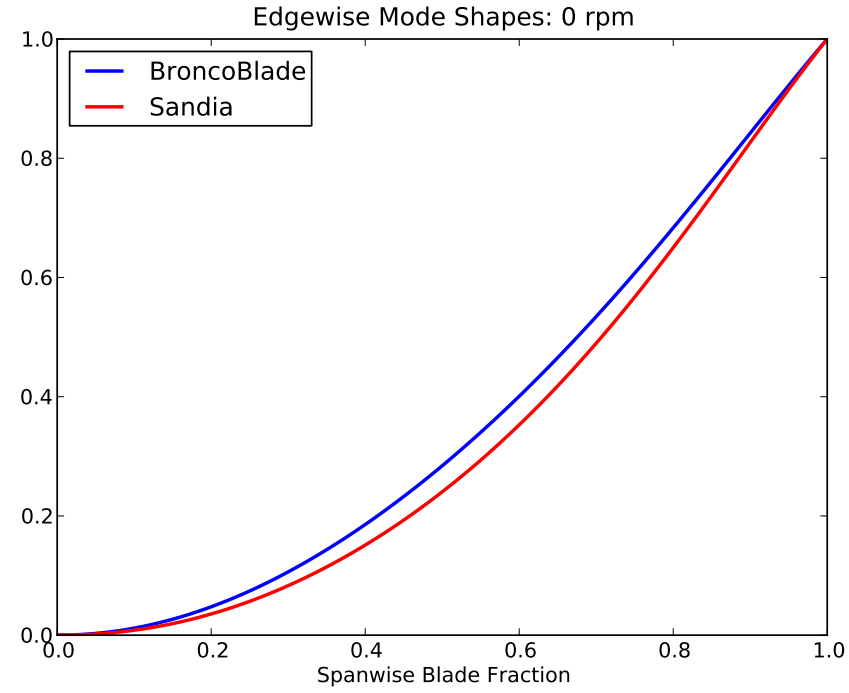


Figure 4.7: *Comparison of edgewise mode shapes: Sandia and BroncoBlade*

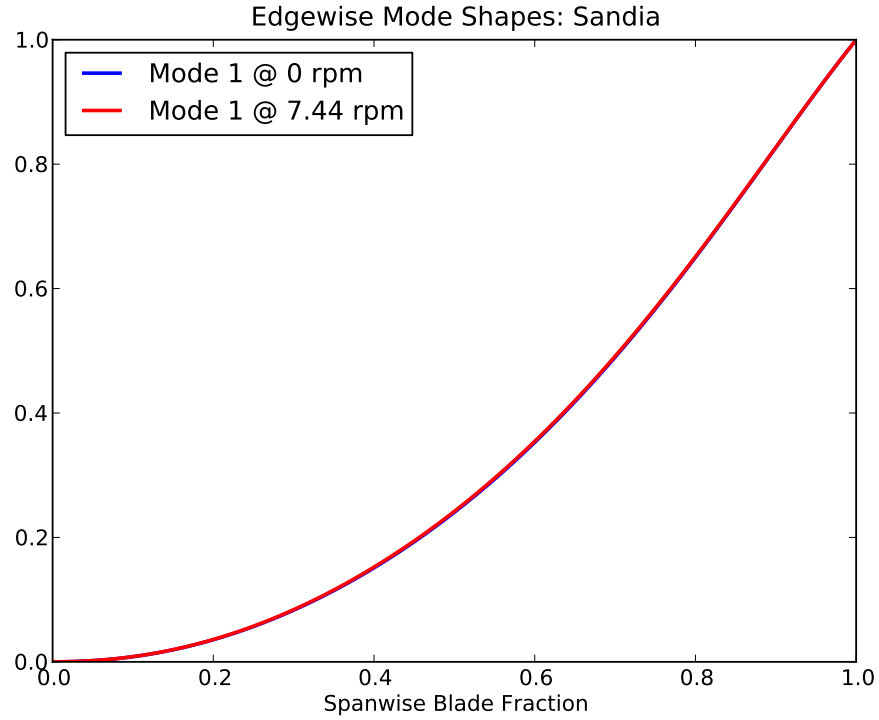


Figure 4.8: *Comparison of Sandia's flapwise mode shapes at 0 and 7.44 RPM*

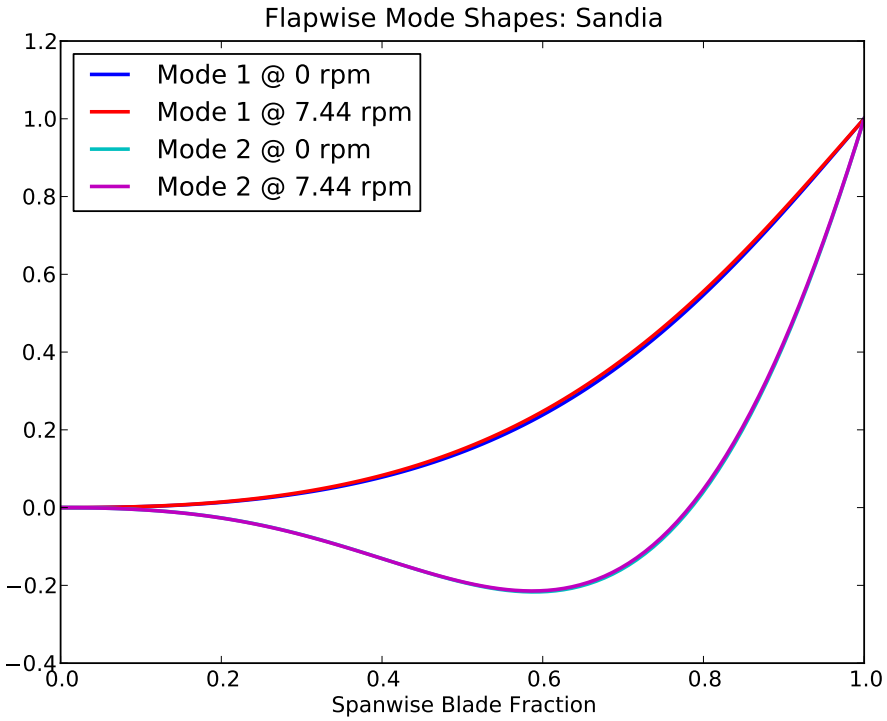


Figure 4.9: *Comparison of Sandia's edgewise mode shapes at 0 and 7.44 RPM*

4.3 FAST Results

In their report [7], Sandia identified eight wind loading conditions to be simulated. Out of these load conditions, the maximum flapwise root moment and maximum tip deflection occurred during EWM50, and maximum edgewise moment occurs during EDC-R. As the extremes, these two load conditions were the ones intended for validation, but because of complications with the pitch controller in the operation turbine, only the EWM50 condition was used.

The EWM50 load condition simulates a 50-year gust when the turbine has been shut down (RPM=0) and the ability to pitch the blades out of the wind has been lost [7]. The assumed wind loading starts with the reference speed of 50 m/s, allowing the simulation to escape the transient period. The wind speed then follows the sequence of dip-spike-dip shown in Figure 4.10. For the Class IB wind site, the wind dips from 50 m/s to 45 m/s, and then spikes to a maximum of 70 m/s [2]. The results in Figure 4.11 and 4.12 show reasonable correlation. The total tip deflection of 12.6m is 2.5% above Sandia's reported value of 12.3m. The maximum flapwise and edgewise root moments are 111,000 Nm and 43,700 Nm, respectively; these are 0.2% and -14% of the Sandia values of 110,700 Nm and 17,300 Nm.

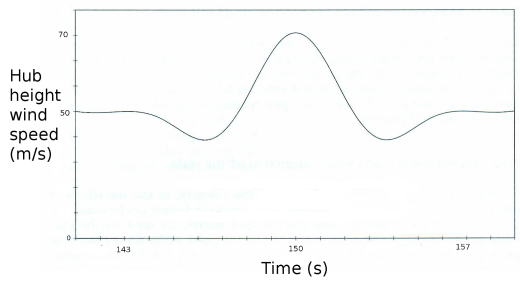


Figure 4.10: *Example of the gust profile used for the EWM50 wind condition [2]*

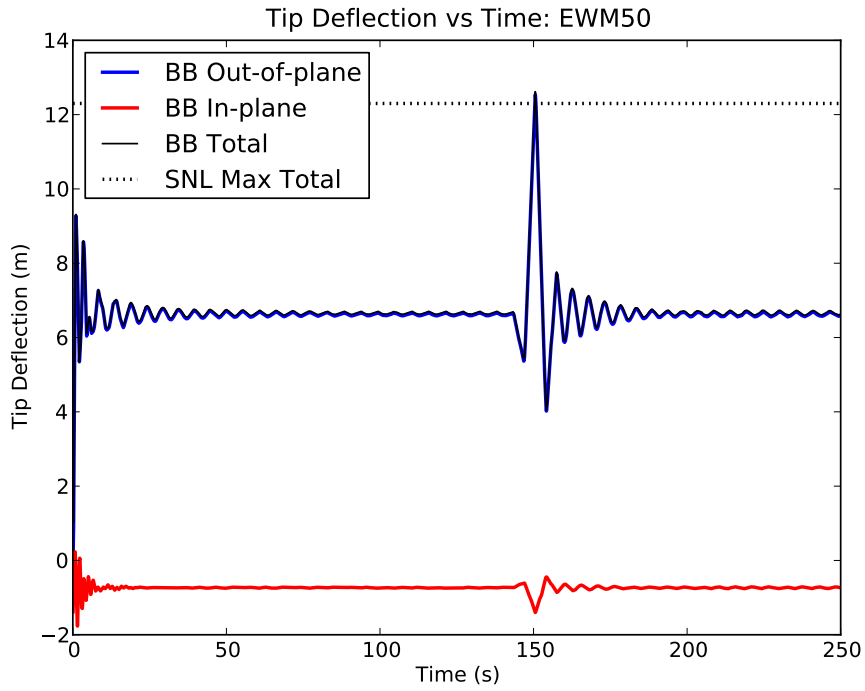


Figure 4.11: *Tip deflections of the BroncoBlade FAST simulation compared with reported Sandia maximum for the EWM50 wind condition*

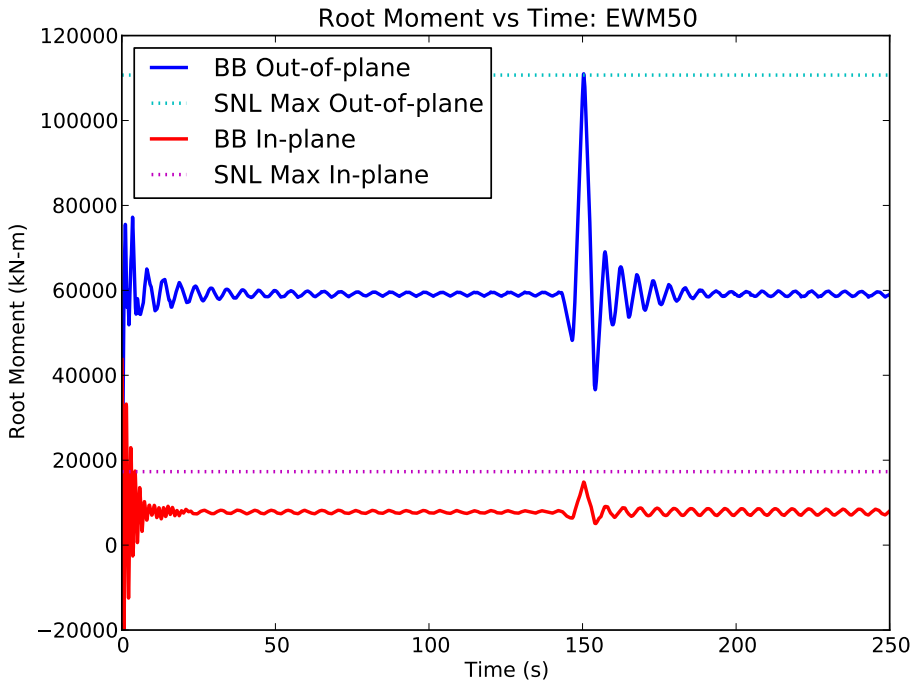


Figure 4.12: *Root moments of the BroncoBlade FAST simulation compared with reported Sandia maxima for the EWM50 wind condition*

4.4 Static Stress Analysis Results

Using the maximum flapwise aerodynamic forces and the gravitational forces for a blade pointed vertically upward, the static loading produces a tip deflection of only 10.8m, short of the 12.6m calculated in the FAST simulation. The simple explanation for this discrepancy is that the gusting wind causes a dynamic response in the blade that cannot be properly modeled by simply applying the static loads.

Sandia calculated strain results using Euler-Bernoulli beam theory and the root moments computed by FAST. While this is a valid method, BroncoBlade should ideally calculate strain results at an element level, rather than for the blade overall. This element level strain will allow for the use of failure criterion to be applied to the composite layup on that element. This leads to the conclusion that a dynamic or quasi-static analysis needs to be incorporated into future versions of BroncoBlade.

4.5 Buckling Analysis Results

Sandia describes their loading analysis as using only the flapwise aerodynamic loads, since the edgewise loads are relatively small in comparison (see the root moments in Figure 4.12). The BroncoBlade code includes both edgewise and flapwise loads. The buckling mode eigenvalues and locations are compared in Table 4.7 and illustrated in Figure 4.13.

What is apparent from Table 4.7 is that BroncoBlade produces higher eigenvalues. In this static buckling analysis where the design loads are applied, the eigenvalue is equivalent to the safety factor of the design with respect to buckling. Sandia cites an acceptable safety factor to be 2.042. Thus, the higher values of BroncoBlade are *less* conservative than those published by Sandia.

It is thought that the difference in buckling modes comes from the differ-

ent methods of load application. BroncoBlade reads the aerodynamic forces on each aerosection from the FAST output, and applies the load as a surface pressure distributed over the skin of that section. Sandia's method of loading uses a concentrated force at the center of each of the 18 aerosections. This artificially concentrates the forces over the geometry, which could be the reason that the Sandia model exhibits a lower buckling safety factor.

(a) 2.84 (b) 3.18 (c) 3.37

Figure 4.13: *First three unique mode shapes with their eigenvalues (scaled)*

4.6 Conclusion of Validation Process

BroncoBlade performs well in creating the blade data file input for FAST. The linear mass, flapwise stiffness, and edgewise stiffness all show strong correlation with the published Sandia results. The mode shapes depart a bit more from Sandia, but this could be related to the manual addition of the third shear web.

The outputs from FAST reasonably match Sandia's descriptions of maximum tip deflection and root moments. Some of the discrepancy could be attributed to differences in the wind loading, since Sandia provides the wind type but not specifics on the duration of the gust.

After applying the aerodynamic loads calculated in FAST to the blade, the results begin to differ. BroncoBlade produces consistently higher eigenvalues, meaning it is less conservative than the Sandia results. This discrepancy is thought

Table 4.7: Buckling Modes

Sandia Eigenvalue	Location	BroncoBlade Eigenvalue (order)	Location
2.173	10-15 meters spar cap	3.183 (2)	10-20 meters spar cap
		3.327 (6)	10-20 meters spar cap
2.183	19.5 meters max chord spar cap		
2.229	72-80 meters spar cap/ aft panel	2.837 (1)	74-83 meters spar cap/ aft panel
		3.37 (3)	72-82 meters spar cap/ aft panel
		3.585 (5)	72-82 meters spar cap/ aft panel
2.327	25-29 meters leading edge		
2.536	23-37 meters trailing edge		
2.589	19.5 meters max chord trailing edge		
		3.557 (4)	84-88 meters spar
		3.979 (7)	65-85 meters aft panel/ spar

to be caused by BroncoBlade's distributed loading method, opposed to Sandia's concentrated loads. Even with this, the buckling modes are similar.

Overall, BroncoBlade's performance compares favorably with the results published by Sandia. It is deemed to have sufficient quality to allow for its use in new design iterations.

Chapter 5

Turbine Design Iterations

Three design iterations, called AQ100-01, -02, and -03, were compared with the baseline SNL100-00 design for both the parked blade EWM50 wind condition, and the operating turbine ECD-R load condition. A buckling analysis was performed on the models using the maximum flapwise forces found in the EWM50 simulation. The iterations maintained the same skin and shear web thicknesses; changes were made to the material type or removed a shear web.

The masses of the models were calculated using ABAQUS and FAST, which showed slightly different values that are shown in Table 5.1. The results for the EWM50 and ECD-R simulations are compiled in Tables 5.2 and 5.3, respectively. The results of the three iterations were then used to suggest future design iterations.

Table 5.1: Mass of Models

Model	Mass from ABAQUS (kg)	Mass from FAST (kg)
SNL100-00	113,511	112,591
AQ100-01	113,108	112,213
AQ100-02	112,000	111,115
AQ100-03	97,714	97,906

Table 5.2: Results of EWM50 Wind Condition (Parked Blade)

Model	Maximum tip deflection (m)	Maximum root moment x (Nm)	Maximum root moment y (Nm)	Lowest Eigenvalue
SNL100-00	12.6	14860	111000	2.873
AQ100-01	12.2	14860	111100	3.02
AQ100-02	12.2	14850	111000	2.67
AQ100-03	3.3	15900	113600	n/a

Table 5.3: Results of ECD-R Wind Condition

Model	Max Flapwise tip deflection (m)	Max Edgewise tip deflection (m)	Max root moment x (Nm)	Max root moment y (Nm)
SNL100-00	7.08	0.576	44700	50990
AQ100-01	6.87	0.546	44560	51050
AQ100-02	6.88	0.550	44170	51010
AQ100-03	1.99	0.051	39300	51120

5.1 SNL100-00: Baseline Design

As described in the validation chapter, the SNL100-00 model uses triaxial fiberglass as the main material in the skin, with uniaxial fiberglass and foam used for reinforcement throughout the blade, and more triaxial fiberglass used to reinforce the the root of the blade. The shear webs are composed of foam sandwiched between to layers of biaxial fiberglass.

Under the EWM50 loading, the skin experienced buckling modes with maximum displacement at three different locations. The lowest eigenvalue was 2.873, with a maximum displacement on the spar cap around 77 m from the root, the second had an eigenvalue of 3.183 at a location around 13 m, and the third has an eigenvalue of 3.55 around 78m on the spar cap and aft panel.

The ECD-R loading was *not* conducted in the validation chapter because the pitch control input for the Sandia results was not available. A pitch file was generated for this chapter, so each blade is run with the same pitch control. The operating turbine incorporates cycling gravitational loads and centripetal loads

that were not present in the stationary EWM50 simulation. The inclusion of these loads better demonstrates the turbine's performance during typical operation.

5.2 AQ100-xx Design Iterations

The first design iteration, AQ100-01, simply changed the material of the 3 shear webs to carbon fiber, and is otherwise geometrically identical to the SNL100-00. The buckling modes exhibited were in the same locations as the SNL100-00 results, but with higher eigenvalues of 3.02, 3.13, and 3.7 respectively.

AQ100-02 kept the same design as AQ100-01, but removed the third shear web that supported the aft panel. The buckling modes that exhibited in the three shear web design did not show much change in eigenvalues. However, the lowest eigenvalue occurred at a new buckling modes that was introduced by the removal of the aft shear web. This mode had a value of 2.67, which is higher than the minimum 2.042 listed by Sandia [7], but shows still shows the potential buckling problems with the unreinforced aft panel. The lowest mode shape is shown in Figure 5.1.

Figure 5.1: *New buckling mode in AQ100-02*

The final iteration, AQ100-03, used the 2-spar configuration of AQ100-02, but replaced the fiberglass skin with carbon fiber. It used the same thickness and layup for the laminate skin and shear webs. It was shown to be an over-design in

terms of buckling since all of the first 17 calculated eigenvalues were negative. The final eigenvalue calculated was -5.77, which means that the first positive buckling mode of the model must be greater than 5.77.

While the AQ100-01 and AQ100-02 models showed slight reductions in tip deflection from the baseline, the all carbon design had tip deflection values less than 30% of the baseline model's. This shows that the switch to carbon fiber provides an unnecessary amount of added stiffness, and should have it's composite layup reduced to save material costs and avoid over-design. Even in it's current over designed state, the all carbon blade is the lightest of the models and can have its mass reduced even more with proper design of the skin and spar laminates.

5.3 Suggested Future Iterations

From the completed design iterations, it is clear that any optimal carbon fiber design lies between the AQ100-02 and AQ100-03 models. These two models share the two carbon fiber shear web design, but differ in the skin laminate. There is an array of designs that are worth investigating in the future.

Rather than replacing all of the fiberglass, the carbon fiber could replace only the spar cap reinforcement. In conjunction with the carbon shear webs, this would form a square carbon fiber tube running the length of the blade. This enclosed tube would add torsional rigidity and bending stiffness, but not carry the expense of using carbon fiber for all of the skin.

The over design of the AQ100-04 iteration can be seen by the reduction of tip deflection and by the high buckling safety factors. The general design of this iteration can be made more effective by reducing the amount of carbon fiber in the skin. To see how thick the skin needs to be, models could be iterated with varying thicknesses of unreinforced CFRP skin. The skin should be thin enough so that

there are some areas that require reinforcement, and thick enough so that some areas that can stay unreinforced. Using the unreinforced iterations to determine an appropriate basic skin laminate, reinforcement can be gradually added in to the areas that have the highest strain values or undergoing buckling.

A potential alternative to increasing the aft panel reinforcement or adding in a third shear web would be to include stringers on the interior surface of the skin. This could be more effective, since typically the aft panel skin is only in compression on the low-pressure surface.

All of the previous iterations have used uniaxial, biaxial, and triaxial composites. If a turbine simulation could be used that incorporates bend-twist and stretch-twist coupling, then the a custom laminate would be very useful for design. Using uniaxial plies stacked in angles other than 0°and 90°could cause these couplings to exist. These couplings could be designed to take advantage of aeroelastic coupling, like changing the angle of attack based on the bending curvature [15].

Chapter 6

Future Work

The end goal for BroncoBlade is to provide a competent and customizable interface, dependable results, and minimal resource consumption for the maximum amount of users. BroncoBlade version 0.1 is intended to be released as an open source software package. This will help promote the use of the program and allow for additional development. There are several improvements that can be made to the initial release of the program that will increase its functionality and efficiency.

6.1 Software Compatibility

BroncoBlade currently uses one commercial software package, which is the finite element solver ABAQUS. Ideally, BroncoBlade should be using an open source FE package, but it was decided to use a professionally documented and tested package during the creation and debugging of BroncoBlade. Now that it has been shown to be functional, BroncoBlade should be adapted to work with the open source program Calculix. Calculix shares a similar input file format with ABAQUS, so the amount of reconfiguring should be minimal.

Because BroncoBlade was developed using the Linux platform, the initial release has only been tested with Linux. To make BroncoBlade as inclusive as

possible, it should be ported to other platforms. The eventual hosting website should allow a potential user to download a compressed file that contains the source code and a directory with an example turbine model. The user should be able to unzip the file and have BroncoBlade operating in a minimal amount of time.

BroncoBlade uses several other open source packages, such as FAST, python, numpy, scipy, matplotlib, etc. BroncoBlade's website will have a list of the required packages and links to their hosting sites. This will help users who are unfamiliar with open source software or do not have the packages already installed on their computers.

6.2 Input Improvements

While the skin geometry input file is working well, there are several other model inputs that need to be upgraded to a more intuitive and user friendly interface.

Currently, material properties are changed by editing a python script file. While this method is functional, it should be replaced by a text file. The text file should first specify the material type, either isotropic or a laminate, then the following list of material properties would be read and interpreted internally in BroncoBlade.

The method of applying reinforcement to the skin is reasonably effective, utilizing an input file with thicknesses and locations. However, for the user to change the constant exterior skin laminate or the shear web laminate, they have to open up python scripts and find the line of code that writes out the laminate. Ideally, the user should not have to open any python files unless they are doing developmental work.

The chordwise boundaries of the mesh are set in the main input file by selecting

the position of the spars and the size of the reinforcement areas. The spanwise boundaries are set by the location of the stations. The method of setting these boundaries is acceptable, but a problem arises in specifying the mesh seeding between them.

Chordwise seeding requires a 5 parameter list to identify the seeding of each section; the vector defining the spanwise seeding has one less term than the number of stations, so it can easily have a length of over 30. Filling out these vectors requires the calculation of the number of nodes based on the desired element size and the size of the area to be meshed.

An improved meshing algorithm would allow for the specification of a preferred element size. BroncoBlade would then apply seeding so that the elements were as close to this size as possible. A single value could be used to govern the entire blade's mesh, or individual chordwise or spanwise sections could have assigned element sizes.

6.3 Module Improvements

Though the modules have proven functional, there is room for improvement in efficiency and accuracy.

6.3.1 Meshing

The meshing algorithm used in BroncoBlade works well for most of the blade, but falls short at the root and the tip. Addressing these problems would allow for more confidence to be put in the mesh.

Initially, the chordwise seeds were applied by seeding along the chord to define the x-values, and then applying the x-values to a polynomial curve fit of the top and bottom surfaces. This worked well for most of the airfoil, but not for

the curvature at the leading edge. To accommodate this curvature, the seeding was switched to the y-direction and a polynomial curve fit was performed for the x-values.

The current method works for all of the blade except for the root. Because the root is cylindrical, the trailing edge encounters the same problems that are mentioned above for the leading edge. Because the trailing edge at the root has a different shape than the rest of the blade, there is method used for the leading edge cannot be applied on the trailing edge.

What is needed is for the chordwise seeds to be applied such that the spacing is defined by the total distance between the nodes, not just the x or y component like it is now.

To use the spanwise spline method of meshing, there needs to be an equal number of nodes on each station. This works fine up until the tip of the blade, where the chord length starts become very small. Because there are the same number of elements around the maximum and minimum chords, the elements at the tip will be very small in comparison to those towards the root.

6.3.2 Stiffness and Mode Shape Calculation

The FEA based stiffness calculator currently used in BroncoBlade has been validated and shown to produce quality results. However, it requires the preparation, analysis, and post-processing of 30+ finite element models. Each station took around 3-5 minutes using 4 processors on a laptop computer. While the individual jobs had reasonable run times, the entire sections module took between 1 and 3 hours to complete. This limits the speed at which designs can be iterated and should ideally be much faster.

One alternative is to examine the cross section as a 2D structure and use integration techniques to calculate the EI value for the section. This method

would avoid the use of the FE solver and would be expected to be much faster. It is not used in the original BroncoBlade release because the time required to write and debug this program was predicted to be much greater than using pre-existing tools to create the FE mesh. There are existing programs that calculate cross section stiffness, but it is not known if they are detailed enough for BroncoBlade or if their interface would be compatible.

Similarly to the stiffness calculation, the mode shape calculation was functional, but non-ideal. The FE frequency analysis did not take very long to run, but for the SNL100-00 the analysis required on the order of 10 GB of RAM to run optimally. These frequency analyses had to be performed on a performance machine, since the author's computer was limited to 6 GB. While some potential users may have the computing power for this, it is expected that many will not have access to high performance computers. Some alternate method should be used; perhaps a method based on simplifying the blade into a beam model, using the stiffness values calculated in the Sections module.

6.4 Module Additions

Version 0.1 of BroncoBlade provides the tools to calculate a model's loading during operation and to evaluate its performance with respect to buckling. However, it lacks the ability to evaluate the modes of failure common in wind turbines.

To determine the strains that result from the simulation loading, a dynamic analysis will have to be performed on the FE model. The nature of the operating turbine takes it out of the static realm, so it can not be evaluated using only the applied forces but requires the addition of inertial loads. When the loading is properly applied, the FE model will yield strain values for all of the elements in the mesh. Using the strain values and the element's laminate information, a damage

or failure criterion could be applied. This would allow for the identification of the elements where failure could occur. Additionally, a fatigue failure module could be added. The results of FAST could be used to predict the loadings over the lifetime of the blade. Each level of loading could be associated with a specific amount of incurred damage. This would allow for the accumulated amount of damage to be calculated for each element. This would be too computationally expensive if every element was used, but a few critical elements could be chosen for evaluation.

Chapter 7

Conclusion

BroncoBlade, a tool for preparation and integration of a finite element model with a turbine simulation program, is reported. It contains four modules: the creation of a finite element model, the calculation of cross-sectional stiffness at locations along the blade, the calculation of mode shapes, and the application of loads calculated by the turbine simulator, FAST.

Analysis results for a model of the SNL100-00 baseline blade compared favorably against reference results published by Sandia National Laboratories. These results include stiffness properties of the blade, tip deflection and root moments from FAST using the EWM50 extreme speed wind condition, and the buckling behavior of the blade's skin.

Three design iterations were evaluated and compared with the all fiberglass SNL100-00 blade model. These designs used carbon fiber to reduce the mass of the blade while adding stiffness. Results from these iterations suggest designs for additional iterations.

Accompanying the initial release of BroncoBlade are a list of suggested enhancements to the program. These include improvements to the user interface, increased module efficiency, and the addition of more modules. With the source

code available, users are free to revise and contribute to the development of this software package.

Source code and further information on BroncoBlade can be obtained by contacting either Alex Quinlan (alex.r.quinlan@wmich.edu) or Dr. Peter Gustafson (peter.gustafson@wmich.edu).

Bibliography

[1] Marshall Buhl. Nwtc design codes: Iecwind, October 2012.

[2] Tony Burton, David Sharpe, Nick Jenkins, and Ervin Bossanyi. *Wind energy handbook*. Wiley, 2001.

[3] William D Callister and David G Rethwisch. *Materials science and engineering: an introduction*, volume 7. Wiley New York, 2007.

[4] Robert D Cook et al. *Concepts and applications of finite element analysis*. John Wiley & Sons, 2007.

[5] Isaac M Daniel, Ori Ishai, Issac M Daniel, and Ishai Daniel. *Engineering mechanics of composite materials*, volume 3. Oxford university press New York, 1994.

[6] Harvey M Deitel, Paul J Deitel, Ben Wiedermann, and Jonathan P Liperi. *Python with Cdrom*. Prentice Hall Professional Technical Reference, 2001.

[7] D Todd Griffith and Thomas D Ashwill. The sandia 100-meter all-glass baseline wind turbine blade: Snl100-00. *Sandia National Laboratories Technical Report, SAND2011-3779*, 2011.

[8] D Todd Griffith and Brian R Resor. Description of model data for snl100-00: The sandia 100-meter all-glass baseline wind turbine blade. *Sandia National Laboratories Technical Report, SAND2011-9309P*, 2011.

[9] Bob Griffiths. Boeing sets pace for composite usage in large civil aircraft. *High-Performance Composites*, 2005.

[10] D.M. Hoyt and D. Graesser. Rapid fea of wind turbine blades — summary of nse composites' structural analysis capabilities for blade nse blademesher in-house software. Technical report, NSE Composites, 2008.

[11] Jason M Jonkman and Marshall L Buhl Jr. Fast user's guide. *Golden, CO: National Renewable Energy Laboratory*, 2005.

[12] David J Laino and A Craig Hansen. User's guide to the wind turbine aerodynamics computer software aerodyn. *Windward Engineering, Salt Lake City, UT, December*, 2002.

[13] Daniel L Laird. *NuMAD User's Manual*. Sandia National Laboratories, 2001.

[14] Germanischer Lloyd. Rules and regulations, iv-industrial services, part 1-guideline for certification of wind turbines, 2010.

[15] James Locke and Ivan Contreras Hidalgo. The implementation of braided composite materials in the design of a bend-twist coupled blade. 2002.

[16] Patrick J Moriarty and A Craig Hansen. *AeroDyn theory manual*. National Renewable Energy Laboratory Colorado, 2005.

[17] DCS Simulia. Abaqus 6.11 analysis user's manual. *Abaqus 6.11 Documentation*, 2011.

Appendix A

BroncoBlade I/O File Examples

A.1 Input

SNL100.bld

Airfoil	Chord(m)	Twist(deg)	Pitch_Axis	Blade_Frac	t/c ratio
Cylinder	5.694	13.308	.5	0.000	1
Cylinder	5.694	13.308	.5	0.005	1
SNL100m0pt007	5.694	13.308	.5	0.007	.9925
SNL100m0pt009	5.694	13.308	.5	0.009	.985
SNL100m0pt011	5.694	13.308	.5	0.011	.9775
SNL100mEllipse97	5.694	13.308	.5	0.013	.97
SNL100mEllipse93pt1	5.792	13.308	.499	0.024	.931
SNL100mEllipse92pt5	5.811	13.308	.499	0.026	.925
SNL100mTransition84	6.058	13.308	.498	0.047	.840
SNL100mTransition76	6.304	13.308	.468	0.068	.760
SNL100mTransition68	6.551	13.308	.453	0.089	.68
SNL100mTransition60	6.835	13.308	.435	0.114	.6
SNL100mTransition51	7.215	13.308	.410	0.146	.51
SNL100mTransition47	7.404	13.177	.400	0.163	.47
SNL100mTransition43pt5	7.552	13.046	0.390	0.179	.435
DU99W405	7.628	12.915	0.380	0.195	.405
DU99W405_38	7.585	12.133	0.378		

SNL100.bld

```
0.222           .38
DU99W350_36                     7.488           11.350          0.377
0.249           .36
DU99W350_34                     7.347           10.568          0.375
0.276           .34
DU97W300                        6.923           9.166           0.375
0.358           .30
DU91W2250_26                    6.429           7.688           0.375
0.439           .26
DU93W210_23                     5.915           6.180           0.375
0.520           .23
DU93W210                        5.417           4.743           0.375
0.602           .21
NACA64618_19                    5.019           3.633           0.375
0.667           .19
NACA64618_18pt5                 4.920           3.383           0.375
0.683           .185
NACA64618                       4.621           2.735           0.375
0.732           .18
NACA64618                       4.422           2.348           0.375
0.764           .18
NACA64618                       3.925           1.380           0.375
0.846           .18
NACA64618                       3.619           0.799           0.375
0.894           .18
NACA64618                       2.824           0.280           0.375
0.943           .18
NACA64618                       2.375           0.210           0.375
0.957           .18
NACA64618                       1.836           0.140           0.375
0.972           .18
NACA64618                       1.208           0.070           0.375
0.986           .18
NACA64618                       0.100           0.000           0.375
1.000           .18
```

SNL100_00.in

```
dir     airfoils                # Airfoil Directory
sched   SNL100.bld              # Airfoil Schedule
length  100                     # Blade Length (m)
seeds   [24,28,24,34,9]         # Seeding for regions [A,B,C,D,E] or [LE, LP, Sp
ar, Aft P, TE]
spseed  [1,1,1,1,3,3,3,3,5,5,5,7,7,7,9,9,9,11,13,15,21,21,23,23,25,25,27,27,29,3
1,31,33,33]
                                # Spanwise Seed  (odd number)
number_aerosections  18
Asize   .02                     # Length of leading reinforcement region (m)
Esize   .5                      # Length of trailing reinforcement region  (m)
spar1   -.75                    # Chordwise location of spar from pitch axis (m)
spar2   .75                     # Chordwise location of spar from pitch axis (m)
SPstrt   2.4                    # Start of spars (m) from root
SPend   94.4                    # Termination of spars (m) from root
```

layup.txt

StaNum x	BlSpan x	RootBuild triax	SparCap uni	TEuni uni	TEfoam foam	LEpan foam	AFTpan foam
1	0.0	160	0	0	0	0	0
2	0.005	140	1	1	0	0	0
3	0.007	120	2	2	0	0	0
4	0.009	100	3	3	0	0	0
5	0.011	80	4	5	0	0	0
6	0.013	70	10	7	0	1	1
7	0.024	63	13	8	0	3.5	3.5
8	0.026	55	13	9	0	13	13
9	0.047	40	20	13	0	30	100
10	0.068	25	30	18	0	50	100
11	0.089	15	51	25	60	60	100
12	0.114	5	68	33	60	60	100
13	0.146	0	94	40	60	60	100
14	0.163	0	111	50	60	60	60
15	0.179	0	119	60	60	60	60
16	0.195	0	136	60	60	60	60
17	0.222	0	136	60	60	60	60
18	0.249	0	136	60	60	60	60
19	0.276	0	128	30	40	60	60
20	0.358	0	119	30	40	60	60
21	0.439	0	111	15	20	60	60
22	0.520	0	102	8	10	60	60
23	0.602	0	85	4	10	60	60
24	0.667	0	68	4	10	60	60
25	0.683	0	64	4	10	55	55
26	0.732	0	47	4	10	45	45
27	0.764	0	34	4	10	30	30
28	0.846	0	17	4	10	15	15
29	0.894	0	9	4	10	10	10
30	0.943	0	5	4	10	5	0
31	0.957	0	5	4	10	5	0
32	0.972	0	5	4	10	5	0
33	0.986	0	5	4	10	5	0
34	1.000	0	0	0	0	0	0

DU99W405.txt

```
1       0
1       −0.00347
0.9966  −0.00261
0.98235 0.0004
0.96706 0.00232
0.95072 0.00303
0.93333 0.00258
0.9149  0.00103
0.89542 −0.0016
0.8749  −0.00539
0.85333 −0.01045
0.83072 −0.01681
0.80706 −0.02452
0.78235 −0.03354
0.7566  −0.04382
0.73    −0.05519
0.70333 −0.06717
0.67667 −0.07958
0.65    −0.09228
0.62333 −0.10515
0.59667 −0.11804
0.57    −0.13083
0.54333 −0.14333
0.51667 −0.15537
0.49    −0.16673
0.46333 −0.17727
0.43667 −0.18683
0.41    −0.19527
0.38333 −0.20248
0.35667 −0.20827
0.33    −0.21243
0.30333 −0.21476
0.27667 −0.21501
0.25    −0.21297
0.22421 −0.2087
0.19982 −0.20257
0.17684 −0.19496
0.15526 −0.18616
0.13508 −0.17635
0.11631 −0.16567
0.09893 −0.15429
0.08297 −0.1423
0.0684  −0.12982
0.05524 −0.11698
0.04348 −0.10392
0.03312 −0.09076
0.02417 −0.07766
0.01662 −0.06446
0.01047 −0.05117
0.00572 −0.03783
0.00238 −0.02333
0.00044 −0.00905
0.00018 −0.00567
0       0
0.00124 0.01552
0.00388 0.03133
0.00792 0.04505
0.01337 0.0579
0.02022 0.0707
0.02847 0.08324
0.03812 0.09549
```

DU99W405.txt

```
0.04918 0.1074
0.06164 0.11884
0.07551 0.12972
0.09077 0.13998
0.10745 0.14953
0.12552 0.15831
0.14499 0.16625
0.16587 0.17323
0.18816 0.17916
0.21184 0.18388
0.23693 0.18729
0.26333 0.18924
0.29    0.18979
0.31667 0.18914
0.34333 0.18746
0.37    0.18491
0.39667 0.18154
0.42333 0.17739
0.45    0.17253
0.47667 0.16701
0.50333 0.16088
0.53    0.15417
0.55667 0.14698
0.58333 0.13938
0.61    0.13142
0.63667 0.12318
0.66333 0.11471
0.69    0.10605
0.71667 0.09724
0.74333 0.08832
0.76961 0.07947
0.79484 0.07094
0.81902 0.06275
0.84216 0.05493
0.86425 0.04754
0.88529 0.04056
0.90529 0.034
0.92425 0.02786
0.94216 0.02212
0.95902 0.0167
0.97484 0.01158
0.98961 0.00678
0.99314 0.00565
0.9966  0.00455
1       0.00347
```

A.2 Output

SNL100−Blade−00.dat

```
------------------------------------------------------------------------------
---------------------- FAST INDIVIDUAL BLADE FILE ------------------------------
1.5 MW baseline blade model properties from "InputData1.5A08V07adm.xls" (from C.
 Hansen) with bugs removed.
---------------------- BLADE PARAMETERS ----------------------------------------
  34        NBlInpSt    − Number of blade input stations (−)
False       CalcBMode   − Calculate blade mode shapes internally {T: ignore mode
 shapes from below, F: use mode shapes from below} [CURRENTLY IGNORED] (flag)
   3.882    BldFlDmp(1) − Blade flap mode #1 structural damping in percent of cr
itical (%)
   3.882    BldFlDmp(2) − Blade flap mode #2 structural damping in percent of cr
itical (%)
   5.900    BldEdDmp(1) − Blade edge mode #1 structural damping in percent of cr
itical (%)
---------------------- BLADE ADJUSTMENT FACTORS --------------------------------
   1.0      FlStTunr(1) − Blade flapwise modal stiffness tuner, 1st mode (−)
   1.0      FlStTunr(2) − Blade flapwise modal stiffness tuner, 2nd mode (−)
   1.0      AdjBlMs     − Factor to adjust blade mass density (−)
   1.0      AdjFlSt     − Factor to adjust blade flap stiffness (−)
   1.0      AdjEdSt     − Factor to adjust blade edge stiffness (−)
---------------------- DISTRIBUTED BLADE PROPERTIES ----------------------------
BlFract  AeroCent  StrcTwst  BMassDen  FlpStff      EdgStff      GJStff       EA
Stff  Alpha  FlpIner  EdgIner  PrecrvRef  PreswpRef  FlpcgOf  EdgcgOf  FlpEAOf
EdgEAOf
(−)      (−)       (deg)     (kg/m)    (Nm^2)       (Nm^2)       (Nm^2)       (N
)     (−)    (kg m)   (kg m)   (m)         (m)         (m)      (m)      (m)
(m)
0.0     0.25    13.308  5742.76987179   320293309389.0  319372302063.0  74.43384
47848   0       0       0       0       0       0       0       0       0
0
0.005   0.25    13.308  5095.46116965   286019378400.0  285168697105.0  74.43384
47889   0       0       0       0       0       0       0       0       0
0
0.007   0.25    13.308  4431.46454162   246921129476.0  249080151272.0  74.41894
80081   0       0       0       0       0       0       0       0       0
```

SNL100−Blade−00.dat

```
0
0.009   0.25    13.308  3772.35158061   208599746214.0  212853795639.0  74.40404
95653   0       0       0       0       0       0       0       0       0
0
0.011   0.25    13.308  3127.23040042   171345954978.0  177639525232.0  74.38915
3559    0       0       0       0       0       0       0       0       0
0
0.013   0.25    13.308  2844.46954689   156852345194.0  160721149885.0  74.37429
47932   0       0       0       0       0       0       0       0       0
0
0.024   0.25    13.308  2639.76813709   141367558279.0  152236892525.0  75.57574
2074    0       0       0       0       0       0       0       0       0
0
0.026   0.25    13.308  2682.66133196   129604657230.0  139188752417.0  75.81156
18375   0       0       0       0       0       0       0       0       0
0
0.047   0.25    13.308  2321.10463024   99829388877.2   120916755138.0  78.62645
35407   0       0       0       0       0       0       0       0       0
0
0.068   0.25    13.308  1943.91138959   75923186720.1   100665959889.0  81.71784
91857   0       0       0       0       0       0       0       0       0
0
0.089   0.25    13.308  1831.33163639   66601768726.8   90379813590.8   84.81087
24533   0       0       0       0       0       0       0       0       0
0
0.114   0.25    13.308  1620.01826628   54279506818.7   79836846124.0   88.38200
78148   0       0       0       0       0       0       0       0       0
0
0.146   0.25    13.308  1616.82509356   48647619378.3   84247063952.1   93.21976
41993   0       0       0       0       0       0       0       0       0
0
0.163   0.25    13.177  1738.67615478   46559683088.7   95733634458.2   95.79058
20564   0       0       0       0       0       0       0       0       0
0
0.179   0.25    13.046  1783.72657574   40966357101.0   105111221165.0  97.75060
```

SNL100−Blade−00.dat

```
94893   0       0       0       0       0       0       0       0       0
0
0.195   0.25    12.915  1796.84803356   38354322436.6   99799418842.6   98.92221
8698    0       0       0       0       0       0       0       0       0
0
0.222   0.25    12.133  1771.0131114    33094318991.6   98055468154.5   98.24977
17291   0       0       0       0       0       0       0       0       0
0
0.249   0.25    11.350  1745.52049279   28092572244.2   93774686429.9   98.09797
04597   0       0       0       0       0       0       0       0       0
0
0.276   0.25    10.568  1574.99156118   22752970561.4   66486126421.6   96.08934
5847    0       0       0       0       0       0       0       0       0
0
0.358   0.25    9.166   1443.5957244    14329276096.3   55230585564.6   90.17245
62302   0       0       0       0       0       0       0       0       0
0
0.439   0.25    7.688   1251.57802147   8732435314.43   33036584674.9   81.95871
68781   0       0       0       0       0       0       0       0       0
0
0.52    0.25    6.180   1103.32524091   5201111970.21   21345813111.1   73.93068
86526   0       0       0       0       0       0       0       0       0
0
0.602   0.25    4.743   935.497360162   3015967853.39   14432955918.7   67.86061
95141   0       0       0       0       0       0       0       0       0
0
0.667   0.25    3.633   780.044102411   1763264032.52   11476227574.9   61.73410
40033   0       0       0       0       0       0       0       0       0
0
0.683   0.25    3.383   741.513027439   1518017441.91   10767128710.6   60.58369
30668   0       0       0       0       0       0       0       0       0
0
0.732   0.25    2.735   607.313343432   991342617.2     8673770279.41   56.99370
57952   0       0       0       0       0       0       0       0       0
0
```

SNL100−Blade−00.dat

```
0.764   0.25    2.348   504.129669833   717762355.701   7390865225.39   54.56996
68914   0       0       0       0       0       0       0       0       0
0
0.846   0.25    1.380   359.208951592   346053289.357   4891933465.82   48.52280
76344   0       0       0       0       0       0       0       0       0
0
0.894   0.25    0.799   289.101269688   202003563.338   3643785418.11   44.80475
15521   0       0       0       0       0       0       0       0       0
0
0.943   0.25    0.280   212.387576133   79977867.6009   1750824270.59   35.16650
80603   0       0       0       0       0       0       0       0       0
0
0.957   0.25    0.210   126.567930947   28301407.6971   710938933.696   29.57022
01957   0       0       0       0       0       0       0       0       0
0
0.972   0.25    0.140   97.8436734749   12538819.906    327836477.308   22.85936
18867   0       0       0       0       0       0       0       0       0
0
0.986   0.25    0.070   64.3764457781   3242088.20298   92979715.3976   15.04042
86362   0       0       0       0       0       0       0       0       0
0
1.0     0.25    0.000   5.1221452       2023.59891257   47155.1000883   1.246275
59905   0       0       0       0       0       0       0       0       0
0
---------------------- BLADE MODE SHAPES -------------------------------------
0.0621412002672  BldFl1Sh(2) - Flap mode 1, coeff of x^2
1.94285689321    BldFl1Sh(3) -            , coeff of x^3
-3.17213960891   BldFl1Sh(4) -            , coeff of x^4
4.10515507636    BldFl1Sh(5) -            , coeff of x^5
-1.93801356093   BldFl1Sh(6) -            , coeff of x^6
-1.20226576502   BldFl2Sh(2) - Flap mode 2, coeff of x^2
5.0093940699     BldFl2Sh(3) -            , coeff of x^3
-22.1778418001   BldFl2Sh(4) -            , coeff of x^4
35.0436041897    BldFl2Sh(5) -            , coeff of x^5
-15.6728906946   BldFl2Sh(6) -            , coeff of x^6
```

SNL100−Blade−00.dat

```
1.14406900084    BldFl1Sh(2) - Edge mode 1, coeff of x^2
0.569475113272   BldFl1Sh(3) -            , coeff of x^3
-2.16181502454   BldFl1Sh(4) -            , coeff of x^4
2.52332522063    BldFl1Sh(5) -            , coeff of x^5
-1.07505431021   BldFl1Sh(6) -            , coeff of x^6
```

Appendix B

BroncoBlade Source Code

BroncoBlade.py

```
#!/usr/bin/python

#####################################################################
#-------------------------------------------------------------------
# This is the master file for creating the Abaqus Input Files
#-------------------------------------------------------------------
#####################################################################
import site
import sys
import os
from subprocess import *
import subprocess
from numpy import *
import pickle as pickle
sdir="/home/quinlan/thesis/gen_mesh/BroncoBladev0.0/"          # Source Directory's Global loc
ation
site.addsitedir(sdir)
from bladeclasses import *
set_printoptions(precision=10,suppress=True)                   # This limits the n
umber of decimal places printed
print " \n\n ====== BroncoBlade v0.0 ====== \n\n "

# parse arguments ----------------------
infile=str(sys.argv[-1])                                       # Reads input file
from command line (final entry)
if os.path.isfile(infile): print "Input file "+infile+" found. \n "
else: print "ERROR: Input file "+infile+" not found."

modelname=infile.split('.')[0]

optcode=list(sys.argv[1:-1])                                   # Reads options cod
e from command line
optlist=process_arguments(optcode)                             # Turn options code
 into a list of usable variables

execfile(sdir+"intro.py")                                     # reads in external f
iles for use in scripts; prints "Intro is done" upon completion

#====================================================================
#  ----- Blade Module -----------
#  Create the overall blade mesh
if 'B' in optlist['Blade']:
    print "Blade Module Started... \n "
#====================================================================
    execfile(sdir+"buildblade.py")                              # This script builds
the blade and creates the elements and nodes
    execfile(sdir+"printblade.py")                             # This creates the mes
h for the entire blade
    if 's' in optlist['Blade']:
        f=open("storedstations.dat","w")
        pickle.dump(Stations,f)
        f.close()
        g=open("asecAREA.dat","w")
        pickle.dump(asecAREA,g)
        g.close()
        print "\n Stations have been saved in 'storedstations.dat'\n"
    print "\n Blade Module Completed \n "
#====================================================================
# ------ Sections Module ---------
#  Calculate the structural properties for each station
if 'S' in optlist['Sections']:
```

BroncoBlade.py

```
    print "\n Sections Module Started... \n "
#=============================================================================
    if 'l' in optlist['Sections']:
        f=open("storedstations.dat","r")
        Stations=pickle.load(f)
        print "\n Stations have been loaded from 'storedstations.dat' \n "

    execfile(sdir+"buildstasections.py")                          # create stations for
both ends of each station
    num_stas=34                                                          # number of stat
ions to be used  !!! This should be read from somewhere
#    for job in [14]:                                                  # TROUBLE SHOOTI
NG: uncomment this to investigate a single section,
                                                                      # ... instead of
 having to cycle through all of them
    for job in (range(num_stas)):                                     # Start cycling
through sections
        print "----------\n \n Begin Station "+str(job)+"\n"  # Print current job bein
g worked on
        dirname="dir_Section"+str(job)
        # Making section directory for storing FE run files
        if os.path.isdir(dirname):
            print "Directory "+dirname+" exists and will be used for section file storage."
        else:
            os.system("mkdir "+dirname)
            print "Directory "+dirname+" has been created and will be used for section file storage."
        if os.path.isfile("Section"+str(job)+".inp"):
            os.system("mv Section"+str(job)+".* "+dirname)

        Csta=Stations['sta'+str(job)]                                # store Current S
tation in a local variable
        gz=float(sta_list[job][4])*100                                # Global z locat
ion of the station.  Used to determine how the station section is treated
        execfile(sdir+"staNodesElms.py")                            # Generates Nodes a
nd Elements for the station sectio
        execfile(sdir+"printstasections.py")                       # This creates abaqus
input files to extract the stiffness of each section of the blade
        execfile(sdir+"getEI.py")                                    # Retrieves displa
cement information and calculates stiffness

        print "chordwise = " + str(cwseed) + " spanwise = " + str(spanseed)
        print "Element 1 area = " +str(SkinElements[0].area)
        print "Element 1 aspect = " +str(SkinElements[0].lensw/SkinElements[0].lencw)

    if 's' in optlist['Sections']:
        f=open("storedstations.dat","w")
        pickle.dump(Stations,f)
        print "\n Stations have been saved in 'storedstations.dat'\n"
        g=open("asecAREA.dat","w")
        pickle.dump(asecAREA,g)
        g.close()

    print "\n Sections Module Completed. \n "

#    os.system("mv Section"+str(job)+"* "+dirname)
#    print "Files moved for Section " + str(job)
#subprocess.call(['mv Section* section-jobs'],shell=True)        # Clean up work
ing directory; store abaqus files in 'station-jobs'

#=============================================================================
# ------ Mode Module ----------
```

BroncoBlade.py

```
#  Get mode shapes and natural frequencies
if 'M' in optlist['Modes']:
    print "\n Modes Module Started... \n "
#=============================================================================
    if 'l' in optlist['Modes']:
        f=open("storedstations.dat","r")
        Stations=pickle.load(f)
        print "\n Stations have been loaded from 'storedstations.dat' \n "

    execfile(sdir+"printFreqTest.py")                              # print frequency an
alysis
    execfile(sdir+"getModes.py")
    execfile(sdir+"genFAST.py")

    print "\n Modes Module Completed. \n "

#=============================================================================
# ------ Loads Module ------------------
# Apply Aeroloads from FAST to the total Blade model
if 'L' in optlist['Loads']:
    print "\n Loads Module Started... \n "
#=============================================================================
    f=open("storedstations.dat","r")
    Stations=pickle.load(f)
    g=open("asecAREA.dat","r")
    asecAREA=pickle.load(g)
    print "\n Stations have been loaded from 'storedstations.dat' \n "

    execfile(sdir+"readelm.py")
    execfile(sdir+"write-fastloads.py")
    print "Static analysis input file written"
    execfile(sdir+"write-fastbuckle.py")
    print "Buckling analysis input file written"

    print "\n Loads Module Completed. \n "

#=============================================================================
print "\n All modules finished."
print "\n BroncoBlade is done."
```

intro.py

```
#----------------------------------------------------------------------


# Read input file from arguments, then open it and extract control values     (d
uplicates code in bar.py)
#----------------------------------------------------------------------


print infile
cf = open(infile)
Controls={}
rc=cf.readlines()
for ln in rc:
    lcont=ln.split()
    Controls[lcont[0]]=lcont[1]
# check values


print "Normalized Airfoil definitions are in the directory: " , Controls['dir']
print "The airfoil schedule file is: ", Controls['sched']
print "The total length of the blade is: ", Controls['length']
#----------------------------------------------------------------------


# Read in the Blade file


#----------------------------------------------------------------------


f = open(Controls['sched'])
# Separate strings into lists for each station


sta_list=[]                                 # initialize station list


read_parameters = f.readlines()
for ln in read_parameters[1:]:              # skips first line header


    sta=ln.split()
    sta_list.append(sta)
number_of_stations=len(sta_list)
#---------------------------------
# Initialize Stations
#---------------------------------
Stations = {}                               # initialize station dictionary (this
 might work better as a list, but that change would take some work)
for i in range(number_of_stations):
    stanum="sta"+str(i)
    Stations[stanum]=Station(i)             # create instances of stations
#---------------------------------
# Read seeding parameters
#---------------------------------
Asize=float(Controls['Asize'])


Esize=float(Controls['Esize'])
```

intro.py

```
spar1=float(Controls['spar1'])


spar2=float(Controls['spar2'])


seed=Controls['seeds'][1:-1].split(',')
Aseed=int(seed[0])

Bseed=int(seed[1])

Cseed=int(seed[2])

Dseed=int(seed[3])

Eseed=int(seed[4])

cwseed=(Aseed+Bseed+Cseed+Dseed+Eseed)*2     # total of each regional seed (x2 f
or top and bottom halves)
z_sparstart=float(Controls['SPstrt'])
z_sparend=float(Controls['SPend'])
# Read layup
if os.path.isfile("layup.txt"):
    print "Layup file 'layup.txt' found."
    Lay=Layup("layup.txt")
else: print "\n Warning: Layup file 'layup.txt' not found."

print " \n Intro is done \n "
```

bladeclasses.py

```
# This module contains classes for the blade stations
#
from numpy import *
from pylab import *
from scipy import interpolate
import warnings
warnings.simplefilter('ignore', np.RankWarning)
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

def xyzplot(x,y,z,xlab="X-vals",ylab="Y-vals",zlab="Z-lab"):
    fig = plt.figure()
    ax = Axes3D(fig)
    for c, m, in [('r', 'o'), ('b', '^')]:
        ax.scatter(x, y, z, c, marker=m)
        ax.set_xlabel(xlab)
        ax.set_ylabel(ylab)
        ax.set_zlabel(zlab)
    plt.show()

def process_arguments(optcode):
    optlist={'Blade':['off'],'Sections':['off'],'Modes':['off'],'Loads':['off']}
    for i in optcode:
        if list(i)[1] == 'B':
            optlist['Blade']=list(i)[1:]
            print "Blade module on"
        elif list(i)[1] == 'S':
            optlist['Sections']=list(i)[1:]
            print "Sections module on"
            if 'r' in list(i)[1:]: print " with FE analysis on"
            if 'p' in list(i)[1:]: print " with FE post-processing on"
        elif list(i)[1] == 'M':
            optlist['Modes']=list(i)[1:]
            if 'r' in list(i)[1:]: print " with FE analysis on"
            if 'p' in list(i)[1:]: print " with FE post-processing on"
        elif list(i)[1] == 'L':
            optlist['Loads']=list(i)[1:]
            idx=optcode.index('-L')+1                # Since the parser will rea
d the results.elm file as a different
            optlist['Loads'].append(optcode[idx])    # argument and not a sub-ar
gument, we have to append it here to
            print "FAST element load file = "+optcode[idx]
            optcode.pop(idx)                         # the list of 'Loads' sub-a
rguments, and then delete it from the argument list (optcode)
        else:
            print "Warning: "+str(list(i)[1])+" is not a valid argument. It will be ignored"
    return optlist

def spline(x,y,xrange,order=20):
    pf=polyfit(x,y,order)
    spy=polyval(pf,xrange)
    return spy

def splint(x,y,xrange):
    tck = interpolate.splrep(x,y,s=0)
    spy = interpolate.splev(xrange,tck,der=0)
    return spy

class Station:
    def __init__(self,jobnum):
        self.stanum=jobnum
```

bladeclasses.py

```
    def setfoil(self,airfoil,directory):
        self.foilname = airfoil
        foilfile = directory + "/" + airfoil + ".txt"
        self.normalized = genfromtxt(foilfile)
        self.x=self.normalized[:,0]
        self.y=self.normalized[:,1]
        self.nx=self.x   # these 2 are just for reference
        self.ny=self.y   #

    def plotfoil(self):
        plot(self.x,self.y)
        title(self.foilname)
        show()

    def top_bot(self):
        for position, item in enumerate(self.x):
            if item == self.t:
                self.half_t=position
            if item == self.h:
                self.half_h=position

        if self.x[0] < 0:
            self.half=self.half_t
        else:
            self.half=self.half_h
        self.h1_x=self.x[0:self.half+1]
        self.h1_y=self.y[0:self.half+1]
        self.h2_x=self.x[self.half:len(self.x)]
        self.h2_y=self.y[self.half:len(self.x)]
        self.h2_x=concatenate((self.h2_x,self.h1_x[0:1]))
        self.h2_y=concatenate((self.h2_y,self.h1_y[0:1]))

    def chordwise_seed(self,z,limz,Asize,Esize,spar1,spar2):
        self.h=min(self.x)
        self.t=max(self.x)
        self.hr=self.h+Asize
        self.tr=self.t-Esize
        self.sp1=spar1
        self.sp2=spar2
        self.crd=self.t-self.h
        [seedA,seedB,seedC,seedD,seedE]=self.seed

# These could be changed to some non-linear seeding function
# X-seeding for blade tip
        if z > limz:
            self.checker="tip"
            self.Lead=Lead=linspace(self.h,(self.h+self.crd*.10942),seedA+seedB+
1)
            self.Spar=Spar=linspace(self.h+self.crd*.10942,self.h+self.crd*.6405
8,seedC+1)
            self.Trail=Trail=linspace(self.h+self.crd*.64058,self.t,seedD+seedE)
            self.cws=concatenate([ Lead[:-1],Spar[:-1],Trail])
# X-seeding for main part of blade
        else:
            self.checker="root"
            self.A=A=linspace(self.h,self.hr,seedA+1)
            self.B=B=linspace(self.hr,self.sp1,seedB+1)
            self.C=C=linspace(self.sp1,self.sp2,seedC+1)
            self.D=D=linspace(self.sp2,self.tr,seedD+1)
            self.E=E=linspace(self.tr,self.t,seedE)
```

bladeclasses.py

```
            self.cws=concatenate([A[:-1],B[:-1],C[:-1],D[:-1],E])

    def spline(self,x,y,xrange,weights=1,order=10):
        w=ones(len(x))
        xmax=x.argmax()
        w[xmax]=100000000
        pf=polyfit(x,y,order)
        spy=polyval(pf,xrange)
        return spy

    def combine_foil(self,y1,y2,lead_y,lead_x,N):
        if y2[N]/lead_y[0] < 0:   # make sure that x and y are continuous
            ylead=lead_y[::-1]
            xlead=lead_x[::-1]
        else:
            ylead=lead_y
            xlead=lead_x
        self.y=concatenate(( y1[N:][:-2],       array([(y1[-2]+y2[-1])/2]),
        y2[N:][::-1]      , ylead ))
        self.x=concatenate(( self.cws[N:][:-2], array([(self.cws[-2]+self.cws[-1
])/2]), self.cws[N:][::-1], xlead ))

    def resize(self,scale):
        self.x=self.x*scale
        self.y=self.y*scale

    def xshift(self,meters):
        self.x=self.x-(min(self.x) + meters)

    def yshift(self,offset=0):
        x_min_index=self.x.argmin()
        self.y_at_xmin=self.y[x_min_index]
        self.y=self.y-self.y_at_xmin+offset

    def set_tcr(self,tcr):
        chrd=max(self.x)-min(self.x)
        thck=max(self.y)-min(self.y)
        tc=thck/chrd
        adjust=tcr/tc
        self.y=self.y*adjust

    def rotate(self,twist):
        theta=radians(twist)
        self.cx=self.x
        self.cy=self.y
        self.rx=cos(theta)*self.cx-sin(theta)*self.cy
        self.ry=sin(theta)*self.cx+cos(theta)*self.cy
        self.x=self.rx
        self.y=self.ry

    def zval(self,z):
        self.z=z

    def coords(self):
        self.xyz=column_stack((self.x,self.y,ones(len(self.x))*self.z))

    def graphdata(self):
        self.thickness=max(self.y)-min(self.y)
        self.local_le=min(self.x)  # local implies neglecting the twist of the s
tation in global coordinates
        self.local_te=max(self.x)
```

bladeclasses.py

```
        self.chord=self.local_te-self.local_le

class SpanSpline:
    def __init__(self,node,Stations):
        self.num = node
        self.sx=[]               # Station X values
        self.sy=[]
        self.sz=[]
        self.zzz=Stations['sta1'].z
        for i in range(len(Stations)):
            sn="sta"+str(i)
            self.sx.append(Stations[sn].xyz[self.num][0])
            self.sy.append(Stations[sn].xyz[self.num][1])
            self.sz.append(Stations[sn].xyz[self.num][2])

    def smooth(self,sta_rtsplit,z_rtsplit):
        dividers=self.spansplit
        seeds=self.splitseed
        self.z=[]

        if self.spansplit > 0:
            for i in range(len(seeds)):     # distribution of 'z' over entire bl
ade (root and tip)
                self.z=concatenate(  (self.z[:-1], linspace(dividers[i],dividers
[i+1],seeds[i])     ))
            for i in range(len(self.z)):         # Find index of root/tip transi
sition (end of spar)
                if self.z[i] == z_rtsplit:
                    index_sparend=i
            # seed division
            self.zr=self.z[:index_sparend+1]  # root z seeds (includes transitio
n point)
            self.zt=self.z[index_sparend:]     # tip z seeds  (includes transiti
on point)
            # value division
            self.szr=self.sz[:sta_rtsplit+1] # root z-vals
            self.szt=self.sz[sta_rtsplit:]   # tip z-vals
            self.sxr=self.sx[:sta_rtsplit+1] # root x-vals
            self.sxt=self.sx[sta_rtsplit:]   # tip x-vals
            self.syr=self.sy[:sta_rtsplit+1] # root y-vals
            self.syt=self.sy[sta_rtsplit:]   # tip y-vals

            self.xr=splint(self.szr,self.sxr,self.zr)
            self.xt=splint(self.szt,self.sxt,self.zt)
            self.x=concatenate((self.xr,self.xt[1:]))
            self.yr=splint(self.szr,self.syr,self.zr)
            self.yt=splint(self.szt,self.syt,self.zt)
            self.y=concatenate((self.yr,self.yt[1:]))

            self.xyz=hstack( (reshape(self.x,(-1,1)), reshape(self.y,(-1,1)), re
shape(self.z,(-1,1)) ) )

        else:
            self.x=linspace(self.sx[0],self.sx[1],self.splitseed)
            self.y=linspace(self.sy[0],self.sy[1],self.splitseed)
            self.z=linspace(0,self.zzz,self.splitseed)
            self.xyz=hstack( (reshape(self.x,(-1,1)), reshape(self.y,(-1,1)), re
shape(self.z,(-1,1)) ) )

class Spar:
    def __init__(self,spline_top,spline_bot,sparnodes,start, end):
```

bladeclasses.py

```
        self.top=spline_top
        self.bot=spline_bot
        self.nn=sparnodes                    # Nodes through spar thickness (from
 top surface to bottom surface)
        self.z_start=start
        self.z_end=end

        for i in range(len(self.top)):
            if self.top[i][2] ≤ start:
                startloc=i
            if self.top[i][2] ≤ end:
                endloc=i
        self.i_start=startloc
        self.i_end=endloc

    def interp(self,full='full'):
        self.sparx=[]
        self.spary=[]
        self.sparz=[]
        if full ≡ 'full':
            for i in range(len(self.top))[self.i_start:self.i_end]:
                self.sparx.append(linspace(self.bot[i][0],self.top[i][0],self.nn
))
                self.spary.append(linspace(self.bot[i][1],self.top[i][1],self.nn
))
                self.sparz.append(ones(self.nn)*self.top[i][2])
        if full ≡ 'stasec':
            for i in range(len(self.top)):
                self.sparx.append(linspace(self.bot[i][0],self.top[i][0],self.nn
))
                self.spary.append(linspace(self.bot[i][1],self.top[i][1],self.nn
))
                self.sparz.append(ones(self.nn)*self.top[i][2])
        self.xyz=hstack( (reshape(self.sparx,(-1,1)), reshape(self.spary,(-1,1))
, reshape(self.sparz,(-1,1)) ) )

class Node:
    def __init__(self,num,spline_num,span_num,x,y,z):
        self.num=num
        self.spline=spline_num
        self.span=span_num
        self.x=x
        self.y=y
        self.z=z
        self.output=str(self.num)+","+str(self.x)+","+str(self.y)+","+str(self.z
)

class Element:   # 8 noded shell element
    def __init__ (self,num,corner1,corner2,corner3,corner4,side5,side6,side7,sid
e8,elx,ely,elz,lencw,lensw):
        self.num=num
        self.n1=corner1
        self.n2=corner2
        self.n3=corner3
        self.n4=corner4
        self.n5=side5
        self.n6=side6
        self.n7=side7
        self.n8=side8
        self.lencw=lencw
        self.lensw=lensw
```

bladeclasses.py

```
        self.area=lencw*lensw
        self.nodes=[self.n1,self.n2,self.n3,self.n4,self.n5,self.n6,self.n7,self
.n8]
        self.output=str(self.num)+", "+str(self.nodes)[1:][:-1]
        self.elcen=[elx,ely,elz]

    def base_sort_cw(self,tex):
        if self.elcen[0] < -0.75:
            cw=0
        if -0.75 < self.elcen[0] ∧ self.elcen[0] < 0.75:
            cw=1
        if 0.75 < self.elcen[0] ∧ self.elcen[0] < (tex-1.0):
            cw=2
        if (tex-1.0) < self.elcen[0]:
            cw=3
        self.cw=cw

    def sort_sw(self,spansplit,aerob):
        sw=-1
        for line in spansplit:
            if self.elcen[2] > line:
                sw+=1
        self.sw=sw
        asnum=-1
        for line in aerob:
            if self.elcen[2] > line:
                asnum+=1
        self.asnum=asnum

class Layup:
    def __init__ (self,layfile='layup.txt'):
        matfile=open(layfile)
        mf=matfile.readlines()
        Lay=[]
        for ln in mf:
            stan=ln.split()
            Lay.append(stan)
        self.lay=Lay
    def get_layer(self,m,n):
        sta=n+2

# reinforcement
        rein_t2=0
        rein_m=0
        if m==0:
            rein_t1=float(self.lay[sta][6])
            rein_m1='foam'
            label="LE-panel"
        if m==1:
            rein_t1=float(self.lay[sta][3])
            rein_m1='uniax'
            label="SparCap-reinf"
        if m==2:
            rein_t1=float(self.lay[sta][7])
            rein_m1='foam'
            label="Aft-panel"
        if m==3:
            rein_t1=float(self.lay[sta][4])
            rein_t2=float(self.lay[sta][5])
            label="TE-reinf"
            rein_m1='uniax'
```

bladeclasses.py

```
            rein_m2='foam'

        if rein_t1 == 0:
            rein1="** delete this line "
        else:
            rein1=( str(rein_t1/1000) +', 3, '+ rein_m1 +', 90, ' + label)

        if rein_t2 == 0:
            rein2="** delete this line "
        else:
            rein2= (str(rein_t2/1000) + ', 3, foam, 90, TE-foam')
# root build-up
        root_t=float(self.lay[sta][2])

        if root_t == 0:
            root="** no root build-up"
        else:
            root=(str(root_t/1000) +', 3, triax, 90, root-buildup ')
        return [root,rein1,rein2]

class Section:
    def __init__ (self,nodes):
        self.nodes=nodes
```

buildblade.py

```
# Build the Blade
#-----------------------
stationzvals=[]
for i in range(number_of_stations):
    stanum="sta"+str(i)
    Stations[stanum].setfoil(sta_list[i][0],Controls['dir'])  # get airfoil
    z=float(sta_list[i][4])*float(Controls['length'])
    stationzvals.append(z)
#-------------------------------------------------
# Seed stations  chordwise (create spline)
#-------------------------------------------------
# Shift leading edge into position
    Stations[stanum].xshift(float(sta_list[i][3]))
# Adjust thickness to chord ratio
    Stations[stanum].set_tcr(float(sta_list[i][5]))
# Apply Chord Length
    Stations[stanum].resize(float(sta_list[i][1]))
    Stations[stanum].seed=[Aseed,Bseed,Cseed,Dseed,Eseed]
    Stations[stanum].chordwise_seed(z,z_sparend,Asize,Esize,spar1,spar2)
    Stations[stanum].oldx =Stations[stanum].x
    Stations[stanum].oldy =Stations[stanum].y
# Split the normalized foil into top and bottom
    Stations[stanum].top_bot()
# Spline 'y' values for top and bottom
    Stations[stanum].y1=y1=Stations[stanum].spline(Stations[stanum].h1_x,Station
s[stanum].h1_y,Stations[stanum].cws)
    Stations[stanum].y2=y2=Stations[stanum].spline(Stations[stanum].h2_x,Station
s[stanum].h2_y,Stations[stanum].cws)
# Smooth out leading edge
    N=Aseed+Bseed
    Stations[stanum].lead_x = lead_x = concatenate((Stations[stanum].cws[:N][::-
1],Stations[stanum].cws[:N]))
    Stations[stanum].lead_y = lead_y = concatenate((y1[:N][::-1],y2[:N]))
    Stations[stanum].lead_seed = lead_seed = linspace(min(lead_y),max(lead_y),le
n(lead_y)+1)
    Stations[stanum].new_x = new_x = spline(lead_y,lead_x,lead_seed)  # seed the
 leading edge with an odd number of nodes
# Recombine top, bottom, and leading edge
    Stations[stanum].combine_foil(y1, y2, lead_seed, new_x, N)
# Shift the average 'y' value
    Stations[stanum].yshift()
# Calculate values for graphs
    Stations[stanum].graphdata()
# Rotate
    Stations[stanum].rotate(float(sta_list[i][2]))
# Bring in the spanwise location, 'z'
    Stations[stanum].zval( z )
# Generate a printable summary
    Stations[stanum].coords()
# Find root/tip transition station
qqq=[]
for s in range(len(Stations)):
    qqq.append(abs(Stations['sta'+str(s)].z-z_sparend))
minq=min(qqq)
for s in range(len(Stations)):
    if qqq[s] == minq:
        sta_rtsplit=s

z_rtsplit=Stations['sta'+str(sta_rtsplit)].z
#-----------------------------------------------
# Create Spanwise Splines  *****FIX ME FIX ME FIX ME*****
```

buildblade.py

```
#-------------------------------------------------
num_spsp=len(Stations['sta0'].xyz)       # number of spanwise splines
SpSpline={}
Spxyz=[]
spansplit=list(stationzvals)
splitseed=[5.0, 3.0, 3.0, 3.0, 3.0, 9.0, 5.0, 15.0, 15.0, 15.0, 19.0, 23.0, 15.0
, 15.0, 15.0, 23.0, 23.0, 25.0, 69.0, 71.0, 73.0, 75.0, 61.0, 17.0, 49.0, 35.0,
83.0, 53.0, 53.0,  17.0, 19.0, 19.0, 19.0]

for i in range(num_spsp):
    spnum="spsp"+str(i)
    SpSpline[spnum]=SpanSpline(i,Stations)
    SpSpline[spnum].splitseed=splitseed
    SpSpline[spnum].spansplit=spansplit
    SpSpline[spnum].smooth(sta_rtsplit,z_rtsplit)
    Spxyz.append(SpSpline[spnum].xyz)
spanseed=len(SpSpline['spsp0'].z)  # there's probably a better way to do this
#-------------------------------------------------
# Create Spars
#-------------------------------------------------
sparmesh=21
spar1_top=0
spar1_bot=2*(Cseed+Dseed+Eseed)
spar2_top=Cseed
spar2_bot=Cseed+2*(Dseed+Eseed)
te_top=Cseed+Dseed
te_bot=Cseed+Dseed+2*Eseed
L_edge=cwseed-Aseed-Bseed
T_edge=Cseed+Dseed+Eseed
Spar1 = Spar(SpSpline['spsp'+str(spar1_top)].xyz,SpSpline['spsp'+str(spar1_bot)].x
yz,sparmesh,z_sparstart,z_sparend)
Spar1.interp()
Spar2 = Spar(SpSpline['spsp'+str(spar2_top)].xyz,SpSpline['spsp'+str(spar2_bot)].x
yz,sparmesh,z_sparstart,z_sparend)
Spar2.interp()
#-------------------------------------------------
# Create Skin Nodes
#-------------------------------------------------
Nodes=[]
SkinNodes=[]  # 2-D matrix of nodes
nn=1
for i in range(cwseed):
    tmp=[]
    for j in range(spanseed):
        tn=Node(nn,i,j,SpSpline["spsp"+str(i)].xyz[j][0],SpSpline["spsp"+str(i)].x
yz[j][1],SpSpline["spsp"+str(i)].xyz[j][2])
        tmp.append(tn)
        nn+=1
    SkinNodes.append(tmp)
#-------------------------------------------------
# Create Spar Nodes
#-------------------------------------------------
SparNodes=[]
SparNodes1=[]
SparNodes2=[]
ss=0
for i in range(spanseed)[Spar1.i_start:Spar1.i_end]:
    tmp=[]
    for j in range(sparmesh):
        tn=Node(nn,i,j,Spar1.xyz[ss][0],Spar1.xyz[ss][1],Spar1.xyz[ss][2])
        if j == 0:                      # Top and bottom of the spar is tied to
```

buildblade.py

```
the skin nodes along that spline

            tn=SkinNodes[spar1_bot][i]
        elif j ≡ (sparmesh−1):
            tn=SkinNodes[spar1_top][i]
        else:
            nn+=1
        ss+=1
        tmp.append(tn)
    SparNodes.append(tmp)
    SparNodes1.append(tmp)

ss=0
for i in range(spanseed)[Spar2.i_start:Spar2.i_end]:
    tmp=[]
    for j in range(sparmesh):
        tn=Node(nn,i,j,Spar2.xyz[ss][0],Spar2.xyz[ss][1],Spar2.xyz[ss][2])
        if j ≡ 0:                        # Top and bottom of the spar is tied to
the skin nodes along that spline
            tn=SkinNodes[spar2_bot][i]
        elif j ≡ (sparmesh−1):
            tn=SkinNodes[spar2_top][i]
        else:
            nn+=1
        ss+=1
        tmp.append(tn)
    SparNodes.append(tmp)
    SparNodes2.append(tmp)
# Create Node list (instead of matrix)
Nodes=[]
for i in range(len(SkinNodes)):
    for j in range(len(SkinNodes[0])):
            Nodes.append(SkinNodes[i][j])
for i in range(len(SparNodes)):
    for j in range(len(SparNodes[0])):
            Nodes.append(SparNodes[i][j])
#−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
# Create Elements
#−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
SkinElements=[]
elemnum=1
# Skin Elements
spar_end=round(float(Controls['SPend'])/float(Controls['length'])*spanseed−2)   #
end spar length / total length * seed − 2
cutoff=round(90/float(Controls['length'])*spanseed−2)   # desired cut off of 90m
/ total length * seed − 2
span=range(spanseed)
chrd=range(cwseed)
chrd.append(0)

for a in range(cwseed/2):
    for b in range(int(spanseed/2)):
        m=a*2
        n=b*2
        # Node Numbers for each of the 8 node positions (4 corners + 4 sides)
        c1=SkinNodes[chrd[m]][span[n]]
        c2=SkinNodes[chrd[m+2]][span[n]]
        c3=SkinNodes[chrd[m+2]][span[n+2]]
        c4=SkinNodes[chrd[m]][span[n+2]]
        s5=SkinNodes[chrd[m+1]][span[n]]
        s6=SkinNodes[chrd[m+2]][span[n+1]]
```

buildblade.py

```
        s7=SkinNodes[chrd[m+1]][span[n+2]]
        s8=SkinNodes[chrd[m]][span[n+1]]
        # area and side lengths
        elencw=sqrt((c2.x-c1.x)**2+(c2.y-c1.y)**2+(c2.z-c1.z)**2)  # element len
gth in chordwise direction
        elensw=sqrt((c4.x-c1.x)**2+(c4.y-c1.y)**2+(c4.z-c1.z)**2)  # element len
gth in spanwise direction
        # Locations of Element Center
        elx=(c1.x + c3.x) *0.5
        ely=(c1.y + c3.y) *0.5
        elz=(c1.z + c3.z) *0.5
        # Create Element instance
        el=Element(elemnum,c1.num,c2.num,c3.num,c4.num,s5.num,s6.num,s7.num,s8.n
um,elx,ely,elz,elencw,elensw)
        elemnum+=1
        SkinElements.append(el)

# Spar Elements  --- Check on nodal ordering!!!!!!!!!!!!!
SparElements=[]
for i in range(len(SparNodes1)/2):
    for j in range(len(SparNodes1[0])/2):
        m=2*i
        n=2*j
        c1=SparNodes1[m][n]
        c2=SparNodes1[m+2][n]
        c3=SparNodes1[m+2][n+2]
        c4=SparNodes1[m][n+2]
        s5=SparNodes1[m+1][n]
        s6=SparNodes1[m+2][n+1]
        s7=SparNodes1[m+1][n+2]
        s8=SparNodes1[m][n+1]
        # area and side lengths
        elencw=sqrt((c2.x-c1.x)**2+(c2.y-c1.y)**2+(c2.z-c1.z)**2)  # element len
gth in chordwise direction
        elensw=sqrt((c4.x-c1.x)**2+(c4.y-c1.y)**2+(c4.z-c1.z)**2)  # element len
gth in spanwise direction
        # Locations of Element Center
        elx=(c1.x + c3.x) *0.5
        ely=(c1.y + c3.y) *0.5
        elz=(c1.z + c3.z) *0.5
        el=Element(elemnum,c1.num,c2.num,c3.num,c4.num,s5.num,s6.num,s7.num,s8.n
um,elx,ely,elz,elencw,elensw)
        elemnum+=1
        SparElements.append(el)
for i in range(len(SparNodes2)/2):
    for j in range(len(SparNodes2[0])/2):
        m=2*i
        n=2*j
        c1=SparNodes2[m][n]
        c2=SparNodes2[m+2][n]
        c3=SparNodes2[m+2][n+2]
        c4=SparNodes2[m][n+2]
        s5=SparNodes2[m+1][n]
        s6=SparNodes2[m+2][n+1]
        s7=SparNodes2[m+1][n+2]
        s8=SparNodes2[m][n+1]
        # area and side lengths
        elencw=sqrt((c2.x-c1.x)**2+(c2.y-c1.y)**2+(c2.z-c1.z)**2)  # element len
gth in chordwise direction
        elensw=sqrt((c4.x-c1.x)**2+(c4.y-c1.y)**2+(c4.z-c1.z)**2)  # element len
gth in spanwise direction
```

buildblade.py

```
        # Locations of Element Center
        elx=(c1.x + c3.x) *0.5
        ely=(c1.y + c3.y) *0.5
        elz=(c1.z + c3.z) *0.5
        el=Element(elemnum,c1.num,c2.num,c3.num,c4.num,s5.num,s6.num,s7.num,s8.n
um,elx,ely,elz,elencw,elensw)
        elemnum+=1
        SparElements.append(el)
Elements=SkinElements+SparElements
#-----------------------------------------------------
# Sections
#-----------------------------------------------------
sectionsplit0=[0.0,0.5,0.7,0.9,1.1,1.3,2.4,2.6,4.7,6.8,8.9,11.4,14.6,16.3,17.9,1
9.5,22.2,24.9,27.6,35.8,43.9,52.0,60.2,66.7,68.3,73.2,76.4,84.6,89.4,94.3,95.7,9
7.2,98.6,100.0]
sectionsplit=stationzvals
# Initialize layup sections with Labels
chrd_sec=["L","S","A","T"]
span_sec=range(34)[1:]
sec=[]
sparsec=[]
for c in chrd_sec:
    row=[]
    for s in span_sec:
        label=c+str(s)
        derp=[]
        derp.append(label)
        row.append(derp)
    sec.append(row)
# Initialize aero sections (applied force sections)
execfile(sdir+"readaerosections.py")
aero_borders=[]
for a in Asecs:
    aero_borders.append(a.rnodes-0.5*a.drnodes-hubrad)
aerosec=[]   # holds skin elements associate with each of the aerodynamic blade
sections used in FAST
for i in range(len(aero_borders)):
    derp=[]
    nerp=[]
    derp.append("AeroE"+str(i))
    aerosec.append(derp)
trail_edge_num=Cseed+Dseed+Eseed-1
# Sort Elements
for e in SkinElements:
#---non-tip
    if e.elcen[2] < 94.4:
#-find the x-position of the trailing edge
        trailing_edge=SpSpline['spsp'+str(trail_edge_num)]
        find_trail=polyfit(trailing_edge.z,trailing_edge.x,5)
        tex=polyval(find_trail,e.elcen[2])
    # sort chordwise
        e.base_sort_cw(tex)
#---tip
    if e.elcen[2] > 94.4:
        e.cw=0
# sort spanwise (base and tip)
    e.sort_sw(sectionsplit[:-1],aero_borders)
    sec[e.cw][e.sw].append(e.num)
    aerosec[e.asnum].append(e.num)
# Sort Spanwise Spar
for s in span_sec:
```

buildblade.py

```
    sparlabel="Spar"+str(s)
    berp=[]
    berp.append(sparlabel)
    sparsec.append(berp)
for e in SparElements:
    e.sort_sw(spansplit,aero_borders)
    sparsec[e.sw].append(e.num)
# Get surface area for aerosections (skin elements only)
asecAREA=[]
for a in aerosec:
    A=0
    for i in range(len(a))[1:]:
        A+=Elements[a[i]-1].area
    asecAREA.append(A)
# Read layup
Lay=Layup("layup.txt")
#-------------------------------------
# Grab nodes for station rigid tie
spanz=[]
for i in range(len(SkinNodes[0])):
    spanz.append(single(SkinNodes[0][i].z))
foo=[]
for s in range(len(Stations)):
    sta='sta'+str(s)
    Stations[sta].spannum=spanz.index(single(Stations[sta].z)    )
    foo.append([s,Stations[sta].z,Stations[sta].spannum])
for f in foo:
    lon=[]
    for m in range(len(SkinNodes)):
            lon.append(SkinNodes[m][f[2]].num)
    f.append(lon)
#-----------------------------------------
# Create Station Reference Nodes
#-----------------------------------------
start_refnodes=nn # note where mesh nodes end and reference nodes start
RefNodes=[]   # These nodes are NOT part of the mesh, but will be tied to the me
sh numerically.  They will be used for the application of boundary conditions an
d loads
for i in range(len(Stations)):
    sta='sta'+str(i)
    RefNodes.append(Node(nn,0,0,0,0,Stations[sta].z))
 # (0,0,z) node created
    nn+=1
RecNodes=[]   # These nodes are part of the mesh, but are also selected as defle
ction measuring points
for i in range(len(Stations)):
    sta='sta'+str(i)
    RecNodes.append(SkinNodes[cwseed-Aseed-Bseed-1][Stations[sta].spannum].num)
       # Leading edge node number  (from skin nodes)
    RecNodes.append(SkinNodes[Cseed+Dseed+Eseed-1][Stations[sta].spannum].num)
       # Trailing edge node number (from skin nodes)

print "Blade has been built"
```

buildaerosections.py

```
#  This file shows how I calculated the locations of the aerosection to be used
in the FAST_AD.ipt input file.
#  ** All of the z-locations are relative to the root of the blade, rather than
the center of rotation.
#  ** The hub radius will have to be included for making the actual data file.

hubrad=2.5
aeroplot=0
class Asec:
    def __init__(self,sec_prox,sec_dist):
        self.pe=mids[sec_prox]+hubrad
        self.de=mids[sec_dist+1]+hubrad
        self.drnodes=self.de-self.pe
        self.rnodes=(self.de+self.pe)/2
        self.prof=sta_list[sec_prox][0]
        self.chrd=float(sta_list[sec_prox][1])
        self.twst=float(sta_list[sec_prox][2])
        self.x=[0,.5*self.chrd,.5*self.chrd,-.5*self.chrd,-.5*self.chrd]
        self.z=[self.rnodes,self.pe,self.de,self.de,self.pe]
        self.output=[self.rnodes,self.twst,self.drnodes,self.chrd,self.prof]
    def plot(self):
        plot(self.z,self.x,'o')
        plot([self.z[1],self.z[2],self.z[3],self.z[4],self.z[1]],[self.x[1],self
.x[2],self.x[3],self.x[4],self.x[1]])
        show()

def checkfoils(foil_list):
    for i in range(len(foil_list)-1):
        if foil_list[i].de ≡ foil_list[i+1].pe:
            print str(i) + ' and ' + str(i+1) + ' have a proper boundary.'
        else:
            print 'WARNING! '+str(i)+' and '+str(i+1)+' have a misfitting boundary.'


Asecs=[]  # manually build sections and add them to this list
sta_zs=[] # Z location of the sandia specified stations
for st in Stations:
    sta_zs.append(Stations[st].z)
sta_zs.sort()
mids=[]   # Z location of the midpoints between stations.  These are also the bo
undaries of the aerosections.
mids.append(0) # must bound at the root
for s in range(len(sta_zs)-1):
    mids.append((sta_zs[s]+sta_zs[s+1])/2)
mids.append(100)  # must bound at the tip
aerosec_len=[]   # Length of the aerosections  (DRNodes)
for i in range(len(mids)-1):
    aerosec_len.append(mids[i+1]-mids[i])
aerosec_cen=[]   # Center of the aerosections   (RNodes w/o hub rad)
for i in range(len(mids)-1):
    aerosec_cen.append(mids[i]+aerosec_len[i]/2)
diff=[]    # diffence between station location and aerocenter
for i in range(len(sta_zs)):
    diff.append(sta_zs[i]-aerosec_cen[i])
pdiff=[]   # difference as a fraction of aerosection length
for i in range(34):
    pdiff.append(diff[i]/aerosec_len[i])
Asecs.append(Asec(0,5))
Asecs.append(Asec(6,7))
for i in range(23)[8:]:
    Asecs.append(Asec(i,i))
```

buildaerosections.py

```
Asecs.append(Asec(23,24))
for i in range(34)[25:]:
    Asecs.append(Asec(i,i))
```

buildstasections.py

```
stiffstaA=[]    # list of stations with z=0 used for calculating EI
ststas=[]       # list containing dictionaries with paris of stiffness-stations
for i in range(number_of_stations):
    S={}
    crd=float(sta_list[i][1])  # chord length for the station
    for zees in [0,1]:
        EIsta=Station(i)  # initialize station
        EIsta.setfoil(sta_list[i][0],Controls['dir'])  # set airfoil

        z=float(sta_list[i][4])*float(Controls['length'])  # used to determine sp
ar status
#-----------------------------------------------------


# Seed stations  chordwise (create spline)


# y-shift and rotation are neglected here
#-----------------------------------------------------


# Adjust x-position (needed to make sure the skin aligns with the shear webs)
        EIsta.xshift(float(sta_list[i][3]))
# Adjust thickness to chord ratio


        EIsta.set_tcr(float(sta_list[i][5]))
# Apply Chord Length


        EIsta.resize(float(sta_list[i][1]))
        EIsta.seed=[Aseed,Bseed,Cseed,Dseed,Eseed]
        EIsta.chordwise_seed(z,z_sparend,Asize,Esize,spar1,spar2)
        EIsta.oldx =EIsta.x
        EIsta.oldy =EIsta.y
# Split the normalized foil into top and bottom


        EIsta.top_bot()
# Spline 'y' values for top and bottom


        EIsta.y1=y1=EIsta.spline(EIsta.h1_x,EIsta.h1_y,EIsta.cws)
        EIsta.y2=y2=EIsta.spline(EIsta.h2_x,EIsta.h2_y,EIsta.cws)
# Smooth out leading edge


        N=Aseed+Bseed
        EIsta.lead_x = lead_x = concatenate((EIsta.cws[:N][::-1],EIsta.cws[:N]))
        EIsta.lead_y = lead_y = concatenate((y1[:N][::-1],y2[:N]))
        EIsta.lead_seed = lead_seed = linspace(min(lead_y),max(lead_y),len(lead_
y)+1)
        EIsta.new_x = new_x = spline(lead_y,lead_x,lead_seed)  # seed the leadin
g edge with an odd number of nodes
# Recombine top, bottom, and leading edge


        EIsta.combine_foil(y1, y2, lead_seed, new_x, N)
# Set spanwise location to either z=0 or z=1, depending on iteration of 'zees'
        EIsta.zval( ([0,20*crd])[zees] )
# Generate a printable summary
```

buildstasections.py

```
        EIsta.coords()
# Add to Stiffness-Station list
        stiffstaA.append(EIsta)
        name='sta'+str(zees)
        S[name]=EIsta
        S['sta0'].zloc=z
    ststas.append(S)
print "'buildstasections' is complete"
```

genFAST.py

```
#---------------------------------------- Part 1 ------------------------------
----------
#----------------------------------  Write Blade Data File  --------------------
----------
subprocess.call(['cp',sdir+'GenBlade.dat','TestBlade.dat'])
blf=open("TestBlade.dat","a")
# cycle stations
for i in range(number_of_stations):
    sta="sta"+str(i)
    s=Stations[sta]
# These are given as Bronco Blade inputs
    blade_fraction=str(  float(sta_list[i][4])  )
    aero_center=str(0.25)
    strc_twst=str(sta_list[i][2])
# These are calculated via Abaqus
    bmass_den=str(s.lmass)
    flp_stff=str(s.EIf) #str(EI[i+1][1])
    edg_stff=str(s.EIe) #str(EI[i+1][2])
    gj_stff=str(s.GJ)
# According to FAST user manual, these are only needed for an ADAMS model
    ea_stff=alpha=flpiner=edginer=precrvRef=preswpRef=flpcgOf=edgcgOf=flpEAOf=ed
gEAOf=str(0)
# write it to the AeroDyn file
    blf.write("%s" % (blade_fraction+'\t'+aero_center+'\t'+strc_twst+'\t'+bmass_den
+'\t'+flp_stff+'\t'+edg_stff+'\t'  ) )
    blf.write("%s" % (gj_stff+'\t'+ea_stff+'\t'+alpha+'\t'+flpiner+'\t'+edginer+'\t'+
precrvRef+'\t' ) )
    blf.write("%s\n" % (preswpRef+'\t'+flpcgOf+'\t'+edgcgOf+'\t'+flpEAOf+'\t'+edgEAO
f ) )
# Move into Mode Shapes
blf.write("%s\n" % "---------------------- BLADE MODE SHAPES ------------------------
----------------" )
f1=flapmode1
f2=flapmode2
e1=edgemode1
t1=["Flap mode 1","     ","     ","     ","     "]
t2=["Flap mode 2","     ","     ","     ","     "]
t3=["Edge mode 1","     ","     ","     ","     "]
for i in range(5):
    blf.write("%s\n" % (str(f1[i]) + "\t BldFl1Sh("+str(i+2)+") - "+t1[i]+", coeff of x^"+
str(i+2) ) )
for i in range(5):
    blf.write("%s\n" % (str(f2[i]) + "\t BldFl2Sh("+str(i+2)+") - "+t2[i]+", coeff of x^"+
str(i+2) ) )
for i in range(5):
    blf.write("%s\n" % (str(e1[i]) + "\t BldFl1Sh("+str(i+2)+") - "+t3[i]+", coeff of x^"+
str(i+2) ) )
blf.close()
```

getEI.py

```
#------------ Run Abaqus:  Submit Section*.inp to FE solver  -------------------
--------
if 'r' in optlist['Sections']:
    section="Section"+str(job)  # input file name
    os.system("cd "+dirname+"; abaqus -j "+section+" -cpus 4 -memory '4 GB' -interactive -double"
)
    #    subprocess.call(['abaqus','-j',section,'-cpus','4','-memory','4 GB','-i
nteractive','-double'])  # perhaps include file for FE options
    print "Section "+ str(job) + " Job complete"  # Completion message

#------------ Post-Process from .odb file -----------------
if 'r' in optlist['Sections']:
    # subprocess.call(['cp',sdir+'/writexy-standard.py','.'])  # Brings code fro
m sdir into PWD.
    # subprocess.call(['abaqus','cae','-noGUI','writexy-standard.py'])  # Run ab
aqus python script.  It writes displacements to file Section*.out
    os.system("cp "+sdir+"/writexy-standard.py "+dirname)
    os.system("cd "+dirname+"; abaqus cae -noGUI writexy-standard.py")
    print "Section "+ str(job) + " post-processing complete"  # Completions message

#-------- Read Results of Analyses ------------------
resultfile=open(dirname+"/Section"+str(job)+".out")
resultlist=resultfile.readlines()
results=[]
for ln in resultlist:
    results.append(ln.split())
Csta.mass=float(results[3][1])                    # get model mass
Csta.lmass=Csta.mass/Csta.sectl                        # get linear mass by divi
ding model mass by section length
U=[]                                                                          #
x-direction displacements
U.append([float(results[13][1]),float(results[15][1]),float(results[17][1])]) #
leading node - torque, flap, edge
U.append([float(results[26][1]),float(results[28][1]),float(results[30][1])]) #
trailing node - torque, flap, edge
V=[]                                                                          #
y-direction displacements
V.append([float(results[39][1]),float(results[41][1]),float(results[43][1])]) #
leading node - torque, flap, edge
V.append([float(results[52][1]),float(results[54][1]),float(results[56][1])]) #
trailing node - torque, flap, edge
#-------- Find Displacement of Center ---------------------------------
from pproctools import *
Psec=postSection(RefNodes[2].x,RefNodes[2].y,RefNodes[3].x,RefNodes[3].y)     #
creates a postSection with the original points of the leading and trailing nodes
Psec.findCenter(U[0][0],V[0][0],U[1][0],V[1][0])                              #
used torque-step displacements to calculate center
Psec.findDisplacement('f',U[0][1],V[0][1],U[1][1],V[1][1])                   #
displacement for flapwise load
Psec.findDisplacement('e',U[0][2],V[0][2],U[1][2],V[1][2])                    #
displacement for edgewise load
#------- Compute EI -------------------------------
Csta.df=abs(Psec.cvf)
Csta.de=abs(Psec.cue)
Csta.EIf=1*(Csta.sectl**3) / (3*Csta.df)
Csta.EIe=1*(Csta.sectl**3) / (3*Csta.de)
Csta.GJ=1*Csta.sectl/Psec.twist
Csta.cfrac=Psec.cfrac   # save center of twist location as a fraction of the cho
rd

# ! Note in the report that global variables (Csta.attribute) is being used for
```

getEI.py

```
debugging purposes, both by the author and any other user/developer
# ! Once the program is proven to work smoothly, these could be made local to sa
ve time/space  (but not the space/time continuum)
# ! Some parameters, like Csta.cfrac, would have to remain, as they are referenc
ed later
```

getModes.py

```
#------------ Run Abaqus ----------------------------
if 'r' in optlist['Modes']:
    subprocess.call(['abaqus','-j',infile[:-3]+"_freq" ,'-interactive','-double'])  # Sub
mit frequency analysis to FE solver
#-------- Post-Process Results of Analyses ------------------


if 'p' in optlist['Modes']:
    subprocess.call(['cp',sdir+'writexy-freq.py','.'])                                  #
 copy post-processing script into PWD
    subprocess.call(['abaqus','cae','-noGUI','writexy-freq.py'])                       #
run post-processing script in abaqus.  Creates freq.rpt
#--------- Read in from file -------------
outs=[]
if os.path.isfile('freq.rpt'): frq=open("freq.rpt")
else: print "Frequency file 'freq.rpt' found."
contents=frq.readlines()
for ln in contents:
    outs.append( ln.split() )
#--------- Calculate Station Deflections -------
deflection=[]
ltdef=[]
for i in range(number_of_stations):                # cycles through stations
    us=[]
    vs=[]
    ltu=[]
    ltv=[]
    for m in [1,2,3]:              # cycles through modes (may need to change ba
sed on which modes are flap and edge)
        # deflection U1
        lead=float(outs[(3+m) + (i*32)][1])
        trail=float(outs[(19+m) + (i*32)][1])
        du=lead+Stations['sta'+str(i)].cfrac*(trail-lead)
        us.append(du)
        ltu.append((lead,trail,du,Stations['sta'+str(i)].cfrac))
        # deflection U2
        lead=float(outs[(1093+m) + (i*32)][1])
        trail=float(outs[(1109+m) + (i*32)][1])
        dv=lead+Stations['sta'+str(i)].cfrac*(trail-lead)
        ltv.append((lead,trail,du,Stations['sta'+str(i)].cfrac))
        vs.append(dv)
    deflection.append([us,vs])
    ltdef.append([ltu,ltv])
stationzvals=[]
for ln in sta_list:
    stationzvals.append(single(ln[4]))
z=array(stationzvals)
a=transpose(vstack([z**2,z**3,z**4,z**5,z**6]))
u1=[]
v1=[]
v2=[]
for s in deflection:
    u1.append(s[0][1])   # x-direction, 2nd mode
    v1.append(s[1][0])   # y-direction, 1st mode
    v2.append(s[1][2])   # y-direction, 3rd mode
flapmode1=linalg.lstsq(a,v1)[0]
flapmode2=linalg.lstsq(a,v2)[0]
edgemode1=linalg.lstsq(a,u1)[0]
# the least squares above has been producing modes where the sum of the coeffici
ents is
# not exactly 1; results looked like .98234235, which is too far from 1 for FAST
```

getModes.py

```
 to accept.
# The code below multiplies the mode shape coefficents so they will add to 1 at
the tip
# This new code can be removed once the least squares calculation produces more
precise results.
newmodes=[]
for i in [flapmode1, flapmode2, edgemode1]:
    adjust=1.0/sum(i)
    tmp=[]
    for j in i:
        tmp.append(j*adjust)
    newmodes.append(tmp)
flapmode1=newmodes[0]
flapmode2=newmodes[1]
edgemode1=newmodes[2]
```

pproctools.py

```
# This is a module!
# This contains functions and classes for post-processing the FEA results
from numpy import *
from pylab import *
from scipy import interpolate
import warnings
warnings.simplefilter('ignore', np.RankWarning)

class postSection:
    def __init__(self,lead_x,lead_y,trail_x,trail_y):
        self.lx=lead_x
        self.ly=lead_y
        self.tx=trail_x
        self.ty=trail_y
        self.slope0=(trail_y-lead_y)/(trail_x-lead_x)
        self.yint0=lead_y-self.slope0*lead_x
        self.cvec0=[trail_y-lead_y,trail_x-lead_x]

    def findCenter(self,lead_u,lead_v,trail_u,trail_v):
        self.lxtq=self.lx+lead_u    # tq for torque
        self.lytq=self.ly+lead_v
        self.txtq=self.tx+trail_u
        self.tytq=self.ty+trail_v
        self.cvec1=[self.txtq-self.lxtq,self.tytq-self.lytq]
        self.twist=arccos(dot(self.cvec0,self.cvec1)/(norm(self.cvec0,2)*norm(se
lf.cvec1,2)))
        self.slope1=(self.lytq-self.tytq)/(self.lxtq-self.txtq)
        self.yint1=self.lytq-self.slope1*self.lxtq
        self.cx=(self.yint1-self.yint0)/(self.slope0-self.slope1)
        self.cy=self.yint1 + self.slope1*self.cx
        self.cfrac=(self.cx-self.lx)/(self.tx-self.lx)

    def findDisplacement(self,eorf,lead_u,lead_v,trail_u,trail_v):
        if eorf == 'e':
            self.cue=lead_u+(trail_u-lead_u)*self.cfrac
            self.cve=lead_v+(trail_v-lead_v)*self.cfrac
        if eorf == 'f':
            self.cuf=lead_u+(trail_u-lead_u)*self.cfrac
            self.cvf=lead_v+(trail_v-lead_v)*self.cfrac
```

printblade.py

```
#-------------------------------------------------
# Printing
#-------------------------------------------------
#==== Main Mesh Input File ================
abq_out_file=infile[:-3]+"_mesh.inp"
abq=open(abq_out_file,'w')
abq.write("%s\n" % "*Part, name=Blade" )
abq.write("%s\n" % "*Node, nset=nall" )
for n in Nodes:
    abq.write("%s\n" % n.output)
abq.write("%s\n" % "*Element, type=S8R, ELSET=skin" )
for i in range(len(SkinElements)):
    abq.write("%s\n" % SkinElements[i].output)
abq.write("%s\n" % "*Element, type=S8R, ELSET=spar" )
for i in range(len(SparElements)):
    abq.write("%s\n" % SparElements[i].output)
# print sections to Elsets
abq.write("%s\n" % "** Skin Element Sets")
for m in range(len(sec)):
    for n in range(len(sec[0])):
        abq.write("%s\n" % ('*Elset, elset=CompositeLayup-' + sec[m][n][0] + '-1' ))

        for i in range(1+len(sec[m][n])/16):        # print element numbers in r
ows of 16 (last row may be less)
            abq.write("\t%s\n" % (str( sec[m][n][(1+16*i):][:16] )[1:][:-1] ))


        abq.write("%s\n" % ('*Shell Section, elset=CompositeLayup-' + sec[m][n][0] + '-1 , co
mposite, layup=CompositeLayup-' + sec[m][n][0] + ", offset=SPOS"))
        abq.write("%s\n" % ('.0006, '+'3, '+'gelcoat, '+'0, '+'gelcoat ' ))
        abq.write("%s\n" % ('.005, '+'3, '+'triax, '+'90, '+'ext-triax ' ))
        reinforcement=Lay.get_layer(m,n)
        abq.write("%s\n%s\n%s\n" % (reinforcement[0],reinforcement[1],reinforcemen
t[2] ))
        abq.write("%s\n" % ('.005, '+'3, '+'triax, '+'90, '+'int-triax ' ))
        abq.write("%s\n" % ('.005, '+'3, '+'resin, '+'0, '+'parasitic_resin ' ))
abq.write("%s\n" % ('*Shell Section, elset=spar , composite, layup=CompositeLayup-Spar' ))
abq.write("%s\n" % ('.003, '+'3, '+'biax, '+'90, '+'double-bias-1 ' ))
abq.write("%s\n" % ('.080, '+'3, '+'foam, '+'0, '+'foam ' ))
abq.write("%s\n" % ('.003, '+'3, '+'biax, '+'90, '+'double-bias-2 ' ))

for asn in range(len(aerosec)):
    abq.write("*Elset, elset=aeroset_"+str(asn)+"\n")
    for i in range(1+len(aerosec[asn])/16):        # print element numbers in ro
ws of 16 (last row may be less)
        abq.write("\t%s\n" % (str( aerosec[asn][(1+16*i):][:16] )[1:][:-1] ))
    abq.write("*Surface, type=ELEMENT, name=aerosurf_"+str(asn)+"\n")
    abq.write("aeroset_"+str(asn)+", SNEG \n")
abq.write("%s\n" % "*End Part" )
abq.write("%s\n" % "*Assembly, name=Assembly")
abq.write("%s\n" % "*Instance, name=Blade-1, part=Blade")
abq.write("%s\n" % "*End Instance")

# Write station nodes
for f in foo:
    abq.write("%s\n" % ('*Nset, nset=_Sta'+str(f[0])+'Set, internal, instance=Blade-1'))
    for i in range(1+len(f[3])/16):        # print element numbers in rows of 16
 (last row may be less)
        abq.write("\t%s\n" % (str( f[3][(1+16*i):][:16] )[1:][:-1] ))

abq.write("%s\n" % ('*Nset, nset=_RecNodes, internal, instance=Blade-1'))
```

printblade.py

```
for i in range(1+len(RecNodes)/16):        # print element numbers in rows of 16
 (last row may be less)
    abq.write("\t%s\n" % (str( RecNodes[(1+16*i):][:16] )[1:][:-1] ))

abq.close()
fr=open("recnodes.txt","w")
for r in RecNodes: fr.write(str(r)+"\n")
fr.close()
```

printFreqTest.py

```
abq_out_file=infile[:-3]+"_freq.inp"
ftest=open(abq_out_file,'w')
ftest.write("%s\n" % ("*include, input="+infile[:-3]+"_mesh.inp") )  # bring in the fil
e written by 'printmain.py'
ftest.write("%s\n" % "*End Assembly")
outfile=ftest  # 'outfile' is used to write material properties


execfile("writemats.py")
ftest.write("%s\n" % "*Step, name=Step-1, perturbation")
ftest.write("%s\n%s\n" % ("*Frequency, eigensolver=Lanczos, acoustic coupling=on, normalization=displac
ement","10, , , ,"))
ftest.write("%s\n%s\n" % ("*Boundary","_Sta0Set, 1, 6"))
ftest.write("%s\n" % "*Output, field, variable=PRESELECT")
ftest.write("%s\n" % "*Output, history")
ftest.write("%s\n" % ("*Node Output, nset=_RecNodes" ))
ftest.write("%s\n" % "U1,U2,UR3")
ftest.write("%s\n" % "*Output, history, variable=PRESELECT")
ftest.write("%s\n" % "*End Step")
ftest.close()
```

printstasections.py

```
#-----------------------------------------------
# Print Extruded Stations Sections
#-----------------------------------------------
sect_fn=dirname+"/Section"+str(job)+".inp"
sect=open(sect_fn,'w')
sect.write("%s\n" % ("*Part, name=Stiffness-Station-Section"+str(job)) )
sect.write("%s\n" % "*Node, nset=all")
for n in Nodes:
    sect.write("%s\n" % n.output)
sect.write("%s\n" % "*Element, type=S8R, ELSET=skin" )
for i in range(len(SkinElements)):
    sect.write("%s\n" % SkinElements[i].output)
sect.write("%s\n" % "*Element, type=S8R, ELSET=spar" )
for i in range(len(SparElements)):
    sect.write("%s\n" % SparElements[i].output)
sect.write("%s\n" % "** Skin Element Sets")
# Print Element Sets
for m in range(len(sec)):
    sect.write("%s\n" % ('*Elset, elset=CompositeLayup-' + sec[m][0] + '-1' ))
    for i in range(1+len(sec[m])/16):        # print element numbers in rows of
16 (last row may be less)
        sect.write("\t%s\n" % (str( sec[m][(1+16*i):][:16] )[1:][:-1] ))
    sect.write("%s\n" % ('*Shell Section, elset=CompositeLayup-' + sec[m][0] + '-1 , composite,
layup=CompositeLayup-' + sec[m][0] + ", offset=SPOS" ))
    sect.write("%s\n" % ('.0006,'+'3, '+'gelcoat, '+'0, '+'gelcoat ' ))
    sect.write("%s\n" % ('.005, '+'3, '+'triax, '+'90, '+'ext-triax ' ))
    reinforcement=Lay.get_layer(m,job)
    sect.write("%s\n%s\n%s\n" % (reinforcement[0],reinforcement[1],reinforcement[2
] ))
    sect.write("%s\n" % ('.005, '+'3, '+'triax, '+'90, '+'int-triax ' ))
    sect.write("%s\n" % ('.005, '+'3, '+'resin, '+'0, '+'parasitic_resin ' ))
# The spar elset has already been created in the *Element command
sect.write("%s\n" % ('*Shell Section, elset=spar , composite, layup=CompositeLayup-Spar' ))
sect.write("%s\n" % ('.003, '+'3, '+'biax, '+'90, '+'double-bias-1' ))
sect.write("%s\n" % ('.080, '+'3, '+'foam, '+'0, '+'foam ' ))
sect.write("%s\n" % ('.003, '+'3, '+'biax, '+'90, '+'double-bias-2' ))
RefNodes.append(SkinNodes[cwseed-Bseed-Aseed-1][spanseed-1]) # leading edge refe
rence node
RefNodes.append(SkinNodes[Cseed+Dseed+Eseed-1][spanseed-1])  # trailing edge ref
erence node
sect.write("%s\n%s\n" % ("*Nset, nset=measure", str(RefNodes[2].num) +", "+str(RefNodes
[3].num) ) )
sect.write("%s\n" % "*End Part")
sect.write("%s\n" % "*Assembly, name=Assembly")
sect.write("%s\n" % ("*Instance, name=Bsec"+str(job)+"-1, part=Stiffness-Station-Section"+str(jo
b)))
sect.write("%s\n" % "*End Instance")
sect.write("%s\n" % "*Node, nset=Ref1")
sect.write("%s\n" % RefNodes[0].output)
sect.write("%s\n" % "*Node, nset=Ref2")
sect.write("%s\n" % RefNodes[1].output)
# Write station nodes
for f in [foo[0],foo[1]]:
    sect.write("%s\n" % ('*Nset, nset=_Sta'+str(f[0])+'Set, internal, instance=Bsec'+str(job)+
'-1'))
    for d in range(1+len(f[3])/16):        # print element numbers in rows of 16
 (last row may be less)
        sect.write("%s\n" % (str( f[3][(16*d):][:16] )[1:][:-1] ))

for f in [foo[0],foo[1]]:
    sect.write("%s\n" % ('*Rigid Body, ref node='+str(RefNodes[f[0]].num)+', tie nset=_Sta'+
```

printstasections.py

```
str(f[0])+'Set'))
sect.write("%s\n" % "*End Assembly")
outfile=sect  # 'outfile' is used to write material properties
execfile("writemats.py")
sect.write("%s\n%s\n" % ("*Boundary", (str(RefNodes[0].num)+", 1, 6")))
tipload=1/cwseed

sect.write("%s\n" % "*Step, name=Step-Torque")
sect.write("%s\n%s\n" % ("*Static",  "1.0, 1.0, 1e-05, 1.0"))
sect.write("%s\n%s\n" % ("*Cload", (str(RefNodes[1].num)+", 6, 1 " ) ) )
sect.write("%s\n" % "*Output, field, variable=PRESELECT")
sect.write("%s\n%s\n%s\n" % ("*Output, history","*Node Output, nset=Bsec"+str(job)+"-1.measure"
,"U1,U2") )
sect.write("%s\n%s\n%s\n" % ("*Output, history","*Element Output","MASS") )
sect.write("%s\n" % "*End Step")

sect.write("%s\n" % "*Step, name=Step-Flapwise")
sect.write("%s\n%s\n" % ("*Static",  "1.0, 1.0, 1e-05, 1.0"))
sect.write("%s\n%s\n" % ("*Cload, op=NEW", (str(RefNodes[1].num)+", 2, 1 " ) ) )
sect.write("%s\n" % "*Output, field, variable=PRESELECT")
sect.write("%s\n%s\n%s\n" % ("*Output, history","*Node Output, nset=Bsec"+str(job)+"-1.measure"
,"U1,U2") )
sect.write("%s\n" % "*End Step")

sect.write("%s\n" % "*Step, name=Step-Edgewise")
sect.write("%s\n%s\n" % ("*Static",  "1.0, 1.0, 1e-05, 1.0"))
sect.write("%s\n%s\n" % ("*Cload, op=NEW", (str(RefNodes[1].num)+", 1, 1 " ) ) )
sect.write("%s\n" % "*Output, field, variable=PRESELECT")
sect.write("%s\n%s\n%s\n" % ("*Output, history","*Node Output, nset=Bsec"+str(job)+"-1.measure"
,"U1,U2") )
sect.write("%s\n" % "*End Step")
sect.close()

mnodes=open(dirname+"/measurenodes.tmp",'w')
mnodes.write("%s\n%s\n%s\n" % (str(job), str(RefNodes[2].num) , str(RefNodes[3].nu
m)) )  #("one","two","three")
mnodes.close()
print "Section" + str(job) +" input has been printed."
```

readaerosections.py

```
#  This file shows how I calculated the locations of the aerosection to be used
in the FAST_AD.ipt input file.
#  ** All of the z-locations are relative to the root of the blade, rather than
the center of rotation.
#  ** The hub radius will have to be included for making the actual data file.

hubrad=2.5
aeroplot=0
class Asec:
    def __init__(self,num):
        self.asecnum=num
    def build(self,sec_prox,sec_dist):
        self.pe=mids[sec_prox]+hubrad
        self.de=mids[sec_dist+1]+hubrad
        self.drnodes=self.de-self.pe
        self.rnodes=(self.de+self.pe)/2
        self.prof=sta_list[sec_prox][0]
        self.chrd=float(sta_list[sec_prox][1])
        self.twst=float(sta_list[sec_prox][2])
        self.x=[0,.5*self.chrd,.5*self.chrd,-.5*self.chrd,-.5*self.chrd]
        self.z=[self.rnodes,self.pe,self.de,self.de,self.pe]
        self.output=[self.rnodes,self.twst,self.drnodes,self.chrd,self.prof]
    def plot(self):
        plot(self.z,self.x,'o')
        plot([self.z[1],self.z[2],self.z[3],self.z[4],self.z[1]],[self.x[1],self
.x[2],self.x[3],self.x[4],self.x[1]])
        show()
def checkfoils(foil_list):
    for i in range(len(foil_list)-1):
        if foil_list[i].de ≡ foil_list[i+1].pe:
            print str(i) + ' and ' + str(i+1) + ' have a proper boundary.'
        else:
            print 'WARNING! '+str(i)+' and '+str(i+1)+' have a misfitting boundary.'
Asecs=[]  # manually build sections and add them to this list
f=open("AQ_FASTsim/SNL13pt2-00-Land_AeroDyn.ipt")
frl=f.readlines()
aerodyn=[]
for ln in frl:
    aerodyn.append(ln.split())
num_asec=int(aerodyn[26][0])
for j in aerodyn[-num_asec:]:
    tmp=Asec((num_asec+1)-len(aerodyn)+aerodyn.index(j))
    tmp.rnodes=float(j[0])
    tmp.atwist=float(j[1])
    tmp.drnodes=float(j[2])
    tmp.chord=float(j[3])
    tmp.nfoil=float(j[4])
    Asecs.append(tmp)
```

readelm.py

```
read = 1
plotvals = 0
startup = .2  # approximate fraction of analysis where steady state results begi
n
import math
if read == 1:
    elmfile=open(optlist['Loads'][-1])
    elm=elmfile.readlines()
    rows=[]
    for ln in elm:
        rows.append(ln.split())
numvars=len(rows[4])
numasec=int(Controls['number_aerosections'])
time_inc=len(rows)
columns=[]
Fnorm=[]
Ftan=[]
for c in range(numvars):
    cmn=[]
    for r in rows[3:]:
        cmn.append(float(r[c]))
    columns.append(cmn)
num_tsteps=len(columns[0])    # total number of time steps conducted in FAST
norm_clmns=[]  # Find columns holding normal and tangential forces
tan_clmns=[]
for i in range(len(rows[1])):
    if rows[1][i][:5] == 'ForcN': norm_clmns.append(i)
    if rows[1][i][:5] == 'ForcT': tan_clmns.append(i)
for i in norm_clmns:   Fnorm.append(columns[i])
for i in tan_clmns:    Ftan.append(columns[i])
tmaxN=[]   # Find time of maximum force for each airfoils
tmaxT=[]
for i in norm_clmns: tmaxN.append(columns[i].index(max(columns[i])))
for i in tan_clmns: tmaxT.append(columns[i].index(max(columns[i])))
totN=[]
totT=[]
for i in range(num_tsteps):
    tmp=[]
    for j in norm_clmns: tmp.append(columns[j][i])
    totN.append(sum(tmp))
    tmp=[]
    for j in tan_clmns: tmp.append(columns[j][i])
    totT.append(sum(tmp))
Nmax_step=totN.index(max(totN))
Tmax_step=totT.index(max(totT))
Ltime=Nmax_step
pitch=[]
for i in range(numasec):
    pitch.append(columns[i*12+10][int(startup*time_inc)])
def rotate_loads(norm,tan,pitch):
    rp=pitch*pi/180  # convert to radians
    Fx=-sin(rp)*norm + cos(rp)*tan
    Fy=cos(rp)*norm  + sin(rp)*tan
    return [Fx,Fy]
```

staNodesElms.py

```
#---- Create Spanwise Splines ----------
#----- spar determination -----------------
if z_sparstart < gz < z_sparend :  # This is the range in which the main spars a
re present
    staspar='true'
else:
    staspar='false'
## !!!!!!!!!!!! THIS NEXT SECTION IS ONLY FOR SANDIA 3-SPAR MODELS!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!
if 14.6 < gz < 60.2:   # This is the range in which spar 3 is present (SANDIA MO
DEL ONLY!!!!!)
    staspar3='true'
else:
    staspar3='false'
staspar3='false'   # uncomment if there are only 2 spars in the model

Csta.sectl=section_length=ststas[job]['sta1'].z
num_spsp=len(ststas[0]['sta0'].x)       # number of spanwise splines

Splines=[]  # list containing spanwise splines

#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
spanseed=491 #   #<<<<<<< Spanwise Mesh Parameter
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

for i in range(num_spsp):
    SS=SpanSpline(i,ststas[job])
    SS.splitseed=spanseed
    SS.spansplit=0  # value of 0 here triggers simplified interpolation
    SS.smooth(1,1)  # (1,1) are ... ?
    Splines.append(SS)
#-------------------------------------------------


# Create Spars


#-------------------------------------------------


if staspar == 'true':
    sparmesh=31  #<<<<<<< Mesh Parameter
    spar1_top=0
    spar1_bot=2*(Cseed+Dseed+Eseed)-2
    spar2_top=Cseed
    spar2_bot=Cseed+2*(Dseed+Eseed)-2
    Spar1 = Spar(Splines[spar1_top].xyz,Splines[spar1_bot].xyz,sparmesh,0,sectio
n_length)
    Spar2 = Spar(Splines[spar2_top].xyz,Splines[spar2_bot].xyz,sparmesh,0,sectio
n_length)
    Spar1.interp('stasec')
    Spar2.interp('stasec')
if staspar3 == 'true':
    spar3x= (gz-14.6)*-0.022451367366587198 + 2.66955
    trailindex= argmax( ststas[job]['sta0'].x )                    # index of tr
ailing edge (used to split the airfoil)
    spardiff=abs(array( ststas[job]['sta0'].x )-spar3x)            # difference
between each chordwise location and the prefered spar3 location
    spar3_top=list(spardiff).index( min(spardiff[:trailindex:2]))   # find the F
IRST minimum value of spardiff (closest to spar3 loctation)
    spar3_bot=list(spardiff[(spar3_top+1):]).index( min(spardiff[trailindex::2])
```

staNodesElms.py

```
)+(spar3_top+1)  # start the indexing process after the first index

                 # to find the SECOND minimum value of spardiff
    Spar3 = Spar(Splines[spar3_top].xyz,Splines[spar3_bot].xyz,sparmesh,0,sectio
n_length)
    Spar3.interp('stasec')
Nodes=[]
SkinNodes=[]  # 2-D matrix of nodes

nn=1
for i in range(cwseed):
    tmp=[]
    for j in range(spanseed):
        tn=Node(nn,i,j,Splines[i].xyz[j][0],Splines[i].xyz[j][1],Splines[i].xyz[
j][2])
        tmp.append(tn)
        nn+=1
    SkinNodes.append(tmp)
SparNodes=[]
SparNodes1=[]

SparNodes2=[]

SparNodes3=[]

if staspar ≡ 'true':
    ss=0


    for i in range(spanseed):


        tmp=[]


        for j in range(sparmesh):
            tn=Node(nn,i,j,Spar1.xyz[ss][0],Spar1.xyz[ss][1],Spar1.xyz[ss][2])
            if j ≡ 0:                       # Top and bottom of the spar is tied
 to the skin nodes along that spline
                tn=SkinNodes[spar1_bot][i]
            elif j ≡ (sparmesh-1):
                tn=SkinNodes[spar1_top][i]
            else:
                nn+=1
            ss+=1
            tmp.append(tn)
        SparNodes.append(tmp)
        SparNodes1.append(tmp)
    ss=0
    for i in range(spanseed):
        tmp=[]
        for j in range(sparmesh):
            tn=Node(nn,i,j,Spar2.xyz[ss][0],Spar2.xyz[ss][1],Spar2.xyz[ss][2])
            if j ≡ 0:
                tn=SkinNodes[spar2_bot][i]
            elif j ≡ (sparmesh-1):
                tn=SkinNodes[spar2_top][i]
            else:
                nn+=1
            ss+=1
            tmp.append(tn)
```

staNodesElms.py

```
        SparNodes.append(tmp)
        SparNodes2.append(tmp)
if staspar3 == 'true':
    ss=0
    for i in range(spanseed):
        tmp=[]
        for j in range(sparmesh):
            tn=Node(nn,i,j,Spar3.xyz[ss][0],Spar3.xyz[ss][1],Spar3.xyz[ss][2])
            if j == 0:                          # Top and bottom of the spar is tied
 to the skin nodes along that spline
                tn=SkinNodes[spar3_bot][i]
            elif j == (sparmesh-1):
                tn=SkinNodes[spar3_top][i]
            else:
                nn+=1
            ss+=1
            tmp.append(tn)
        SparNodes.append(tmp)
        SparNodes3.append(tmp)
for i in range(len(SkinNodes)):
    for j in range(len(SkinNodes[0])):
        Nodes.append(SkinNodes[i][j])
if staspar == 'true':
    for i in range(len(SparNodes)):
        for j in range(len(SparNodes[0])):
            Nodes.append(SparNodes[i][j])
Nodes=list(set(Nodes))   # Remove duplicates from Nodes
start_refnodes=nn # note where mesh nodes end and reference nodes start
RefNodes=[]
for i in range(len(S)):  # create reference nodes for Station A and Station B
    sta='sta'+str(i)
    RefNodes.append(Node(nn,0,0,0,0,ststas[job][sta].z))
    nn+=1
#-----------------------------------------------------

# Create Elements

#-----------------------------------------------------

SkinElements=[]
elemnum=1
# Skin Elements

span=range(spanseed)
chrd=range(cwseed)
chrd.append(0)
for a in range(cwseed/2):
    for b in range(int(spanseed/2)):
        m=a*2
        n=b*2
        c1=SkinNodes[chrd[m]][span[n]]
        c2=SkinNodes[chrd[m+2]][span[n]]
        c3=SkinNodes[chrd[m+2]][span[n+2]]
        c4=SkinNodes[chrd[m]][span[n+2]]
        s5=SkinNodes[chrd[m+1]][span[n]]
        s6=SkinNodes[chrd[m+2]][span[n+1]]
        s7=SkinNodes[chrd[m+1]][span[n+2]]
        s8=SkinNodes[chrd[m]][span[n+1]]
        # area and side lengths

        elencw=sqrt((c2.x-c1.x)**2+(c2.y-c1.y)**2+(c2.z-c1.z)**2)  # element len
```

staNodesElms.py

```
gth in chordwise direction
        elensw=sqrt((c4.x-c1.x)**2+(c4.y-c1.y)**2+(c4.z-c1.z)**2)  # element len
gth in spanwise direction
        # Locations of Element Center

        elx=(c1.x + c3.x) *0.5
        ely=(c1.y + c3.y) *0.5
        elz=(c1.z + c3.z) *0.5
        el=Element(elemnum,c1.num,c2.num,c3.num,c4.num,s5.num,s6.num,s7.num,s8.n
um,elx,ely,elz,elencw,elensw)
        elemnum+=1
        SkinElements.append(el)
SparElements=[]
if staspar ≡ 'true':
    for i in range(spanseed/2):
        for j in range(sparmesh/2):
            m=2*i
            n=2*j
            c1=SparNodes1[m][n]
            c2=SparNodes1[m+2][n]
            c3=SparNodes1[m+2][n+2]
            c4=SparNodes1[m][n+2]
            s5=SparNodes1[m+1][n]
            s6=SparNodes1[m+2][n+1]
            s7=SparNodes1[m+1][n+2]
            s8=SparNodes1[m][n+1]
            # area and side lengths

            elencw=sqrt((c2.x-c1.x)**2+(c2.y-c1.y)**2+(c2.z-c1.z)**2)  # element
 length in chordwise direction

            elensw=sqrt((c4.x-c1.x)**2+(c4.y-c1.y)**2+(c4.z-c1.z)**2)  # element
 length in spanwise direction
            # Locations of Element Center

            elx=(c1.x + c3.x) *0.5
            ely=(c1.y + c3.y) *0.5
            elz=(c1.z + c3.z) *0.5
            el=Element(elemnum,c1.num,c2.num,c3.num,c4.num,s5.num,s6.num,s7.num,
s8.num,elx,ely,elz,elencw,elensw)
            elemnum+=1
            SparElements.append(el)

    for i in range(spanseed/2):
        for j in range(sparmesh/2):
            m=2*i
            n=2*j
            c1=SparNodes2[m][n]
            c2=SparNodes2[m+2][n]
            c3=SparNodes2[m+2][n+2]
            c4=SparNodes2[m][n+2]
            s5=SparNodes2[m+1][n]
            s6=SparNodes2[m+2][n+1]
            s7=SparNodes2[m+1][n+2]
            s8=SparNodes2[m][n+1]
            # area and side lengths

            elencw=sqrt((c2.x-c1.x)**2+(c2.y-c1.y)**2+(c2.z-c1.z)**2)  # element
 length in chordwise direction

            elensw=sqrt((c4.x-c1.x)**2+(c4.y-c1.y)**2+(c4.z-c1.z)**2)  # element
```

staNodesElms.py

```
 length in spanwise direction
            # Locations of Element Center

            elx=(c1.x + c3.x) *0.5
            ely=(c1.y + c3.y) *0.5
            elz=(c1.z + c3.z) *0.5
            el=Element(elemnum,c1.num,c2.num,c3.num,c4.num,s5.num,s6.num,s7.num,
s8.num,elx,ely,elz,elencw,elensw)
            elemnum+=1
            SparElements.append(el)
if staspar3 ≡ 'true':
    for i in range(spanseed/2):
        for j in range(sparmesh/2):
            m=2*i
            n=2*j
            c1=SparNodes3[m][n]
            c2=SparNodes3[m+2][n]
            c3=SparNodes3[m+2][n+2]
            c4=SparNodes3[m][n+2]
            s5=SparNodes3[m+1][n]
            s6=SparNodes3[m+2][n+1]
            s7=SparNodes3[m+1][n+2]
            s8=SparNodes3[m][n+1]
            # area and side lengths

            elencw=sqrt((c2.x-c1.x)**2+(c2.y-c1.y)**2+(c2.z-c1.z)**2)  # element
 length in chordwise direction

            elensw=sqrt((c4.x-c1.x)**2+(c4.y-c1.y)**2+(c4.z-c1.z)**2)  # element
 length in spanwise direction
            # Locations of Element Center

            elx=(c1.x + c3.x) *0.5
            ely=(c1.y + c3.y) *0.5
            elz=(c1.z + c3.z) *0.5
            el=Element(elemnum,c1.num,c2.num,c3.num,c4.num,s5.num,s6.num,s7.num,
s8.num,elx,ely,elz,elencw,elensw)
            elemnum+=1
            SparElements.append(el)
Elements=SkinElements+SparElements
# sections need reworking
chrd_sec=["L","S","A","T"]
sec=[]
sparsec=[]
for c in chrd_sec:
    tmp=[]
    label=c+"_stiffsec"
    tmp.append(label)
    sec.append(tmp)
X=[]
for s in Splines: X.append(s.x[0])
trail_edge_num=X.index(max(X))
# Sort Elements


for e in SkinElements:
#---non-tip


    if gz < 94.4:
#-find the x-position of the trailing edge
```

staNodesElms.py

```
        trailing_edge=Splines[trail_edge_num]
        find_trail=polyfit(trailing_edge.z,trailing_edge.x,5)
        tex=polyval(find_trail,e.elcen[2])
    # sort chordwise


        e.base_sort_cw(tex)
#---tip


    else:
        e.cw=0
# sort spanwise (base and tip)


    sec[e.cw].append(e.num)
for e in SparElements:
    sparsec.append(e.num)
spanz=[]
for i in range(len(SkinNodes[0])):
    spanz.append(single(SkinNodes[0][i].z))
foo=[]
for i in range(len(ststas[job])):
    sta='sta'+str(i)
    foo.append([i,ststas[job][sta].z,spanz.index(single(ststas[job][sta].z)) ])
for f in foo:
    lon=[]
    for m in range(len(SkinNodes)):
        lon.append(SkinNodes[m][f[2]].num)
    f.append(lon)
print "Station Extrusion Sections have been built"
```

write−fastbuckle.py

```
#-------------- Write Load File -------------------------
fst=open(modelname+"_fastbuckle.inp","w")  #
fst.write("*include, input="+modelname+"_mesh.inp \n")      # include mesh file
fst.write("*End Assembly \n")
# print material properties
outfile=fst
execfile("writemats.py")
# write step and stuff
fst.write("*Step, name=Step-1, perturbation \n")
fst.write("*Buckle \n")
fst.write("20, , 38, 60 \n")
fst.write("*Boundary \n")
fst.write("_Sta0Set, 1, 6 \n")
#-------------- Write Loads ------------------------
fst.write("*Dsload \n")
for i in range(numasec):
    fn=Fnorm[i][Ltime]
    ft=Ftan[i][Ltime]
    p=pitch[i]
    fxfy=rotate_loads(fn,ft,p)  # Total force on section from FAST
    dfx=fxfy[0]/asecAREA[i]     # Force distributed over the total area of the s
ection
    dfy=fxfy[1]/asecAREA[i]     # Force distributed over the total area of the s
ection
    fst.write("%s\n" % ("Blade-1.aerosurf_"+str(i) + ", TRVEC, " + str(dfx) + ", 1,0,0 "
))
    fst.write("%s\n" % ("Blade-1.aerosurf_"+str(i) + ", TRVEC, " + str(dfy) + ", 0,1,0 "
))
fst.write("*Output, field, variable=PRESELECT \n")
fst.write("*Output, history, variable=PRESELECT \n")
fst.write("*End Step")
fst.close()
```

write−fastloads.py

```
#−−−−−−−−−−−−−− Write Load File −−−−−−−−−−−−−−−−−−−−−−−−
fst=open(modelname+"_fastloads.inp","w")  #
fst.write("*include, input="+modelname+"_mesh.inp \n")      # include mesh file
fst.write("*End Assembly \n")
# print material properties
outfile=fst
execfile("writemats.py")
# write step and stuff
fst.write("*Step, name=Step−1 \n")
fst.write("*Static \n")
fst.write("1., 1., 1e−05, 1. \n")
fst.write("*Boundary \n")
fst.write("_Sta0Set, 1, 6 \n")
execfile(sdir+"readaerosections.py")  # this provides data for the twist.  It probabl
y could be passed in more efficiently
aerosurface_xyloads=[]
#−−−−−−−−−−−−−− Write Loads −−−−−−−−−−−−−−−−−−−−−−−−
fst.write("*Dsload \n")
for i in range(numasec):
    fn=Fnorm[i][Ltime]
    ft=Ftan[i][Ltime]
    p=Asecs[0].atwist
    fxfy=rotate_loads(fn,0,p)  # Total force on section from FAST
    dfx=fxfy[0]/asecAREA[i]     # Force distributed over the total area of the s
ection
    dfy=fxfy[1]/asecAREA[i]     # Force distributed over the total area of the s
ection
    aerosurface_xyloads.append([dfx,dfy])
    fst.write("%s\n" % ("Blade−1.aerosurf_"+str(i) + ", TRVEC, " + str(dfx) + ", 1,0,0 "
))
    fst.write("%s\n" % ("Blade−1.aerosurf_"+str(i) + ", TRVEC, " + str(dfy) + ", 0,1,0 "
))
fst.write("*Output, field, variable=PRESELECT \n")
fst.write("*Output, history, variable=PRESELECT \n")
fst.write("*End Step")
fst.close()
if plotvals ≡ 1:
    from pylab import *
    plot(columns[0][2:],columns[1][2:])
```

writemats.py

```
outfile.write("%s\n%s\n" % ("** MATERIALS", "**"))

outfile.write("%s\n" % ("*Material, name=triax"))
outfile.write("%s\n%s\n" % ("*Density", " 1850,"))
outfile.write("%s\n%s\n" % ("*Elastic, type=LAMINA", " 27.7e9, 13.65e9, .395, 7.2e9, 7.2e9, 7.2e9"))

outfile.write("%s\n" % ("*Material, name=biax"))
outfile.write("%s\n%s\n" % ("*Density", " 1780,"))
outfile.write("%s\n%s\n" % ("*Elastic, type=LAMINA", " 13.6e9, 13.3e9, .50, 11.8e9, 11.8e9, 11.8e9")
)

outfile.write("%s\n" % ("*Material, name=uniax"))
outfile.write("%s\n%s\n" % ("*Density", " 1920,"))
outfile.write("%s\n%s\n" % ("*Elastic, type=LAMINA", " 41.8e9, 14.0e9, .28, 2.63e9, 2.63e9, 2.63e9")
)

outfile.write("%s\n" % ("*Material, name=foam"))
outfile.write("%s\n%s\n" % ("*Density", " 200,"))
outfile.write("%s\n%s\n" % ("*Elastic", " .256e9, .3"))

outfile.write("%s\n" % ("*Material, name=gelcoat"))
outfile.write("%s\n%s\n" % ("*Density", " 1235,"))
outfile.write("%s\n%s\n" % ("*Elastic", " 3.44e9, .3"))

outfile.write("%s\n" % ("*Material, name=resin"))
outfile.write("%s\n%s\n" % ("*Density", " 1100,"))
outfile.write("%s\n%s\n" % ("*Elastic", " 3.5e9, .3"))
```

writexy−freq.py

```
# −*− coding: mbcs −*−
#
# Abaqus/CAE Release 6.12−2 replay file
# Internal Version: 2012_06_28−23.43.29 119883
# Run by quinlan on Tue Dec 18 15:40:14 2012
#
# from driverUtils import executeOnCaeGraphicsStartup
# executeOnCaeGraphicsStartup()
#: Executing "onCaeGraphicsStartup()" in the site directory ...
from abaqus import *
from abaqusConstants import *
from caeModules import *
from driverUtils import executeOnCaeStartup
executeOnCaeStartup()
############################################################
jobname='AQ100_03_freq.odb'   ### ODB file name goes here
############################################################
o2 = session.openOdb(name=jobname)
session.viewports['Viewport: 1'].setValues(displayedObject=o2)
odb = session.odbs[jobname]
frec=open("recnodes.txt")
lines=frec.readlines()
RecNodes=[]
for ln in lines:
    RecNodes.append(  ln.split()[0] )
session.xyDataListFromField(odb=odb, outputPosition=NODAL, variable=(('U',    N
ODAL, ((COMPONENT, 'U1'), (COMPONENT, 'U2'), )), ), nodeLabels=(( 'BLADE−1', (R
ecNodes )), ))
U=[]
V=[]
for r in RecNodes:
    x = session.xyDataObjects['U:U1 PI: BLADE−1 N: '+r]
    U.append(x)
    y = session.xyDataObjects['U:U2 PI: BLADE−1 N: '+r]
    V.append(y)
session.xyReportOptions.setValues(numDigits=7, layout=SEPARATE_TABLES)
session.writeXYReport(fileName='freq.rpt', appendMode=OFF, xyData=U)
session.writeXYReport(fileName='freq.rpt', appendMode=ON, xyData=V)
```

writexy−standard.py

```
# −*− coding: mbcs −*−
#
# Abaqus/CAE Release 6.12−2 replay file
# Internal Version: 2012_06_28−23.43.29 119883
# Run by quinlan on Thu Nov 29 13:36:01 2012
#
# from driverUtils import executeOnCaeGraphicsStartup
# executeOnCaeGraphicsStartup()
#: Executing "onCaeGraphicsStartup()" in the site directory ...
import sys
from abaqus import *
from abaqusConstants import *
from caeModules import *
from driverUtils import executeOnCaeStartup
executeOnCaeStartup()
f=open("measurenodes.tmp")
ins=f.readlines()
job=ins[0].split()[0]
odbase_name='Section'+job+'.odb'

lead=ins[1].split()[0]
trail=ins[2].split()[0]
print str(job)+','+str(lead)+',' +str(trail)
o1 = session.openOdb(name=odbase_name)
odb = session.odbs[odbase_name]
U=[]
V=[]
# some temporary changes need to be made here.
if job[:2] == '0_':
    bsec = 'BSEC0'
elif job[:3] == '14_':
    bsec ='BSEC14'
else:
    bsec='BSEC'+str(job)
Ulead=session.XYDataFromHistory(name='U−lead', odb=odb, outputVariableName='Spatial
displacement: U1 PI: '+bsec+'−1 Node '+lead+' in NSET MEASURE',
                                        steps=('Step−Torque','Step−Flapwise','Step−Edgewise'
 ), )
Utrail=session.XYDataFromHistory(name='U−trail', odb=odb, outputVariableName='Spatia
l displacement: U1 PI: '+bsec+'−1 Node '+trail+' in NSET MEASURE',
                                        steps=('Step−Torque','Step−Flapwise','Step−Edgewise'
 ), )
Vlead=session.XYDataFromHistory(name='V−lead', odb=odb, outputVariableName='Spatial
displacement: U2 PI: '+bsec+'−1 Node '+lead+' in NSET MEASURE',
                                        steps=('Step−Torque','Step−Flapwise','Step−Edgewise'
 ), )
Vtrail=session.XYDataFromHistory(name='V−trail', odb=odb, outputVariableName='Spatia
l displacement: U2 PI: '+bsec+'−1 Node '+trail+' in NSET MEASURE',
                                        steps=('Step−Torque','Step−Flapwise','Step−Edgewise'
 ), )
mass = session.XYDataFromHistory(name='MASS', odb=odb, outputVariableName='Mass:
MASS for Whole Model', steps=('Step−Torque', ), )

session.xyReportOptions.setValues(numDigits=9, layout=SEPARATE_TABLES)
session.writeXYReport(fileName='Section'+job+'.out', appendMode=OFF, xyData=(mass))
session.writeXYReport(fileName='Section'+job+'.out', appendMode=ON, xyData=(Ulead))
session.writeXYReport(fileName='Section'+job+'.out', appendMode=ON, xyData=(Utrail)
)
session.writeXYReport(fileName='Section'+job+'.out', appendMode=ON, xyData=(Vlead))
session.writeXYReport(fileName='Section'+job+'.out', appendMode=ON, xyData=(Vtrail)
)
```