An Efficient Reduced Order Modeling Method for Analyzing Composite Beams Under Aeroelastic Loading

Ben J. Names

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Aerospace Engineering

Mayuresh J. Patil, Chair
Robert Canfield
Gary D. Seidel

April 19, 2016
Blacksburg, Virginia

Keywords: Cross-Sectional Analysis, Reduced Order Modeling, Composite Beam
Copyright 2016, Ben J. Names

An Efficient Reduced Order Modeling Method for Analyzing Composite Beams Under Aeroelastic Loading

Ben J. Names

(ACADEMIC ABSTRACT)

Composite materials hold numerous advantages over conventional aircraft grade metals. These include high stiffness/strength-to-weight ratios and beneficial stiffness coupling typically used for aeroelastic tailoring. Due to the complexity of modeling composites, designers often select safe, simple geometry and layup schedules for their wing/blade cross-sections. An example of this might be a box-beam made up of 4 laminates, all of which are quasi-isotropic. This results in neglecting more complex designs that might yield a more effective solution, but require a greater analysis effort.

The present work aims to show that the incorporation of complex cross-sections are feasible in the early design process through the use of cross-sectional analysis in conjunction with Timoshenko beam theory. It is important to note that in general, these cross-sections can be inhomogeneous: made up of any number of various materials systems. In addition, these materials could all be anisotropic in nature. The geometry of the cross-sections can take on any shape. Through this reduced order modeling scheme, complex structures can be reduced to 1 dimensional beams. With this approach, the elastic behavior of the structure can be captured, while also allowing for accurate 3D stress and strain recovery. This efficient structural modeling would be ideal in the preliminary design optimization of a wing structure. Furthermore, in conjunction with an efficient unsteady aerodynamic model such as the doublet lattice method, the dynamic aeroelastic stability can also be efficiently captured.

This work introduces a comprehensively verified, open source python API called AeroComBAT (Aeroelastic Composite Beam Analysis Tool). By leveraging cross-sectional analysis, Timoshenko beam theory, and unsteady doublet-lattice method, this package is capable of efficiently conducting linear static structural analysis, normal mode analysis, and dynamic aeroelastic analysis. AeroComBAT can have a significant impact on the design process of a composite structure, and would be ideally implemented as part of a design optimization.

An Efficient Reduced Order Modeling Method for Analyzing Composite Beams Under Aeroelastic Loading

Ben J. Names

(GENERAL AUDIENCE ABSTRACT)

In today's world, composites are still on the cutting edge of aircraft and spacecraft design. Composites have the ability to be lighter and stronger than their equivalent metallic counterparts making them ideal for high performance vehicles. Unfortunately, with higher performance comes higher complexity involved in the analysis. In order to keep maintain simplicity in the design process, safe and simple design are chosen over other options, severely limiting the design space. In order to open up the exploration of the design space, the tool AeroComBAT (Aeroelastic Composite Beam Analysis Tool) was developed. AeroComBAT uses reduced order modeling methods, allowing structural analysis for beam-like composite structures to be quickly iterated over during the design process while maintaining the ability to recover highly accurate stresses and aeroelastic stability of the structure.

I dedicate this work to my parents who were always supportive and patient, and to Rosemary Girard who listened to all of my frustrations and kept an eye on the road ahead for me when I had my blinders on.

Acknowledgements

They say it takes a village to raise a child. The academic corollary of this is that it takes a department to raise a graduate student; possibly in addition to that original village. My effort and focus helped me to get to where I am, but I never would have gotten so far without the help of many individuals.

I would like to first thank my parents Donald Names and Lisa Grosh, as well as my girlfriend Rosemary Girard. These three individuals were always supportive, and would remind me of the big picture when I got too bogged down in the stressful details.

I am grateful of my advisor Dr. Mayuresh Patil. He always made time to see me when I was having issues with my code or struggling to understand a piece of theory. He could consistently take a complex theory and simplify it down into easily digestible concepts. I've enjoyed working with him immensely and I hope to do so again in the future. In addition, it was Dr. Patil's undergraduate "Aircraft Structures" course that solidified my interest in structures, and so I attribute my decision to specialize in structures to his excellent teaching.

I want to thank Dr. Robert Canfield for solidifying many of the fundamentals in fluid-structure interaction in aircraft. In addition, he also helped to reinforce and expand upon the energy method principles I first learned in Aircraft Structures.

I want to thank Dr. Scott Case for the excellent composite courses he taught as they served as the foundation of my composite knowledge. In Dr. Case's"Mechanics of Composite Materials" course, you build on a program throughout the semester. I used the opportunity to learn python, and developed many of the necessary skills that I utilized to program AeroComBAT.

I want to thank Dr. Rakesh Kapania for first exposing my to variational calculus and in general variational methods, which has been one of the most powerful tools I now have at my disposal.

Contents

1 Introduction **1**

2 Cross-Sectional Analysis **4**
2.1 Introduction . . . 4
2.2 Anisotropic Cross-Sectional Analysis . . . 5
2.2.1 Internal Virtual Work . . . 5
2.2.2 External Virtual Work . . . 8
2.2.3 First Variation in the Total Virtual Work . . . 10
2.2.4 Solutions to the Governing Differential Equations . . . 11
2.3 Cross-Sectional Analysis Stiffness Verification . . . 23
2.4 Cross-Sectional Analysis Stress Recovery Verification . . . 31
2.5 Conclusion . . . 43

3 Global Beam Behavior **44**
3.1 Introduction . . . 44
3.2 Linear Static Finite Element Formulation . . . 45
3.2.1 Overcoming Shear Locking . . . 48
3.2.2 Arbitrarily Oriented Beam . . . 51
3.3 Normal Modes Finite Element Formulation . . . 52
3.4 Timoshenko Beam Formulation Verification . . . 54
3.4.1 Linear Static Verification . . . 54
3.4.2 Dynamic Normal Modes Analysis . . . 61
3.5 Conclusion . . . 63

4 Unsteady Aerodynamic Model **64**
4.1 Introduction . . . 64
4.1.1 Brief Overview of the Kernel Function . . . 65
4.2 Implementation of the Doublet Lattice Method . . . 65

4.2.1 The Downwash-Pressure Relation 67
4.2.2 Evaluating the Modified Kernel Function 70
4.3 The Substantial Derivative and Integration Matrices 72
4.3.1 The Substantial Derivative Matrix 72
4.3.2 The Integration Matrix 73
4.4 Doublet Lattice Verification 74
4.5 Conclusion 75

5 Flutter Solution **77**
5.1 Introduction and Background 77
5.1.1 The Non-Iterative PK-Method 79
5.1.2 Mode Tracking 82
5.2 PK-Method Verification 82
5.3 Parametric Composite Study 86

6 Conclusion **92**
6.1 Summary 92
6.2 Future Work 93

AeroComBAT API Documentation **97**

List of Figures

2.1 Vectorial representation of the displacement of point A within a cross-section . . . 6
2.2 The warping displacement subject to $F_x = 1$ . . . 15
2.3 The warping displacement subject to $F_y = 1$ . . . 15
2.4 The warping displacement subject to $F_z = 1$ . . . 16
2.5 The warping displacement subject to $M_x = 1$ . . . 16
2.6 The warping displacement subject to $M_y = 1$ . . . 17
2.7 The warping displacement subject to $M_z = 1$ . . . 17
2.8 First set of cross-sections used to verify AeroComBAT . . . 23
2.9 Diagram of box beam geometry . . . 27
2.10 The cross-sectional mesh used by AeroComBAT . . . 28
2.11 Case B1 Stiffness Results . . . 28
2.12 Case Layup 1 Stiffness Results . . . 29
2.13 Case Layup 2 Stiffness Results . . . 29
2.14 Case Layup 3 Stiffness Results . . . 29
2.15 Cross-section mesh used by AeroComBAT and NASTRAN for 3D stress recovery . . . 31
2.16 Highlighted section shows where stress was sampled in NASTRAN model . . . 32
2.17 Layup 3 σ_{zz} Stress . . . 34
2.18 Layup 3 σ_{zz} AeroComBAT % Error . . . 34
2.19 Layup 3 σ_{xz} Stress . . . 35
2.20 Layup 3 σ_{xz} AeroComBAT % Error . . . 35
2.21 Layup 3 σ_{yz} Stress . . . 36
2.22 Layup 3 σ_{yz} AeroComBAT % Error . . . 36
2.23 Layup 3 σ_{xx} Stress . . . 37
2.24 Layup 3 σ_{xx} AeroComBAT % Error . . . 37
2.25 Layup 3 σ_{xy} Stress . . . 38

2.26 Layup 3 σ_{xy} AeroComBAT % Error 38
2.27 Layup 3 σ_{yy} Stress 39
2.28 Layup 3 σ_{yy} AeroComBAT % Error 39
2.29 The x normal stress in the NASTRAN aluminum box beam under torsion 40
2.30 The x normal stress in the AeroComBAT aluminum box beam under torsion 41
2.31 The warping at the tip of the beam where the torque is applied 42
2.32 The warping at the end of the beam where the beam is constrained 42

3.1 Non-dimensional displacement of a discretized cantilever beam in pure bending 49
3.2 Non-dimensional shear strain of a discretized cantilever beam in pure bending 50
3.3 The finite element displacement error for NASTRAN and AeroComBAT 55
3.4 The finite element rotation error for NASTRAN and AeroComBAT 55
3.5 Displacement error for NASTRAN and AeroComBAT with corrected shear terms 56
3.6 Rotation error for NASTRAN and AeroComBAT with corrected torsion term 57
3.7 Displacement comparison with warp restraint 58
3.8 Rotation comparison with warp restraint 59
3.9 Displacement comparison with incorrectly enforced warping . 60
3.10 Rotation comparison with incorrectly enforced warping 60
3.11 The third natural mode of the solid NASTRAN model exhibits a great amount of dynamic motion within the beam's cross-section 62
3.12 The third natural mode of the solid NASTRAN model exhibits a great amount of dynamic motion within the beam's cross-section 62

4.1 A lifting surface discretized for the doublet-lattice method . . 66
4.2 The detailed geometry of a single doublet panel 66
4.3 Relating the aerodynamic model to the structural model . . . 72
4.4 The geometry of the verification case 74

4.5 Chordwise loading of wing due to first bending mode 75

5.1 The interpolation of the flutter point frequencies at a velocity 81

5.2 The interpolation of the flutter point damping at a velocity . . 81

5.3 Planform view of the wing geometry 82

5.4 Cross-section of the aluminum beam 83

5.5 Damping of flutter modes vs airspeed 84

5.6 Frequency of flutter modes vs airspeed 84

5.7 Composite cross-section for flutter box 86

5.8 Parametric flutter speeds for CUS box beam 87

5.9 Parametric flutter speeds for CAS box beam 88

5.10 Modal damping at $\theta = -23$ degrees 89

5.11 Modal damping at $\theta = -22.9$ degrees 89

5.12 Modal damping at $\theta = -23$ degrees 90

5.13 Modal damping at $\theta = -22.9$ degrees 90

List of Tables

2.1 Cross-section model information 24

2.2 Validation case 1 results 24

2.3 Validation case 2 results 24

2.4 Validation case 3 results 25

2.5 Validation case 4 results 25

2.6 Validation case 5 results 25

2.7 Validation case 6 results 26

2.8 Validation case 7 results 26

2.9 Validation case 8 results 26

2.10 Box Beam Material Properties 27

2.11 Box beam verification layup schedules 27

3.1 Box beam verification layup schedules 61

3.2 Box beam verification layup schedules 62

5.1 Normal Mode Frequency Comparison 83

5.2 Layup schedules for parametric box beam studies 86

Chapter 1

Introduction

With regards to structural analysis, beam models have always been and will continue to be more efficient tools when compared with their shell and solid element counter-parts. This is in part due to the fact that for the lowest order models, a beam element has 12 degrees of freedom (DOF), a quadrilateral laminate element has 20 DOF, and a hexagonal solid element has 24 DOF [20]. In addition, beams end up needing 10s of elements, whereas shells need 100s to 1000s of elements and solids need at least 1000s of elements. Finally, once higher dimensionality is considered, maintaining well conditioned elements with reasonable aspect ratios becomes difficult, often driving up the element count. As such, an orders of magnitude higher number shell or solid elements are required to model a structure, rather than when using beams. Unfortunately, the application of beam theories that most are familiar with such as the traditional Euler-Bernoulli and Timoshenko theories have many short-comings, the most significant being in the material assumptions. In most structural analysis texts when these two theories are used, it is assumed that the beam is made out of a single, isotropic material. There are some rudimentary ad hoc theories in which areas are scaled to account for the difference in material stiffnesses such as the rule of mixtures or stiffness smearing. The application of these methods however is limited. The cross-section shape of these beams is also often limited to simple geometries, such as a square, circle, I-beam, or C-beam, none of which typically accurately models the actual geometry of the design.

As the industry trends towards incorporating more composites into designs, beam formulations are problematic. Traditional ad hoc beam models become less relevant as they are either too simplistic to capture the relevant

physics, and theoretically correct models are too complex to understand and implement. The stress analysis of composite structures today is done almost exclusively with 2D laminate elements, as it is the most efficient element type which can accurately capture the physics within the structure. The biggest problem with 2D laminate elements is that for a large system such as an aircraft or wind-turbine, changing the outer mold line shape of a major component such as a wing is time consuming, making such options expensive. For example, it might be determined in the early design process of a UAV that the chord should be increased by 5%. A small and simple geometry change like this leads to a laborious and time consuming process. This type of change would first require an update of the CAD model, followed by the generation of a completely new FEM. In total this could take hours or even days. If there is no time to do the above, the geometry of the component might have little to no initial structural analysis input. The design may be based on simple, rudimentary closed form solutions in which the mechanical behavior of composite beams cannot be accurately captured. As a result, the wing might be shaped purely due to aerodynamic, controls, systems, or other disciplines constraints. In order to correct this bias, cross-sectional analysis based on reduction of 3D elasticity can be used to reinstate beam theories as viable models.

Two of the most accurate and efficient beam theories that have emerged from a desire to incorporate composites as well as higher-order cross-sectional warping effects into the design process were developed by Hodges [14] and Giavotto et al. [9]. Hodges' beam theory VABS (Variational Asymptotic Beam Sectional analysis) is the more well-known as it is used extensively in industry, and has been highly verified. The formulation of Giavotto et al. was used to develop a code called NABSA (Nonhomogeneous Anisotropic Beam Section Analysis), although it was used less extensively. Recently the formulation was implemented in the development of the code BECAS (Beam Cross-sectional Analysis Software) by Blasques [5]. Despite the excellence both of these tools present, the two have flaws which motivated the current work. Neither VABS nor BECAS are open source, and neither are standalone tools capable of conducting any structural analysis. In order to solve this problem, the completely open-source python module AeroComBAT (Aeroelastic Composite Beam Analysis Tool) was developed, leveraging the cross-sectional analysis formulation first presented by Giavotto et al., the Timoshenko beam finite element formulation presented by Reddy [20], and the doublet lattice formulation first proposed by Rodden and Albano [2]. This module has been

extensively verified against results from BECAS, VABS, NABSA, and NX NASTRAN for its capabilities in cross-sectional analysis, static and normal mode beam analysis, 3D stress recovery and aeroelastic dynamic stability. In the following chapters, the derivations for the cross-sectional analysis formulation and the 3D Timoshenko beam finite element formulation will be discussed. Furthermore a technical description of the implementation of the doublet lattice method will be provided as well as the the procedure for solving the dynamic aeroelastic stability problem. At the end of each chapter, the validation cases and results produced by AeroComBAT will be presented.

Chapter 2

Cross-Sectional Analysis

2.1 Introduction

When one conducts a beam analysis, two implicit assumptions are made. The first is that the structure resembles a beam geometrically, i.e., one dimension of the structure is much larger than the other two. The second more subtle assumption is that the local behavior of the beam (such as the stress and strain within the beam cross-section) is dominated by the global beam behavior. Consider the Euler-Bernoulli beam theory, one of the pillars of undergraduate structural engineering courses. The governing differential equation of this theory is:

$$\frac{d^2}{dz^2}\left(EI_{yy}\frac{d^2u}{dz^2}\right) = p_x \tag{2.1}$$

where u is the transverse displacement, p_x is the distributed transverse load, and EI_{yy} is the bending stiffness. In this case, the global beam behavior is the transverse displacement of the beam u, and the strain within the cross-section of the beam is the local (or complete 3D) behavior. From equation 2.1, it is clear where the global behavior of the beam is expressed. The relation between the local and global behavior is not as clear. Recall that in the derivation, we assume that the strain in the beam is proportional to the radius of curvature in the beam:

$$\epsilon_{zz} = -\frac{x}{\rho} \tag{2.2}$$

where the radius of curvature ρ can be expressed as:

$$\frac{1}{\rho} = \frac{d^2u}{dz^2} \tag{2.3}$$

By combining equations 2.2 and 2.3, we get:

$$\epsilon_{zz} = -x\frac{d^2u}{dz^2} \tag{2.4}$$

It is clear now using equation 2.4 to see how the local behavior of the beam (in this case the strain ϵ_{zz}) is expressed completely using only the global behavior of the beam (in this case the transverse displacement u). All cross-sectional analysis, when employed in conjunction with a beam theory is a way of describing the local 3D behavior in the beam cross-section as a function of the global 1D behavior. In the following section, a comprehensive explanation of the derivation first carried out by Giavotto et al. [9] and then later by Blasques [5] will be conveyed.

2.2 Anisotropic Cross-Sectional Analysis

In order to link the global beam behavior to its local cross-sectional behavior, Giavotto et al. [9] and Blasques [5] used the principal of virtual work in order to derive the equilibrium equations of the cross-section. By drawing connections between how the internal loads and warping vary along the length of the beam, the equilibrium equations can be solved, yielding the stress and stiffness properties of the cross-section.

Recall that the general procedure when using the principal of virtual work is to derive an expression for the first variation in the total virtual work in the system.

$$\delta W_{total} = \delta W_{ext} - \delta W_{int} \tag{2.5}$$

Setting the variation of the total work equal to zero, the virtual work is minimized in the system due to variations in the displacement field. The internal and external virtual work are derived in the sections below.

2.2.1 Internal Virtual Work

In order to compose the internal strain energy in the beam, the displacement field within the cross-section of the beam must first be derived. This can be

done as:

$$\{s\} = \{v\} + \{g\} \tag{2.6}$$

where s is the total displacement of the point within the cross-section, v is the cross-section rigid body displacement, and g is the local warping of the cross-section. Already a distinction has been made between the global behavior (v) and the local behavior (g). This can also be seen below in figure 2.1. The rigid body displacement v can be rewritten as:

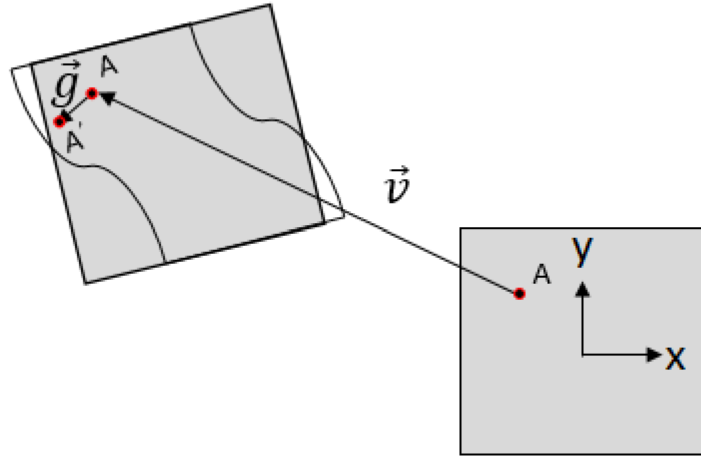


Figure 2.1: Vectorial representation of the displacement of point A within a cross-section

$$v = Zr \tag{2.7}$$

where $r = \{\chi_x \quad \chi_y \quad \chi_z \quad \psi_x \quad \psi_y \quad \psi_z\}^T$ are the three rigid body displacements and rotations of the cross-section reference point. Additionally, $Z = \begin{bmatrix} I_3 & n^T \end{bmatrix}$ where n is:

$$n = \begin{bmatrix} 0 & 0 & y \\ 0 & 0 & -x \\ -y & x & 0 \end{bmatrix} \tag{2.8}$$

and I_3 is a 3x3 identity matrix. Using the infinitesimal strain tensor:

$$\epsilon_{ij} = \frac{1}{2}(s_{i,j} + s_{j,i}) \tag{2.9}$$

expressions for all six strains can be written in matrix form as:

$$\{\epsilon\} = [B]\{s\}+[S]\{s_{,3}\} = [B][Z]\{r\}+[S][Z]\left\{\frac{\partial r}{\partial z}\right\}+[B]\{g\}+[S]\frac{\partial\{g\}}{\partial z} \quad (2.10)$$

where

$$[B] = \begin{bmatrix} \partial/\partial x & 0 & 0 \\ 0 & \partial/\partial y & 0 \\ \partial/\partial x & \partial/\partial y & 0 \\ 0 & 0 & \partial/\partial x \\ 0 & 0 & \partial/\partial y \\ 0 & 0 & 0 \end{bmatrix} \quad (2.11)$$

and

$$[S] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.12)$$

It can also be shown that $[B][Z] = [S][Z][T_r]$ where

$$[T_r] = \begin{bmatrix} 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (2.13)$$

Using the simplification in equation 2.13, the strain can be written as:

$$\{\epsilon\} = [S][Z]\{\psi\} + [B]\{g\} + [S]\frac{\partial\{g\}}{\partial z} \quad (2.14)$$

where $\{\psi\} = [T_r]\{r\} + \frac{\partial\{r\}}{\partial z}$. Less formally, ψ is a column vector of the beam section strains. Written in this form, it is clear from equation 2.14 that the strain anywhere in the cross-section has three contributions. The first term is the strain caused by a difference in the rigid body displacements between two adjacent cross-sections. These are the strains that are typically considered in most traditional beam theory applications. To be more explicit, this strain

originates form the rigid cross-section displacement v. The second term is the strain caused by cross-section warping, and the third term is the strain caused by a difference in warping displacements between two adjacent cross-sections. With the strain in the beam fully described, the expression for the internal virtual work per unit length in the beam can be composed:

$$\delta W'_{int} = \int \delta\epsilon^T \sigma dA = \int \delta\epsilon^T\ Q\epsilon dA \tag{2.15}$$

Substituting the strain from equation 2.14 into the internal virtual work in equation 2.15 results in:

$$\delta W'_{int} = \int_A \left(\delta\psi^T Z^T S^T + \delta g^T B^T + \delta\frac{\partial g}{\partial z}^T S^T\right) Q \left(SZ\psi + Bg + S\frac{\partial g}{\partial z}\right) dA \tag{2.16}$$

By discretizing the cross-section surface using finite element formulation:

$$g(x, y, z) = N_i(x, y)u_i \tag{2.17}$$

where $N_i(x, y)$ is a matrix of shape functions and u_i is a vector of nodal warping displacements. The warping displacement discretization from equation 2.17 can be substituted into the strain energy expression 2.16, and can be written in matrix form as:

$$\delta W'_{int} = \int \delta\epsilon^T \sigma dA = \begin{bmatrix} \delta u \\ \delta\psi \\ \delta\frac{\partial u}{\partial z} \end{bmatrix}^T \begin{bmatrix} E & R & C \\ R^T & A & L^T \\ C^T & L & M \end{bmatrix} \begin{bmatrix} u \\ \psi \\ \frac{\partial u}{\partial z} \end{bmatrix} \tag{2.18}$$

where:

$$\begin{aligned} \underset{(6\times6)}{A} &= \textstyle\int_A Z^T S^T Q S Z dxdy & \underset{(n_d\times6)}{R} &= \textstyle\int_A N^T B^T Q S Z dxdy \\ \underset{(n_d\times n_d)}{E} &= \textstyle\int_A N^T B^T Q B N dxdy & \underset{(n_d\times n_d)}{C} &= \textstyle\int_A N^T B^T Q S N dxdy \\ \underset{(n_d\times6)}{L} &= \textstyle\int_A N^T S^T Q S Z dxdy & \underset{(n_d\times n_d)}{M} &= \textstyle\int_A N^T S^T Q S N dxdy \end{aligned} \tag{2.19}$$

2.2.2 External Virtual Work

In order to use principle of virtual work, the external virtual work must also be derived. When the beam is loaded, internal stresses develop within the beam. When looking at a differential slice of the beam (i.e. the beam cross-section), those internal stresses can be viewed as external tractions action on

the beam cross-section. These surface tractions are $p = \{\sigma_{xz} \quad \sigma_{yz} \quad \sigma_{zz}\}^T$. As such, the external virtual work per unit length is:

$$\delta W'_{ext} = \int_A \frac{\partial(\delta s^T p)}{\partial z} dA \tag{2.20}$$

Recall that the expression for the displacement of any point in the cross-section can now be expressed as:

$$s = v + g = Zr + Nu \tag{2.21}$$

Expanding equation 2.20 with equation 2.21 results in:

$$\delta W'_{ext} = \delta\frac{\partial r^T}{\partial z}\int_A Z^T p dA + \delta r^T \int_A Z^T \frac{\partial p}{\partial z} dA + \delta\frac{\partial u^T}{\partial z}\int_A N^T p dA + \delta u^T \int_A N^T \frac{\partial p}{\partial z} dA \tag{2.22}$$

This expression can be initially simplified recalling that integrating the tractions over the surface area of the cross-section yields a set of 3 force and 3 moment resultants acting on the cross-section, which are:

$$F = \int_A p dA \qquad M = \int_A n^T p dA \tag{2.23}$$

Combining these two equations, we can get:

$$\Theta = \int_A Z^T p dA \tag{2.24}$$

where Θ is the vector of equivalent internal forces and moments acting on the cross-section. Additionally, a similar simplification can be applied to the work terms containing the tractions and warping displacements:

$$P = \int_A N^T p dA \qquad \frac{\partial P}{\partial z} = \int_A N^T \frac{\partial p}{\partial z} dA \tag{2.25}$$

Using equations 2.24 and 2.25, the area integrations in the equations 2.22 can be carried out reducing the external virtual work to:

$$\delta W'_{ext} = \delta\frac{\partial r^T}{\partial z}\Theta + \delta r^T \frac{\partial \Theta}{\partial z} + \delta\frac{\partial u^T}{\partial z} P + \delta u^T \frac{\partial P}{\partial z} \tag{2.26}$$

In order to introduce the section strain parameter into the external virtual work expression, recall that:

$$\psi = T_r r + \frac{\partial r}{\partial z} \tag{2.27}$$

By adding and subtracting ψ from equation 2.26 and simultaneously using equation 2.27, the external virtual work can be simplified to:

$$\delta W'_{ext} = \delta r^T \frac{\partial \Theta}{\partial z} - \delta r^T T_r^T \Theta + \delta \psi^T \Theta + \delta \frac{\partial u^T}{\partial z} P + \delta u^T \frac{\partial P}{\partial z} \tag{2.28}$$

Written in matrix form, equation 2.28 becomes:

$$\delta W'_{ext} = \begin{bmatrix} \delta u \\ \delta \psi \\ \delta \frac{\partial u}{\partial z} \end{bmatrix}^T \begin{bmatrix} \frac{\partial P}{\partial z} \\ \theta \\ P \end{bmatrix} + \delta r^T \left(\frac{\partial \theta}{\partial z} - T_r^T \theta \right) \tag{2.29}$$

2.2.3 First Variation in the Total Virtual Work

Recalling equation 2.5, the total virtual work in the system in matrix form is:

$$\delta W'_{total} = \begin{bmatrix} \delta u \\ \delta \psi \\ \delta \frac{\partial u}{\partial z} \end{bmatrix}^T \begin{bmatrix} \frac{\partial P}{\partial z} \\ \theta \\ P \end{bmatrix} + \delta r^T \left(\frac{\partial \Theta}{\partial z} - T_r^T \Theta \right) - \begin{bmatrix} \delta u \\ \delta \psi \\ \frac{\partial u}{\partial z} \end{bmatrix}^T \begin{bmatrix} E & R & C \\ R^T & A & L^T \\ C^T & L & M \end{bmatrix} \begin{bmatrix} u \\ \psi \\ \frac{\partial u}{\partial z} \end{bmatrix} \tag{2.30}$$

At this point it is convenient to factor out variational terms and set the total virtual work equal to zero in order to get the governing differential equations:

$$\delta W'_{total} = \begin{bmatrix} \delta u \\ \delta \psi \\ \delta \frac{\partial u}{\partial z} \end{bmatrix}^T \left(\begin{bmatrix} \frac{\partial P}{\partial z} \\ \theta \\ P \end{bmatrix} - \begin{bmatrix} E & R & C \\ R^T & A & L^T \\ C^T & L & M \end{bmatrix} \begin{bmatrix} u \\ \psi \\ \frac{\partial u}{\partial z} \end{bmatrix} \right) + \delta r^T \left(\frac{\partial \Theta}{\partial z} - T_r^T \Theta \right) = 0 \tag{2.31}$$

In order for the variation in the total virtual work to be set to be equal to zero without the the trivial case of setting the variations equal to zero, the quantities within the two sets of parenthesis must both be independently equal to zero. These become the four governing differential equations:

$$\begin{gathered} Eu + R\psi + C\frac{\partial u}{\partial z} = \frac{\partial P}{\partial z} \\ R^T u + A\psi + L^T \frac{\partial u}{\partial z} = \Theta \\ C^T u + L\psi + M\frac{\partial u}{\partial z} = P \\ \frac{\partial \Theta}{\partial z} = T_r^T \Theta \end{gathered} \tag{2.32}$$

These differential equations can be simplified further by differentiating the third equation with respect to z and setting it equal to the first equation,

resulting in the final governing differential equations of the cross-section.

$$\begin{aligned} M\tfrac{\partial^2 u}{\partial z^2} + (C^T - C)\tfrac{\partial u}{\partial z} + L\tfrac{\partial \psi}{\partial z} - Eu - R\psi = 0 \\ L^T\tfrac{\partial u}{\partial z} + R^T u + A\psi = \Theta \\ \tfrac{\partial \Theta}{\partial z} = T_r^T \Theta \end{aligned} \tag{2.33}$$

2.2.4 Solutions to the Governing Differential Equations

The governing differential equilibrium equations of the cross-section have two types of solutions. The first less frequently used type are called the extremity solutions. The second more useful are the central solutions.

The extremity solution

First coined by Giavotto et al. [9], the extremity solutions correspond to the solutions of the governing equations when the internal forces and moments are all equal to zero: $\Theta = 0$. Assuming exponential solutions of the type:

$$u = \tilde{u}e^{\lambda z} \qquad \psi = \tilde{\psi}e^{\lambda z} \tag{2.34}$$

The governing differential equations 2.33 become the eigenvalue problem:

$$\left(\lambda^2 \begin{bmatrix} M & 0 \\ 0 & 0 \end{bmatrix} + \lambda \begin{bmatrix} (C^T - C) & L \\ -L^T & 0 \end{bmatrix} - \begin{bmatrix} E & R \\ R^T & A \end{bmatrix}\right) \begin{bmatrix} \tilde{u} \\ \tilde{\psi} \end{bmatrix} = 0 \tag{2.35}$$

The eigenvectors correspond to the modal warping shapes of the cross-section, while the eigenvalues correspond to the wavelengths of the mode shapes. Referring back to the assumed solutions, it might become clear that the extremity solutions are an expression of St. Venants principle [22].Typically this principle is used to justify the simplification of tractions on a body to equivalent forces and moments. Interestingly in this case the central solutions are a description of how quickly local displacement fields decay over the length of the beam.

The central solution

The central solution is the second type of solution that can be achieved through this cross-sectional analysis approach, and it corresponds to the solution of the equilibrium equations for any general $\Theta \neq 0$. In order to gather

more information on how the force and moment resultants vary through the length of the beam, the equilibrium equations are differentiated:

$$\frac{\partial^2 \Theta}{\partial z^2} = 0 \tag{2.36}$$

Through this, we can glean that the force and moment resultants vary at most linearly in z. This process can be repeated on the warping equilibrium equations. This reveals that the nodal warping u and the section strains ψ vary at most linearly with z. Having determined that Θ, u, and ψ all vary linearly with z, it is convenient to relate these parameters with equation 2.37

$$\begin{array}{ccc} u = X\Theta & , & \frac{\partial u}{\partial z} = \frac{\partial X}{\partial z}\Theta \\ \psi = Y\Theta & , & \frac{\partial \psi}{\partial z} = \frac{\partial Y}{\partial z}\Theta \end{array} \tag{2.37}$$

where X and Y are the nodal warping and section strain mode shapes associated with a particular force or moment resultant. For further details on how the conclusions of equation 2.36 and 2.37 were reached, please see Blasques' BECAS Theory Manual [5]. These mode shapes and their gradients are unknown. Having determined that the warping displacements are at most linear in z, the equilibrium equations simplify to:

$$\begin{gathered} (C^T - C)\frac{\partial X}{\partial z}\Theta + L\frac{\partial Y}{\partial z}\Theta - EX\Theta - RY\Theta = 0 \\ L^T\frac{\partial X}{\partial z}\Theta + R^T X\Theta + AY\Theta = \Theta \\ \frac{\partial \Theta}{\partial z} = T_r^T\Theta \end{gathered} \tag{2.38}$$

With Θ, u, and ψ being linear in z, it is pertinent to write the first derivative of the equilibrium equations with respect to z as well:

$$\begin{gathered} E\frac{\partial X}{\partial z}\Theta + R\frac{\partial Y}{\partial z}\Theta = 0 \\ R^T\frac{\partial X}{\partial z}\Theta + A\frac{\partial Y}{\partial z}\Theta = \frac{\partial \Theta}{\partial z} \end{gathered} \tag{2.39}$$

Displacement Redundancy

A point within the cross-section has no way of differentiating between its translation due to warping or due to the rigid translation and rotation of the cross-section. The cross-section's rigid body translations can be filtered out

by ensuring that the average translations experienced by the cross-section are equal to zero:

$$\sum_{i=1}^{n} u_{x_i} = 0 \qquad \sum_{i=1}^{n} u_{y_i} = 0 \qquad \sum_{i=1}^{n} u_{z_i} = 0 \tag{2.40}$$

where n is the number of nodes in the cross-section. In order to ensure the cross-section's rigid body rotations are filtered out, a similar approach can be taken:

$$\sum_{i=1}^{n} -z_i u_{y_i} + y_i u_{z_i} = 0 \qquad \sum_{i=1}^{n} z_i u_{x_i} - x_i u_{z_i} = 0 \qquad \sum_{i=1}^{n} -y_i u_{x_i} + x_i u_{y_i} = 0 \tag{2.41}$$

It should be noted that since we define the cross-section to exist in the $x-y$ plane, z_i will always equal zero. As such, equations 2.41 simplifies to:

$$\sum_{i=1}^{n} y_i u_{z_i} = 0 \qquad \sum_{i=1}^{n} -x_i u_{z_i} = 0 \qquad \sum_{i=1}^{n} -y_i u_{x_i} + x_i u_{y_i} = 0 \tag{2.42}$$

The integral constraints in equations 2.40 and 2.42 can be applied in matrix form as:

$$[D]^T \{u\} = 0 \tag{2.43}$$

where:

$$D = \begin{bmatrix} 1 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 1 \\ 0 & 0 & y_1 & \dots & 0 & 0 & y_n \\ 0 & 0 & -x_1 & \dots & 0 & 0 & -x_n \\ -y_1 & x_1 & 0 & \dots & -y_n & x_n & 0 \end{bmatrix}^T \tag{2.44}$$

This additional set of 6 equations will ensure that the warping mode shapes X will have no rigid body translation or rotation contributions.

Solving for the Warping and Section-Strain Mode Shapes

The cross-sectional analysis derivation is fairly long and complex, so it is important to remember the goals of the derivation, and the steps that have been taken to achieve them. Recall that the full, 3D strain state at any point within the cross-section can be expressed using equation 2.14. In order to

apply this equation at the beginning of the derivation, we needed to know ψ, u, and $\frac{\partial u}{\partial z}$. Now using equation 2.37, these three parameters can all be expressed as linear combinations of Θ and their respective solutions which we call mode shapes. Therefore the full strain state at any point within the cross-section can be determined by substituting equations 2.37 into equation 2.14 resulting in:

$$\{\epsilon\} = [S][Z][Y]\{\Theta\} + [B][N][X]\{\Theta\} + [S][N]\left[\frac{\partial X}{\partial z}\right]\{\Theta\} \tag{2.45}$$

Equation 2.45 not only allows for the strain (and of course stress) to be recovered within the cross-section, but also to define the cross-section compliance matrix. In order to solve for the mode shapes (X, Y, $\frac{\partial X}{\partial z}$, and $\frac{\partial Y}{\partial z}$), the equilibrium equations from equations 2.38 and 2.39 must be solved in conjunction with the constraint equations 2.40 and 2.41. In matrix form, the two systems of equations result in:

$$\begin{bmatrix} E & R & D \\ R^T & A & 0 \\ D^T & 0 & 0 \end{bmatrix} \begin{Bmatrix} \frac{\partial X}{\partial z} \\ \frac{\partial Y}{\partial z} \\ \Lambda \end{Bmatrix} = \begin{Bmatrix} 0 \\ T_r^T \\ 0 \end{Bmatrix} \tag{2.46}$$

$$\begin{bmatrix} E & R & D \\ R^T & A & 0 \\ D^T & 0 & 0 \end{bmatrix} \begin{Bmatrix} X \\ Y \\ \Lambda \end{Bmatrix} = \begin{bmatrix} (C^T - C) & L \\ L^T & 0 \\ 0 & 0 \end{bmatrix} \begin{Bmatrix} \frac{\partial X}{\partial z} \\ \frac{\partial Y}{\partial z} \end{Bmatrix} + \begin{Bmatrix} 0 \\ I_6 \\ 0 \end{Bmatrix} \tag{2.47}$$

While the mode shapes have been thoroughly discussed, their nature may still be confusing to the reader. For example, recall that the matrix X contains 6 different column vectors associated with cross-section warping displacement mode shapes. Each one of these column vectors corresponds to the displacement field caused by applying one of the six possible force or moment resultants to the cross-section. In order to add some additional physical significance, below are six figures portraying each of the possible warping displacement mode shapes of thin-walled aluminum box beam cross-section. Note that while these mode shapes might share similarities with the warping mode shapes of other cross-sections, they are unique to this cross-section. The displacements have also been scaled up for ease of visibility.

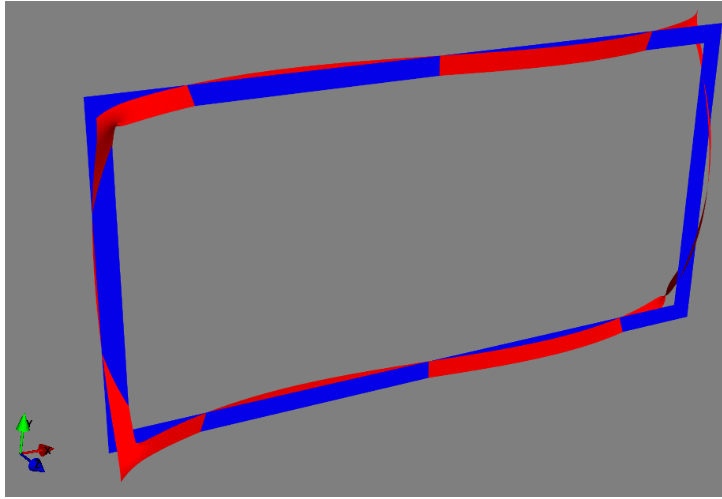


Figure 2.2: The warping displacement subject to $F_x = 1$

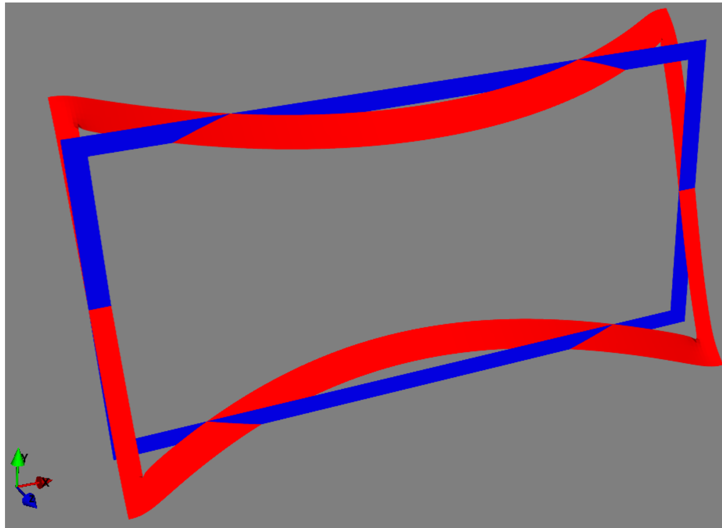


Figure 2.3: The warping displacement subject to $F_y = 1$

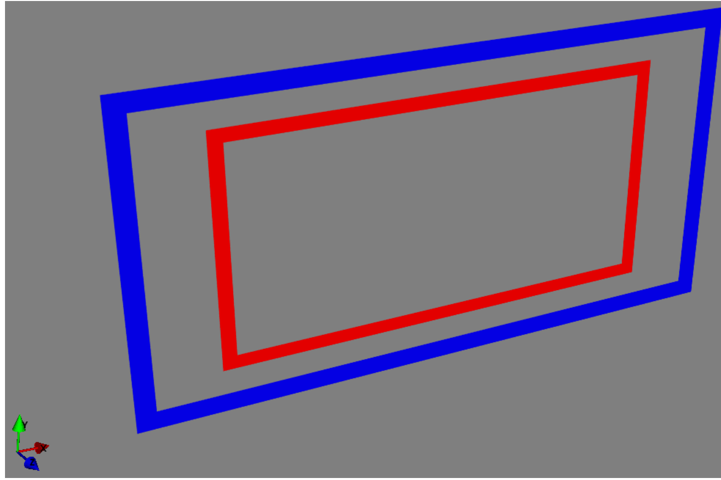


Figure 2.4: The warping displacement subject to $F_z = 1$

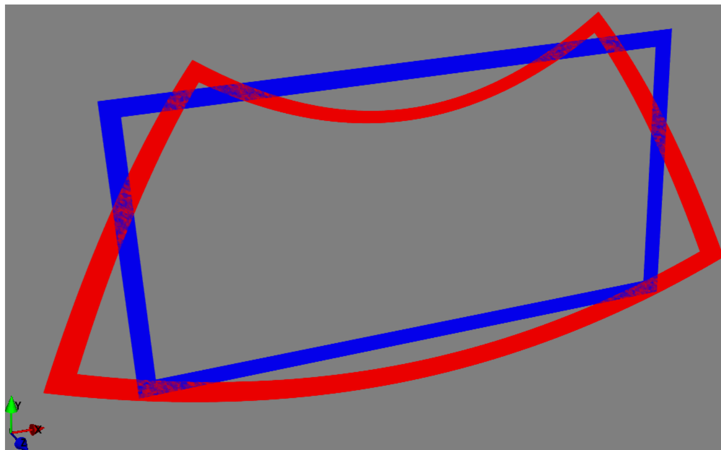


Figure 2.5: The warping displacement subject to $M_x = 1$

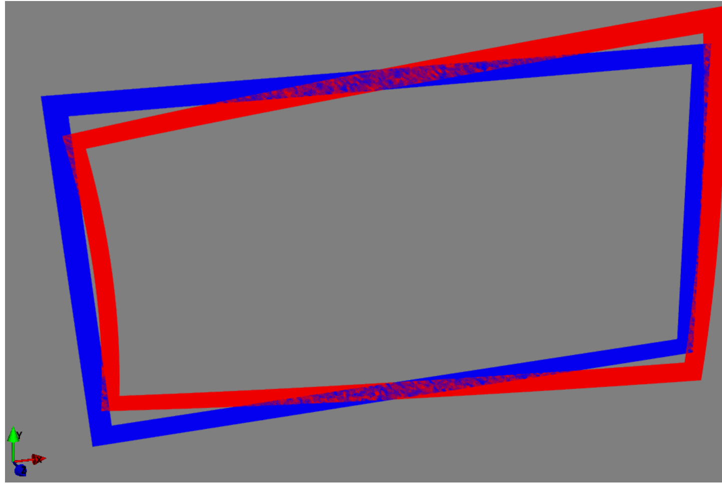


Figure 2.6: The warping displacement subject to $M_y = 1$

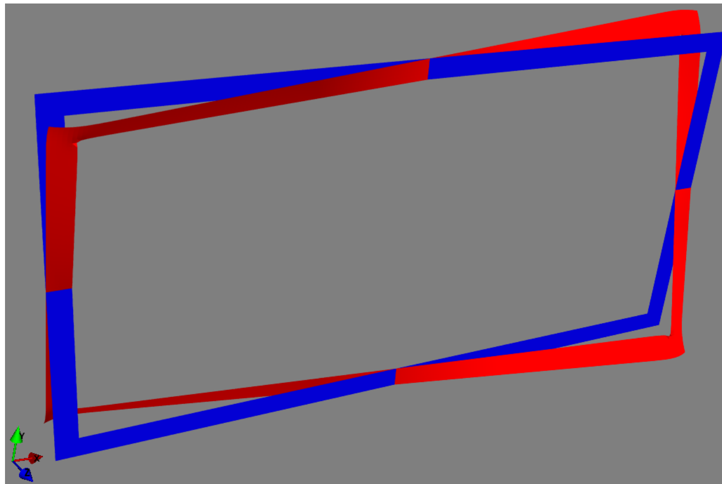


Figure 2.7: The warping displacement subject to $M_z = 1$

Construction of the Cross-Sectional Compliance Matrix

The cross-section compliance matrix is the matrix that relates the equivalent forces and moments to the beam generalized strains and curvatures. In order to determine the cross-section compliance matrix, the principle of virtual work must be applied once more. Using the new definition of the strain as a

function of only Θ, the internal virtual work can be expressed as:

$$\delta W'_{int} = \int_A \delta\Theta \left(([S][Z][Y])^T + ([B][N][X])^T + \left([S][N]\frac{\partial X}{\partial z}\right)^T \right) Q \left([S][Z][Y] + [B][N][X] + [S][N]\frac{\partial X}{\partial z}\right) \Theta dA \tag{2.48}$$

Written in matrix form with the area integration evaluated, the internal virtual work is:

$$\delta W'_{int} = \{\delta\Theta\}^T \begin{bmatrix} X \\ \frac{\partial X}{\partial z} \\ Y \end{bmatrix}^T \begin{bmatrix} E & C & R \\ C^T & M & L \\ R^T & L^T & A \end{bmatrix} \begin{bmatrix} X \\ \frac{\partial X}{\partial z} \\ Y \end{bmatrix} \{\Theta\} \tag{2.49}$$

Since only linear elastic materials are being considered, it can be shown that the internal virtual work is equal to the complimentary virtual work:

$$\delta W'_{int} = \delta W'^{*}_{int} \Rightarrow \int_A \sigma^T \delta\epsilon dA = \delta\Theta^T F_s \Theta \tag{2.50}$$

Equating the internal virtual work with the complementary internal virtual work gives equation 2.51. By equating these two quantities, it is possible to pick out the cross-section compliance matrix.

$$\delta\Theta^T F_s \Theta = \{\delta\Theta\}^T \begin{bmatrix} X \\ \frac{\partial X}{\partial z} \\ Y \end{bmatrix}^T \begin{bmatrix} E & C & R \\ C^T & M & L \\ R^T & L^T & A \end{bmatrix} \begin{bmatrix} X \\ \frac{\partial X}{\partial z} \\ Y \end{bmatrix} \{\Theta\} \tag{2.51}$$

Explicitly stated, the section compliance matrix is:

$$F_s = \begin{bmatrix} X \\ \frac{\partial X}{\partial z} \\ Y \end{bmatrix}^T \begin{bmatrix} E & C & R \\ C^T & M & L \\ R^T & L^T & A \end{bmatrix} \begin{bmatrix} X \\ \frac{\partial X}{\partial z} \\ Y \end{bmatrix} \tag{2.52}$$

Having derived the compliance matrix the cross-sectional stiffness matrix K_s can be calculated as:

$$K_s = F_s^{-1} \tag{2.53}$$

Additional Cross-Section Characteristics

From the cross-sectional compliance matrix, the location of the shear center within the cross-section can also be determined. This location is important as it is the location where the shear and torsion effects are decoupled. This location is also important as it is the point about which the cross-section rotates due to torsion [14]. In order to determine the x and y coordinates of this point within the cross-section plane, let us apply the two possible transverse loads (F_x and F_y) knowing that at the shear center, the resultant curvature $\kappa_z = 0$. The three moments that arise due to the transverse loading are:

$$M_x = -F_y(L-z) \qquad M_y = F_x(L-z) \qquad M_z = -F_x y_{sc} + F_y x_{sc} \quad (2.54)$$

Multiplying the force and moment resultant ($\Theta = \{F_x, F_y, 0, M_x, M_y, M_z\}$) by the cross-section compliance matrix yields a set of generalized beam strains and curvatures. In order to determine when the shear and torsion behavior uncouples, the only curvature of interest in this case is κ_z. Therefore the resulting equation for the torsion curvature is:

$$\kappa_z = [F_{16} + F_{26}(L-z) - F_{66} y_{sc}]F_x + [F_{26} - F_{46}(L-z) + F_{66} x_{sc}]F_y = 0 \quad (2.55)$$

In order for equation 2.55 to be true regardless of the values of F_x and F_y, then the quantities multiplying F_x and F_y must be independently zero. Therefore:

$$x_{sc} = -\frac{F_{26} + F_{46}(L-z)}{F_{66}} \quad , \quad y_{sc} = \frac{F_{16} + F_{56}(L-z)}{F_{66}} \qquad (2.56)$$

Notice that as Hodges points out, if F_{46} and F_{56} are not zero (and there is torsion-bending coupling), then the shear center location is not truly a cross-section property as it will vary along the length of the beam. As such it is important to pay attention to the magnitude of F_{46} and F_{56} compared with F_{16}, F_{26}, and F_{66} in order to determine if the shear center can be considered a cross-section property. For circumferentially uniform cross-sections (CUS) F_{46} and F_{56} are zero and the shear center can be considered a cross-sectional property. Another important cross-section location is the tension center. This is the point about which if a tensile force is applied, it produces no moments within the cross-section. This is also the point about which bending

moment rotations are taken for the cross-section. Applying only a tension force F_z to the cross-section at some x_t, y_t, the resulting moments are:

$$M_x = F_z y_t \quad , \quad M_y = -F_z x_t \tag{2.57}$$

Multiplying the force and moment resultant ($\Theta = \{0, 0, F_z, M_x, M_y, 0\}$) by the cross-section compliance matrix yields the set of generalized beam strains and curvatures. In order to determine when the tension and bending behavior uncouples, the only curvatures of interest in this case is κ_x and κ_y. Therefore the resulting equation for the bending curvatures are:

$$\kappa_x = F_z[F_{34} + F_{44} y_t - F_{45} x_t] = 0 \quad , \quad \kappa_y = F_z[F_{35} + F_{45} y_t - F_{55} x_t] = 0 \tag{2.58}$$

For equations 2.58 to be true for any force F_z, then the the quantities multiplying F_z in both cases must always be zero. Therefore the tension center coordinates can then be defined as:

$$x_t = -\frac{F_{34}F_{45} - F_{35}F_{44}}{F_{44}F_{55} - F_{45}^2} \quad , \quad y_t = -\frac{F_{34}F_{55} - F_{35}F_{45}}{F_{44}F_{55} - F_{45}^2} \tag{2.59}$$

Transforming the Stiffness Matrix to a Desired Location

The reference axis of a cross-section is a important location although for many simple beam theory applications this point is often only implicitly taken into account. It is the control point for the cross-section. This is the point about which forces are applied to the cross-section as well as where global beam displacements are found.

When one calculates the bending stiffness for an Euler-Bernoulli beam, the second area moment (say I_{xx}) must be calculated. The formula for the second area moment is:

$$I_{xx} = \int\int y^2 dA \tag{2.60}$$

Where is this integral taken about? Most would evaluate this about the mass centroid, although in reality it could be taken about any point within the cross-section's plane. If true that this integral could be evaluated anywhere, why do so many choose the mass centroid? Often for symmetric homogeneous cross-sections, the mass center also coincides with the shear center as well as the tension center. As such both the static and dynamic behavior of a beam is simple when the mass and stiffness characteristics are taken about this point. In calculating the cross-sectional analysis integral in equation

2.60 about the mass center, it was implicitly stated that the reference axis for the cross-section would be the mass center. After one has completed the cross-sectional analysis laid out by Giavotto et al. [9], where is the reference axis of the beam? It was implicitly chosen as the origin of the x-y plane in which the cross-section is defined. If we wanted to ensure the reference axis was located at the mass center as in the previous example, we would position the cross-section mass center at the origin of the x-y plane and carry out the cross-sectional analysis.

Unfortunately this is really only a "quick fix" and doesn't really address the problem at hand. Suppose one wants to set the reference axis to be the shear center or tension center? If the cross-section is not thin-walled, little if any closed form solutions exist for the locations of these points. A simple brute force solution would be to conduct the cross-sectional analysis twice. Once to determine the location of say the shear center, and then once to get the cross-section stiffness matrix about the shear center. The much more elegant solution would be to transform the cross-section stiffness matrix from the origin to the desired reference axis point. Consider what the cross-sectional stiffness matrix does. Provided the beam generalized strains at the origin of the x-y plane, it can determine what the required generalized forces are to produce those generalized strains:

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{14} & K_{15} & K_{16} \\ K_{12} & K_{22} & K_{23} & K_{24} & K_{25} & K_{26} \\ K_{13} & K_{23} & K_{33} & K_{34} & K_{35} & K_{36} \\ K_{14} & K_{24} & K_{34} & K_{44} & K_{45} & K_{46} \\ K_{15} & K_{25} & K_{35} & K_{45} & K_{55} & K_{56} \\ K_{16} & K_{26} & K_{36} & K_{46} & K_{56} & K_{66} \end{bmatrix} \begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \kappa_x \\ \kappa_y \\ \kappa_z \end{Bmatrix} = \begin{Bmatrix} F_x \\ F_y \\ F_z \\ M_x \\ M_y \\ M_z \end{Bmatrix} \tag{2.61}$$

In order to transform the stiffness matrix to another point, the generalized forces and strains must first be transformed. To move the generalized forces from the original reference axis to another reference point, the following equation can be used:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -y_{ref} & 1 & 0 & 0 \\ 0 & 0 & x_{ref} & 0 & 1 & 0 \\ y_{ref} & -x_{ref} & 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} F_x \\ F_y \\ F_z \\ M_x \\ M_y \\ M_z \end{Bmatrix}_0 = \begin{Bmatrix} F_x \\ F_y \\ F_z \\ M_x \\ M_y \\ M_z \end{Bmatrix}_{ref} \tag{2.62}$$

To transform the generalized beam strains from the original reference axis point to another reference point a similar approach can be used:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & -y_{ref} \\ 0 & 1 & 0 & 0 & 0 & x_{ref} \\ 0 & 0 & 1 & y_{ref} & -x_{ref} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \kappa_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}_0 = \begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \kappa_x \\ \kappa_y \\ \kappa_z \end{Bmatrix}_{ref} \tag{2.63}$$

Substituting equations 2.63 and 2.62 into equation 2.61, the stiffness matrix itself can be transformed such that:

$$[K_{ref}]_s = [T_2]^{-1}[K]_s[T_1] \tag{2.64}$$

where T_1 is the transformation matrix from equation 2.63 and T_2 is the transformation matrix from equation 2.62. Using equations 2.62-2.64, we can efficiently move the cross-section reference axis to any desired point.

Cross-Section Mass Matrix

Deriving the cross-section's mass matrix is a much simpler endeavor. As laid out by Hodges [14], the cross-section mass matrix can simply be expressed as:

$$[M]_s = \begin{bmatrix} m & 0 & 0 & 0 & 0 & -m\vec{y} \\ 0 & m & 0 & 0 & 0 & m\vec{x} \\ 0 & 0 & m & m\vec{y} & -m\vec{x} & 0 \\ 0 & 0 & m\vec{y} & I_{xx} & -I_{xy} & 0 \\ 0 & 0 & -m\vec{x} & -I_{xy} & I_{yy} & 0 \\ -m\vec{y} & m\vec{x} & 0 & 0 & 0 & I_{xx}+I_{yy} \end{bmatrix} \tag{2.65}$$

where $\vec{x}$ and $\vec{y}$ are the components of the vector pointing from the origin of the reference axis to the mass center of the beam cross-section. It should also be noted that the moment of inertia terms are taken about the reference axis. The mass center can be calculated by determining the first moment of inertia and dividing it by the total mass per unit length of the cross-section. This results in the following:

$$x_{mc} = \frac{\sum_{i=1}^{N} x_i m_i}{\sum_{i=1}^{N} m_i} \qquad y_{mc} = \frac{\sum_{i=1}^{N} y_i m_i}{\sum_{i=1}^{N} m_i} \tag{2.66}$$

where N is the total number of elements in the discretized cross-section. Note that with this mass matrix, it is assumed that under any dynamic motion the kinetic energy due to warping dynamics is negligible. This concludes the derivation of the cross-sectional analysis used in the AeroComBAT module.

2.3 Cross-Sectional Analysis Stiffness Verification

The first set of cross-sections that AeroComBAT's cross-sectional analysis package was verified against were those Blasques [5] verified BECAS with against VABS. These cross-sections can be seen below in figure 2.8

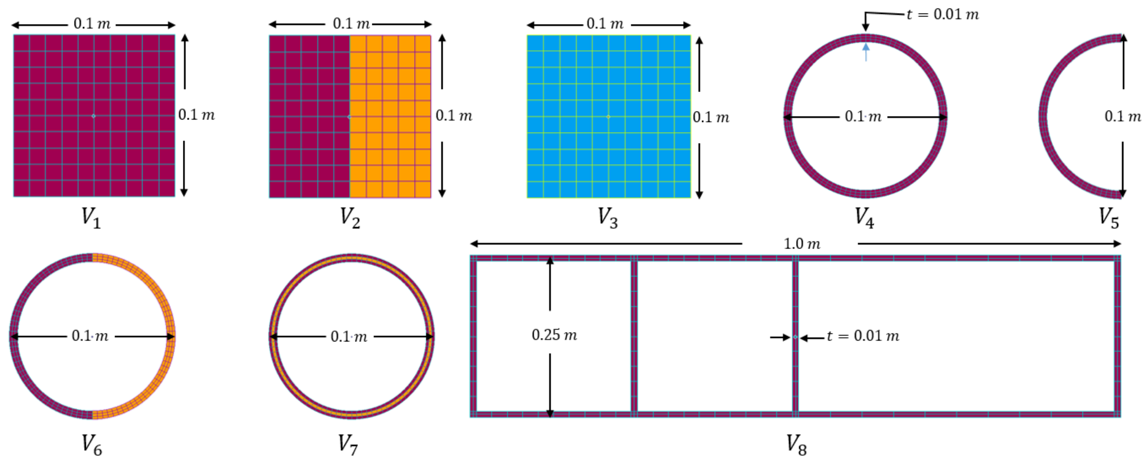


Figure 2.8: First set of cross-sections used to verify AeroComBAT

The colors in figure 2.8 correspond to different materials. The maroon material is isotropic Material 1, the orange material is made of a more compliant isotropic Material 2, and the blue material is an orthotropic Material 3, all of which can be seen below in table 2.1.

Table 2.1: Cross-section model information

Material	Material 1	Material 2	Material 3
E_{zz}	100	10	480
$E_{xx} = E_{yy}$	100	10	120
G_{yz}	41.667	4.1667	50
$G_{xz} = G_{xy}$	41.667	4.1667	60
ν_{yz}	0.2	0.2	0.26
$\nu_{xz} = \nu_{xy}$	0.2	0.2	0.19

* Table reproduced from [5]

The comparison of the results for validation cases V_1 through V_8 can be seen below in tables 2.2 - 2.9:

Table 2.2: Validation case 1 results

	AeroComBAT	BECAS	% Difference
K_{11}	3.4899E-1	3.4899E-1	0.
K_{22}	3.4899E-1	3.4899E-1	0.
K_{33}	1.0E+0	1.0E+0	0.
K_{44}	8.3384E-4	8.3384E-4	0.
K_{55}	8.3384E-4	8.3384E-4	0.
K_{66}	5.9084E-4	5.9084E-4	0.

Table 2.3: Validation case 2 results

	AeroComBAT	BECAS	% Difference
K_{11}	1.28E-1	1.28E-1	0.
K_{22}	1.92E-1	1.92E-1	0.
K_{33}	5.5E-1	5.5E-1	0.
K_{44}	4.59E-4	4.59E-4	0.
K_{55}	4.59E-4	4.59E-4	0.
K_{66}	2.77E-4	2.77E-4	0.
K_{26}	-3.93E-3	-3.93E-3	0.
K_{35}	1.13E-2	1.13E-2	0.

Table 2.4: Validation case 3 results

	AeroComBAT	BECAS	% Difference
K_{11}	5.039E-1	5.039E-1	0.
K_{22}	4.201E-1	4.201E-1	0.
K_{33}	4.8E+0	4.8E+0	0.
K_{44}	4.001E-3	4.001E-3	0.
K_{55}	4.001E-3	4.001E-3	0.
K_{66}	7.737E-4	7.737E-4	0.

Table 2.5: Validation case 4 results

	AeroComBAT	BECAS	% Difference
K_{11}	1.249E-1	1.249E-1	0.
K_{22}	1.249E-1	1.249E-1	0.
K_{33}	5.965E-1	5.965E-1	0.
K_{44}	2.697E-3	2.697E-3	0.
K_{55}	2.697E-3	2.697E-3	0.
K_{66}	2.248E-3	2.248E-3	0.

Table 2.6: Validation case 5 results

	AeroComBAT	BECAS	% Difference
K_{11}	4.964E-2	4.964E-2	0.
K_{22}	6.244E-2	6.244E-2	0.
K_{33}	2.982E-1	2.982E-1	0.
K_{44}	1.349E-3	1.349E-3	0.
K_{55}	1.349E-3	1.349E-3	0.
K_{66}	9.120E-4	9.120E-4	0.
K_{26}	-7.529E-3	-7.529E-3	0.
K_{35}	1.805E-2	1.805E-2	0.

Table 2.7: Validation case 6 results

	AeroComBAT	BECAS	% Difference
K_{11}	3.99E-2	3.99E-2	0.
K_{22}	6.87E-2	6.87E-2	0.
K_{33}	3.28E-1	3.28E-1	0.
K_{44}	1.48E-3	1.48E-3	0.
K_{55}	1.48E-3	1.48E-3	0.
K_{66}	1.08E-3	1.08E-3	0.
K_{26}	6.78E-3	6.78E-3	0.
K_{35}	1.62E-2	1.62E-2	0.

Table 2.8: Validation case 7 results

	AeroComBAT	BECAS	% Difference
K_{11}	8.3114E-2	8.3114E-2	0.
K_{22}	8.3114E-2	8.3114E-2	0.
K_{33}	3.9784E-1	3.9784E-1	0.
K_{44}	1.8012E-3	1.8012E-3	0.
K_{55}	1.8012E-3	1.8012E-3	0.
K_{66}	1.5010E-3	1.5010E-3	0.

Table 2.9: Validation case 8 results

	AeroComBAT	BECAS	% Difference
K_{11}	7.61E-1	7.61E-1	0.
K_{22}	2.93E-1	2.93E-1	0.
K_{33}	2.92E+0	2.92E+0	0.
K_{44}	3.29E-2	3.29E-2	0.
K_{55}	2.93E-1	2.94E-1	0.34
K_{66}	3.95E-2	3.95E-2	0.
K_{26}	-8.19E-3	-8.26E-3	0.847
K_{35}	5.69E-2	5.75E-2	1.04

From these results, it is clear that the AeroComBAT is capable of handling geometrically complex, inhomogeneous cross-sections with isotropic and orthotropic materials. This accuracy is not surprising since AeroComBAT and BECAS both share the same formulation. In addition, the same meshes and element types were also used.

In order to verify AeroComBAT with cross-sections less academic in nature, it was also verified against four cross-sections analyzed by VABS and NABSA [12]. The material properties and layup schedules can be seen below in Tables 2.10 and 2.11.

Table 2.10: Box Beam Material Properties

Material	E_{11}	$E_{22} = E_{33}$	$G_{12} = G_{13}$	G_{23}	$\nu_{12} = \nu_{13}$	ν_{23}
$AS4/3501-6$	142 GPa	9.8 GPa	6 GPa	4.9 GPa	0.3	0.34
$AS4/3501-6^*$	142 GPa	9.8 GPa	6 GPa	4.9 GPa	0.3	0.42

Table 2.11: Box beam verification layup schedules

Case	Material	Laminate 1	Laminate 2	Laminate 3	Laminate 4
B1	$AS4/3501-6$	$[15]_6$	$[15]_6$	$[15]_6$	$[15]_6$
Layup 1	$AS4/3501-6^*$	$[0]_6$	$[0]_6$	$[0]_6$	$[0]_6$
Layup 2	$AS4/3501-6^*$	$[30/0]_3$	$[30/0]_3$	$[30/0]_3$	$[30/0]_3$
Layup 3	$AS4/3501-6^*$	$[15]_6$	$[15/-15]_3$	$[-15]_6$	$[-15/15]_3$

The ply thicknesses for all cases was $t = 0.172$mm. The geometry of the cross-section can be seen below in figure 2.9, and the mesh used by AeroComBAT can be seen in figure 2.10.

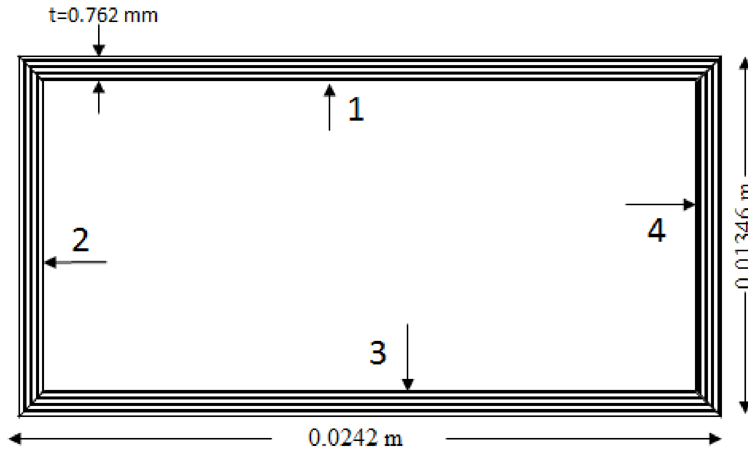


Figure 2.9: Diagram of box beam geometry

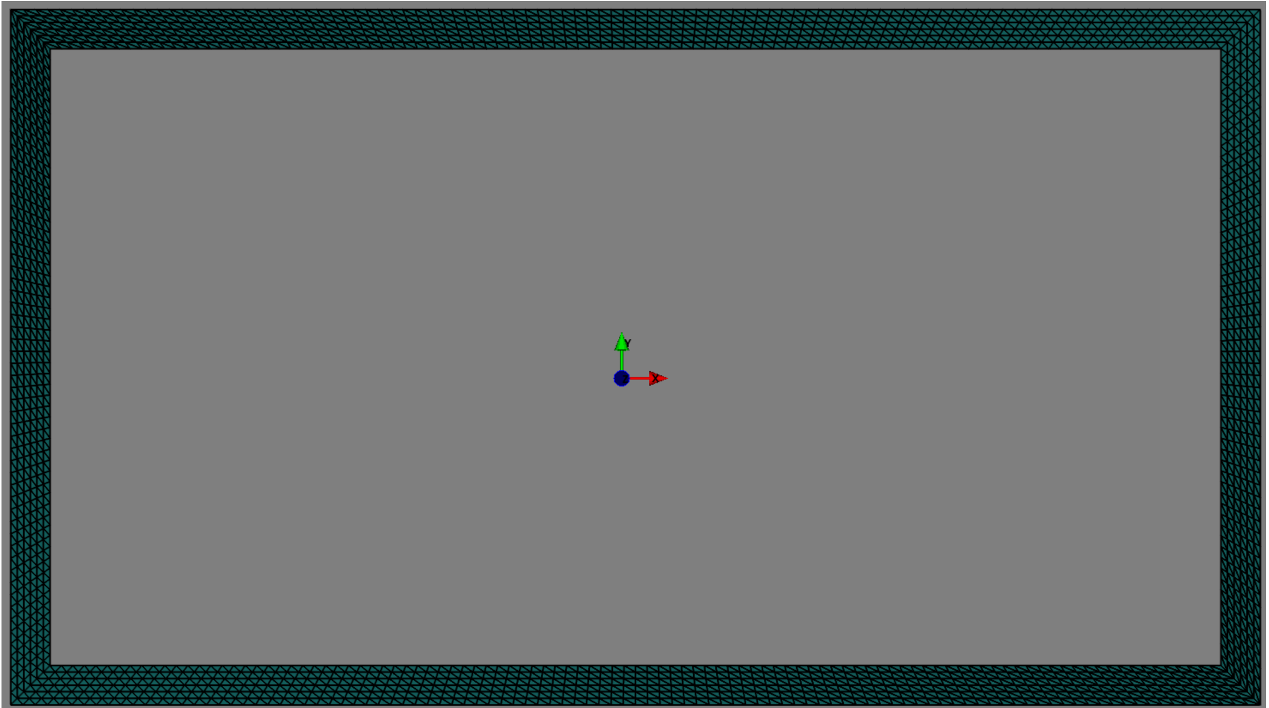


Figure 2.10: The cross-sectional mesh used by AeroComBAT

This geometry of the cross-section comes directly from the geometry used by Hodges. Note that based on figure 2.10 it might appear as though the triangular elements are used in the cross-sectional analysis. This is a deception of the visualizer MayaVi which plots surfaces only as triangulated surfaces. In reality, linear quadrilateral elements are used to discretize the cross-section. Below in figures 2.11- 2.14 are the cross-sectional stiffness term results for the four different cases. It should be noted that the units of all of the entries in these tables are MPa with the exception of the percentages.

B1	K11	K14	K22	K25	K33	K36	K44	K55	K66
AeroComBAT	6.22E+02	-3.59E+02	2.71E+02	-3.89E+02	9.91E+03	7.41E+02	4.57E+02	1.19E+03	1.16E+02
NABSA	6.22E+02	-3.59E+02	2.71E+02	-3.89E+02	9.91E+03	7.41E+02	4.57E+02	1.19E+03	1.16E+02
VABS	3.47E+02	-2.03E+02	1.45E+02	-2.06E+02	9.96E+03	7.52E+02	3.77E+02	9.32E+02	1.19E+02
NABSA % Error	0.02	0.01	0.06	0.00	0.00	0.04	0.01	0.02	0.00
VABS % Error	79.04	76.45	87.44	88.92	0.48	1.42	21.25	27.64	2.38

Figure 2.11: Case B1 Stiffness Results

Layup 1	K11	K22	K33	K44	K55	K66
AeroComBAT	3.06E+02	1.29E+02	1.21E+04	5.82E+02	1.47E+03	5.50E+01
NABSA	3.06E+02	1.31E+02	1.22E+04	5.99E+02	1.48E+03	5.63E+01
VABS	3.07E+02	1.32E+02	1.22E+04	5.99E+02	1.48E+03	5.65E+01
NABSA % Error	0.09	1.78	0.63	2.88	0.97	2.16
VABS % Error	0.14	2.29	0.63	2.88	0.97	2.52

Figure 2.12: Case Layup 1 Stiffness Results

Layup 2	K11	K14	K22	K25	K33	K36	K44	K55	K66
AeroComBAT	6.77E+02	-1.79E+02	2.87E+02	-1.89E+02	8.55E+03	3.54E+02	4.11E+02	1.04E+03	1.19E+02
NABSA	6.76E+02	-1.82E+02	2.92E+02	-1.92E+02	8.62E+03	3.59E+02	4.23E+02	1.05E+04	1.22E+02
VABS	6.01E+02	-1.61E+02	2.57E+02	-1.68E+02	8.62E+03	3.59E+02	4.18E+02	1.03E+04	1.22E+02
NABSA % Error	0.14	1.50	1.86	1.21	0.81	1.41	2.89	90.09	2.25
VABS % Error	12.79	11.13	11.56	12.55	0.81	1.41	1.61	89.96	2.25

Figure 2.13: Case Layup 2 Stiffness Results

Layup 3	K11	K12	K13	K22	K23	K33	K44	K45	K46	K55	K56	K66
AeroComBAT	6.10E+02	-5.25E-01	-1.27E+03	2.67E+02	8.46E-01	9.44E+03	4.08E+02	2.54E+00	1.21E+02	9.75E+02	2.45E+00	1.17E+02
NABSA	6.09E+02	-5.66E-01	-1.27E+03	2.72E+02	9.93E-01	9.45E+03	4.19E+02	2.60E+00	1.24E+02	9.86E+02	2.47E+00	1.19E+02
VABS	6.09E+02	-5.81E+00	-1.27E+03	5.34E+01	1.21E+01	9.45E+03	4.19E+02	2.56E+00	1.24E+02	9.86E+02	2.50E+00	1.20E+02
NABSA % Error	0.11	7.31	0.12	1.94	14.82	0.11	2.73	2.15	2.13	1.08	0.88	2.04
VABS % Error	0.22	90.96	0.12	399.77	93.03	0.11	2.73	0.83	2.13	1.08	1.97	2.61

Figure 2.14: Case Layup 3 Stiffness Results

There are several very clear trends that appear from these results. The first not so surprising trend is that the stiffnesses produced by NABSA are very close to those produced by AeroComBAT. The reason this is not so surprising is that they use the same formulation.

For case B1, it is clear that AeroComBAT was able to achieve nearly the exact same results as NABSA which again is what was expected since they share the same original formulation. Transitioning to case Layup 1, the results are somewhat puzzling. Generally the error everywhere is low, however it is clear that there is a significant discrepancy, especially in the axial (K_{33}) and bending (K_{44} and K_{55}) stiffness terms. Referring back to Table 2.11, it can be seen that the cross-section for Layup 1 is relatively simple as all of the fibers are running in the 0 degree direction. Since the axial and bending stiffnesses (K_{33},K_{44} and, K_{55}) are simple integration terms and the fibers all run in the 0 degree direction, the actual stiffnesses can be analytically calculated. For example:

$$K_{33} = E * A \rightarrow A = \frac{K_{33}}{E} \tag{2.67}$$

Assuming that the same E was used to generate the stiffnesses, the cross-sectional area can be factored out of all of the stiffness terms. The analytical area of the cross-section is $4.703mm^2$. Using equation 2.67, the areas of the cross-sections are $4.703mm^2$, $4.763mm^2$, and $4.763mm^2$ for AeroComBAT, NABSA, and VABS, respectively. This quick hand calculation indicates that the reported axial stiffnesses (if not the rest of the stiffnesses reported as well) for NABSA and VABS are wrong for the geometry and material properties included in the original paper [12]. Since the exact source of error in the original data is unclear, it is difficult to predict how that error propagates and how it has effected the different stiffnesses. Stranger still is that these errors do not appear in case B1. This might be explained by the fact that the NABSA and VABS data from case B1 originated from a different paper [11], and so it is possible the error was not committed for that earlier set of results.

Similar trends can be seen for case Layup 2. In general, AeroComBAT's results are close to those produced by NABSA and VABS except for the K_{55} bending stiffness term, although this is also believed to be a typo in the original data published by Hodges [12]. The reasoning for this conclusion is as follows.

Consider the case Layup 2. The AeroComBAT K_{55} bending stiffness appears to be roughly an order of magnitude less than those predicted by NABSA and VABS. In addition, observe that the K_{55} bending stiffness for case Layup 1 is roughly 1.48E3 MPa. In Layup 1, all of the laminate fibers are running in the axial direction, so the bending stiffness would be at its highest in this case. For Layup 2 when half of the plies are rotated 30 degrees, one would expect the bending stiffness to go down. Hodges data implies that NABSA and VABS predict the bending stiffnesses to increase to a value of 1.05E4 MPa. In contrast, AeroComBAT predicts the bending stiffness to go down to 1.02E3 MPa. In addition, the K_{44} stiffnesses appear to match this trend as its magnitude decreases from case Layup 1 to Layup 2 for all three programs. As such, it is likely that the error in the AeroComBAT K_{55} term is merely due to a typo in the original data. The error in the stiffness terms from case Layup 3 is similar to that produced by cases Layup 1 and Layup 2.

Over all, these four case studies show AeroComBAT to predict the cross-sectional stiffness terms with roughly a 2% difference with respect to NABSA

and VABS. Furthermore a significant portion of this error appears to be the result of incorrect data [12]. It is likely that most sources of the relatively small error are due to slight differences in the geometry of Hodges' models.

2.4 Cross-Sectional Analysis Stress Recovery Verification

Having the ability to accurately predict the stiffness of any discretized cross-section does a great deal to make beam analysis viable for composite structures. Without an accurate method to capture the stress within the beam cross-section, however, beam analysis becomes hard to justify. Without accurate stress recovery, predicting the stress failure of the structure is impossible. In order to verify the accuracy of the AeroComBAT 3D stress recovery, the Layup 3 composite cross-section from Section 2.3 was considered. A model using this cross-section was generated in NASTRAN using solid CHEXA elements as seen in figure 2.15.

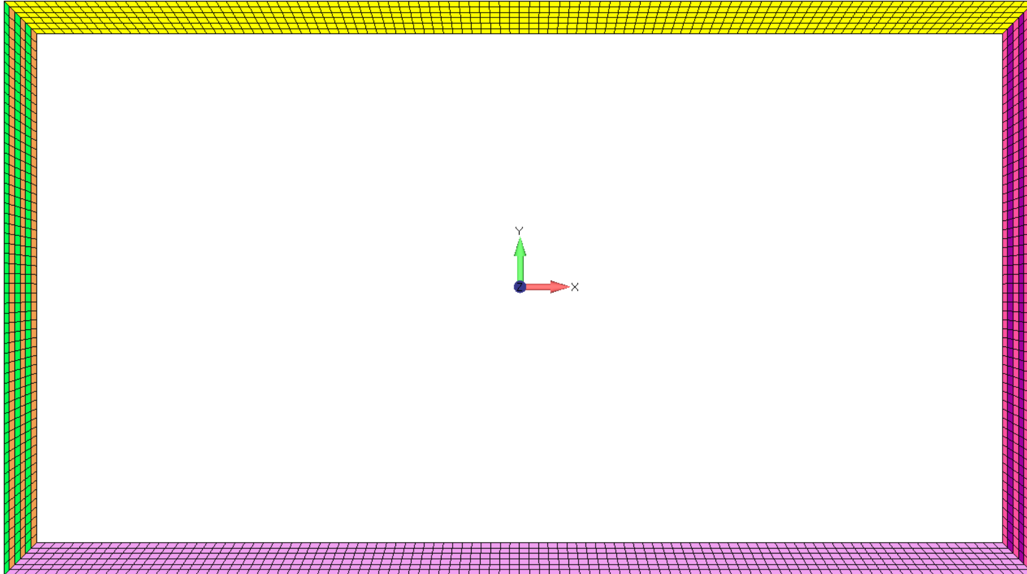


Figure 2.15: Cross-section mesh used by AeroComBAT and NASTRAN for 3D stress recovery

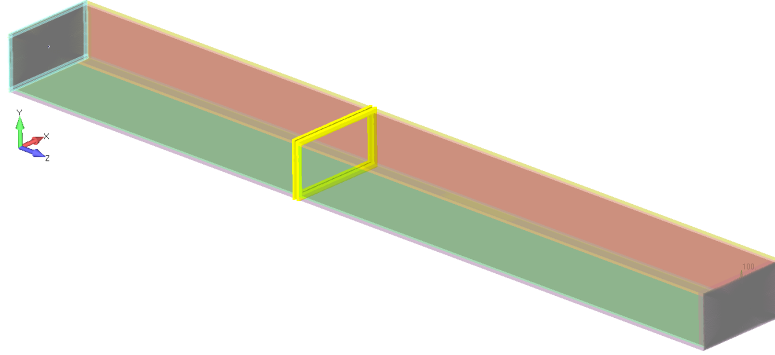


Figure 2.16: Highlighted section shows where stress was sampled in NASTRAN model

The identical cross-sectional mesh was used in the AeroComBAT model in an attempt to make fair comparisons of the stress states. In order to avoid warp restraint effects which AeroComBAT cannot currently take into account, the NASTRAN beam solid model was given a length of 0.204 meters and the stresses were sampled from the highlighted row of elements in figure 2.16. In total, the NASTRAN model contained 558,729 nodes and 477,003 elements. The nodes at the root of the beam were constrained using an RBE3 element, and the opposite end of the beam was loaded with a force of $[F_x, F_y, F_z] = [444N, 444N, 44444N]$ using an RBE3 element. The reason that the RBE3 element was used on both ends was to mitigate warp-restraint effects. In order to explain how this works, the NASTRAN RBE3 element must first be explained.

The RBE3 element has many master nodes and one slave node, and the displacement and rotation of the slave node is determined based on the average displacements of the master nodes. By connecting a very stiff spring to the slave node of the RBE3 at the constrained end of the beam, theoretically the average displacement and rotation of the beam should be zero while simultaneously allowing the master nodes to displace. In the NASTRAN model, a slice of elements at z = 0.1018 meters was surveyed for their full stress state $(\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{xz}, \sigma_{yz})$. In figures 2.23- 2.18, the stress states predicted by AeroComBAT and the error with respect to NASTRAN can be seen. For a given stress state, the error in an element was calculated with:

$$\sigma_{error} = \frac{\sigma_{NASTRAN} - \sigma_{AeroComBAT}}{max(\sigma_{NASTRAN})} 100 \tag{2.68}$$

The three stresses displayed are the dominant beam stresses, $(\sigma_{zz}, \sigma_{xz}, \sigma_{yz})$.

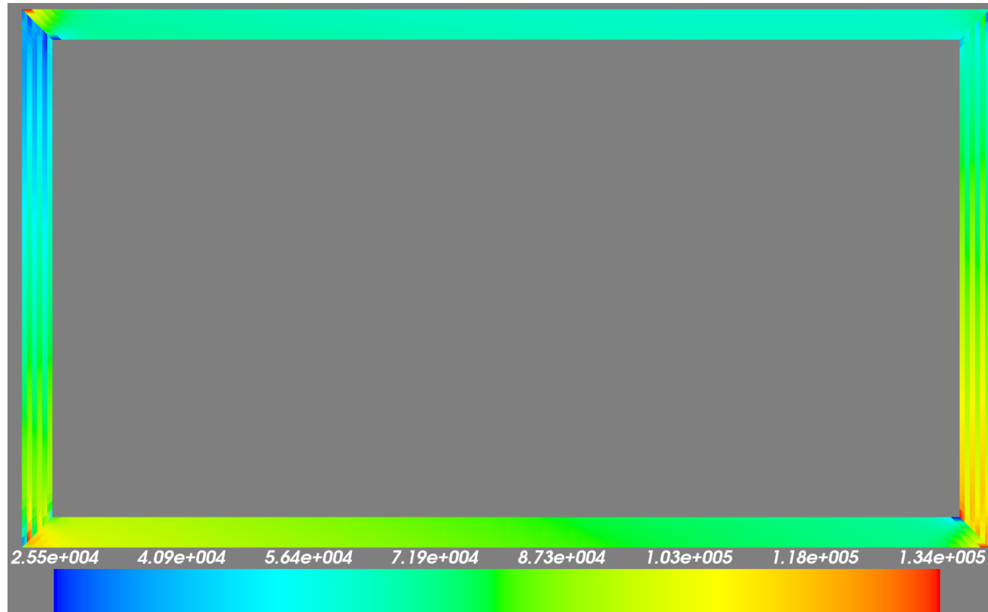


Figure 2.17: Layup 3 σ_{zz} Stress

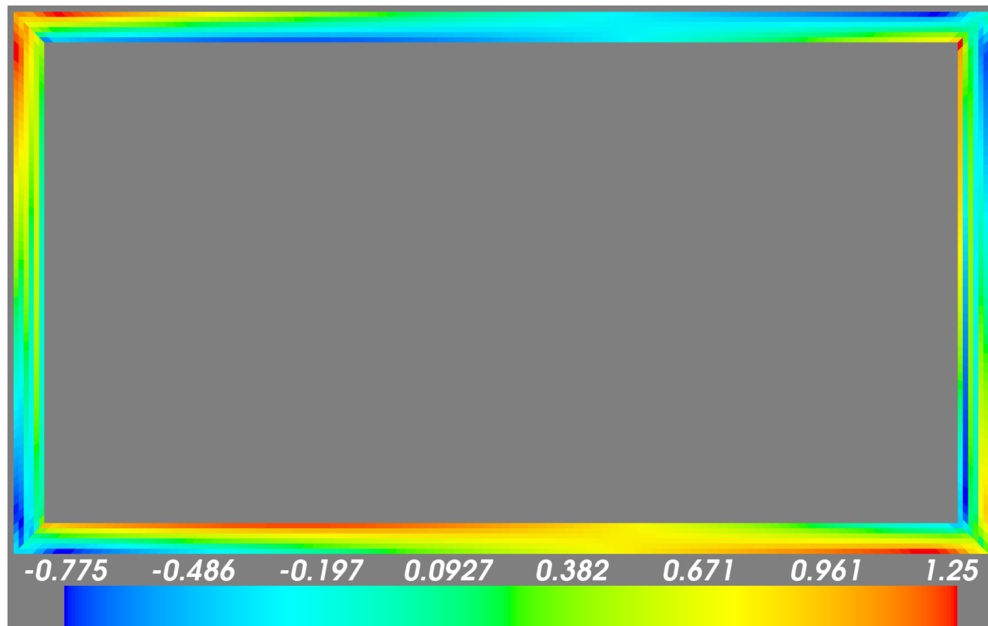


Figure 2.18: Layup 3 σ_{zz} AeroComBAT % Error

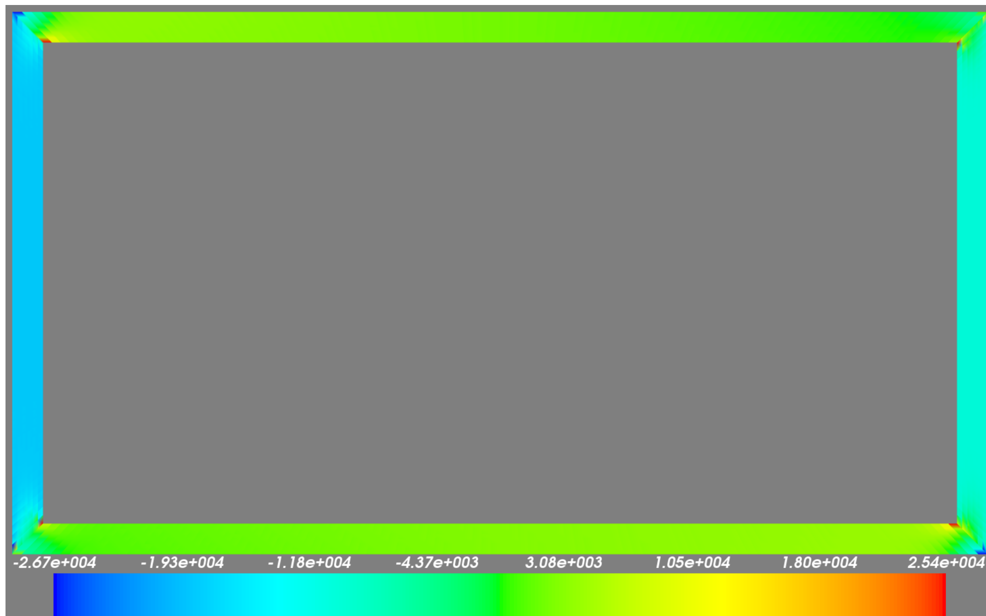


Figure 2.19: Layup 3 σ_{xz} Stress

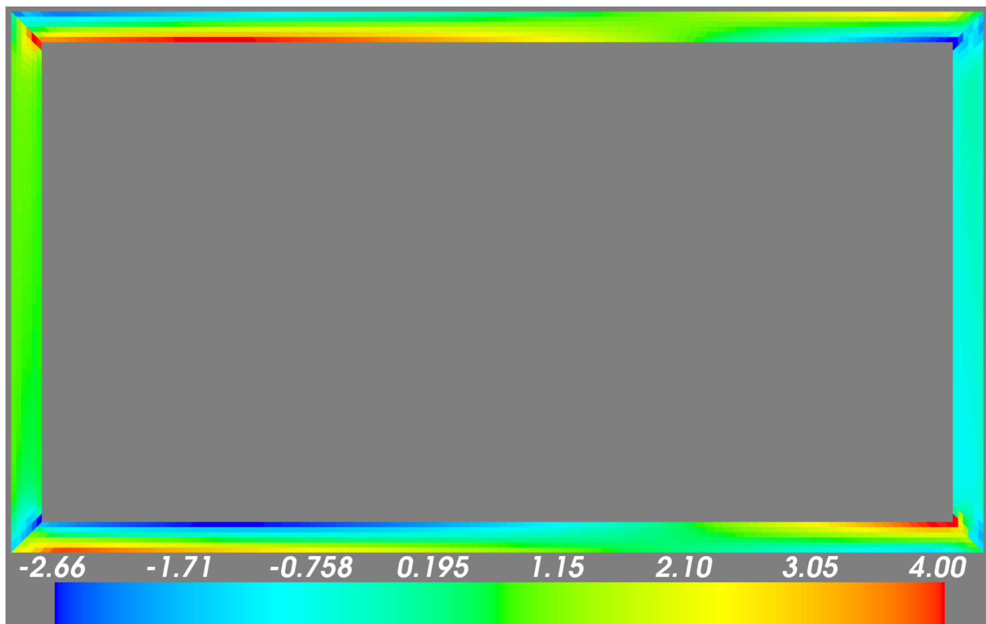


Figure 2.20: Layup 3 σ_{xz} AeroComBAT % Error

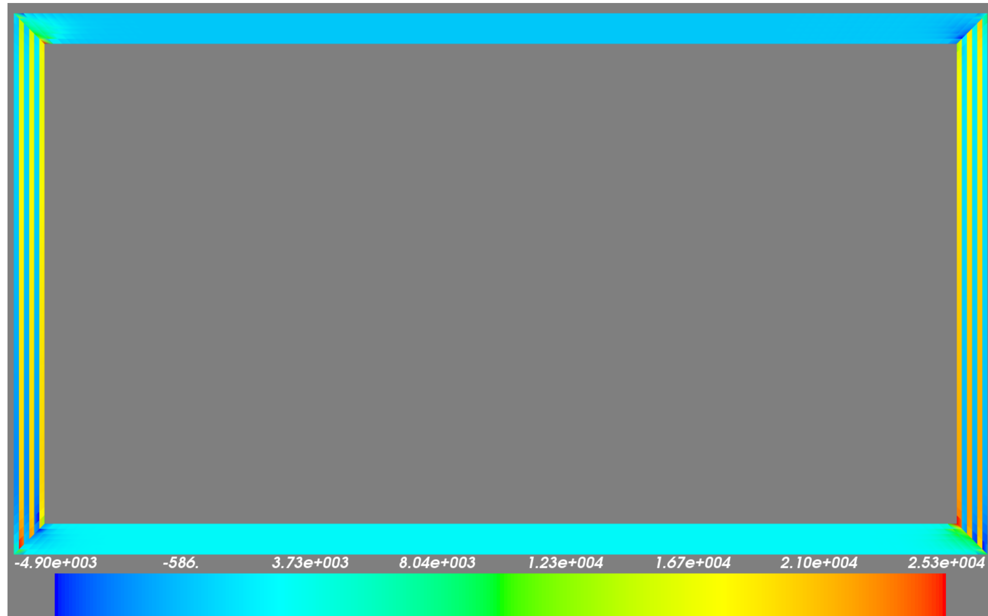


Figure 2.21: Layup 3 σ_{yz} Stress

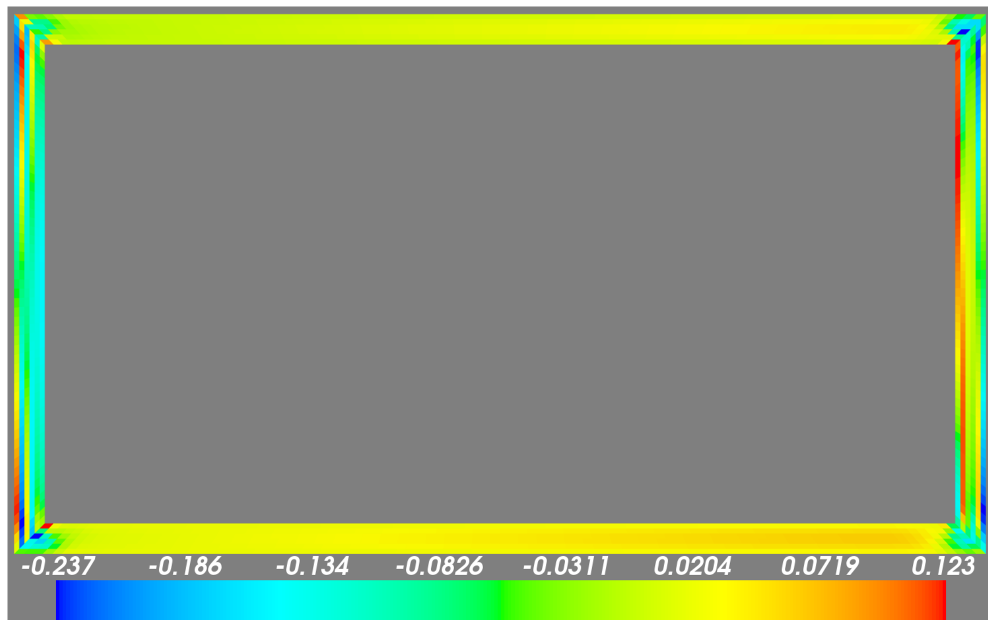


Figure 2.22: Layup 3 σ_{yz} AeroComBAT % Error

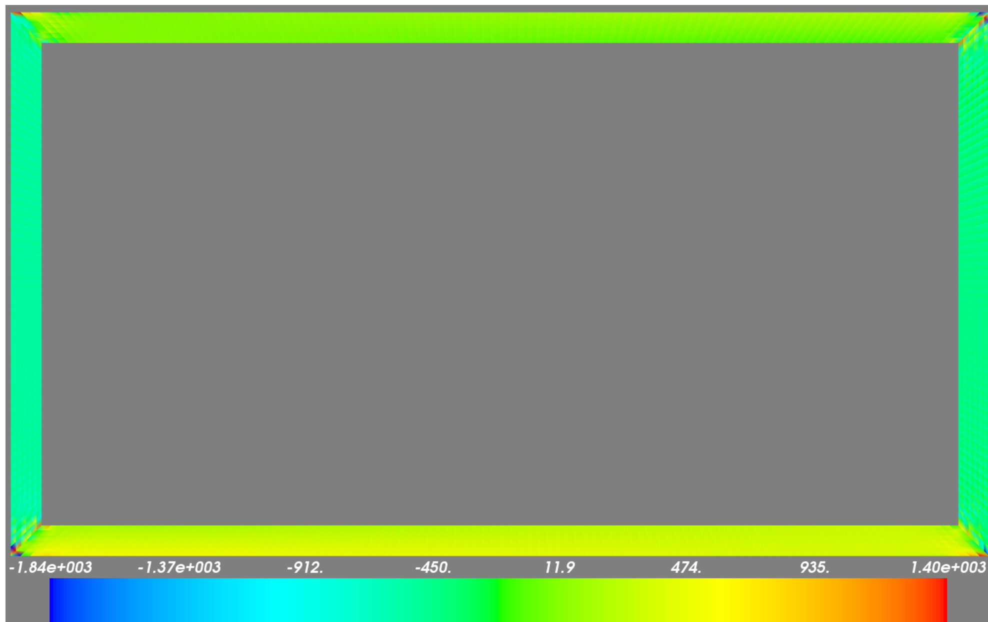


Figure 2.23: Layup 3 σ_{xx} Stress

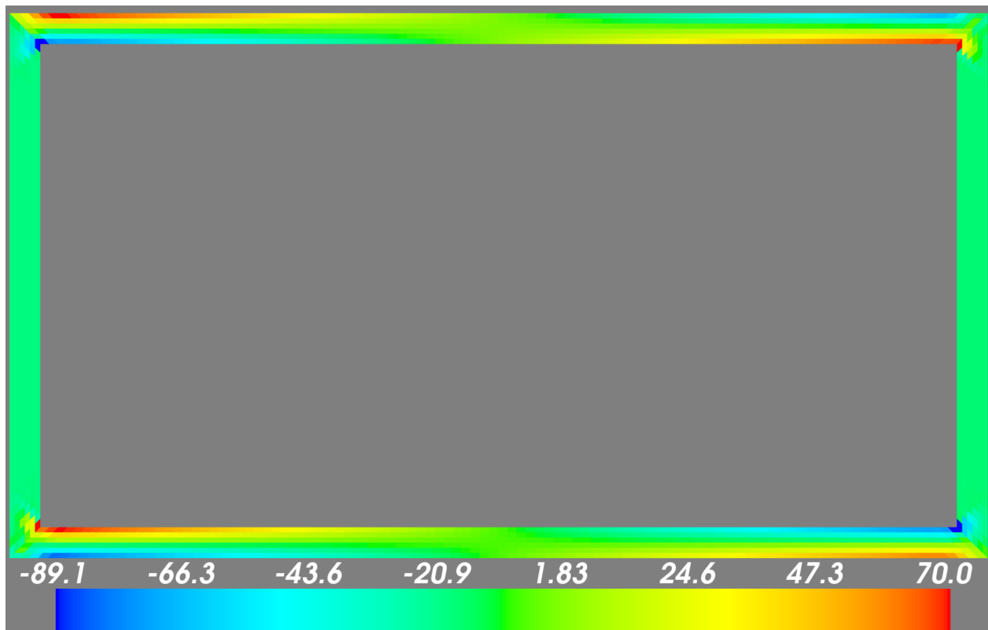


Figure 2.24: Layup 3 σ_{xx} AeroComBAT % Error

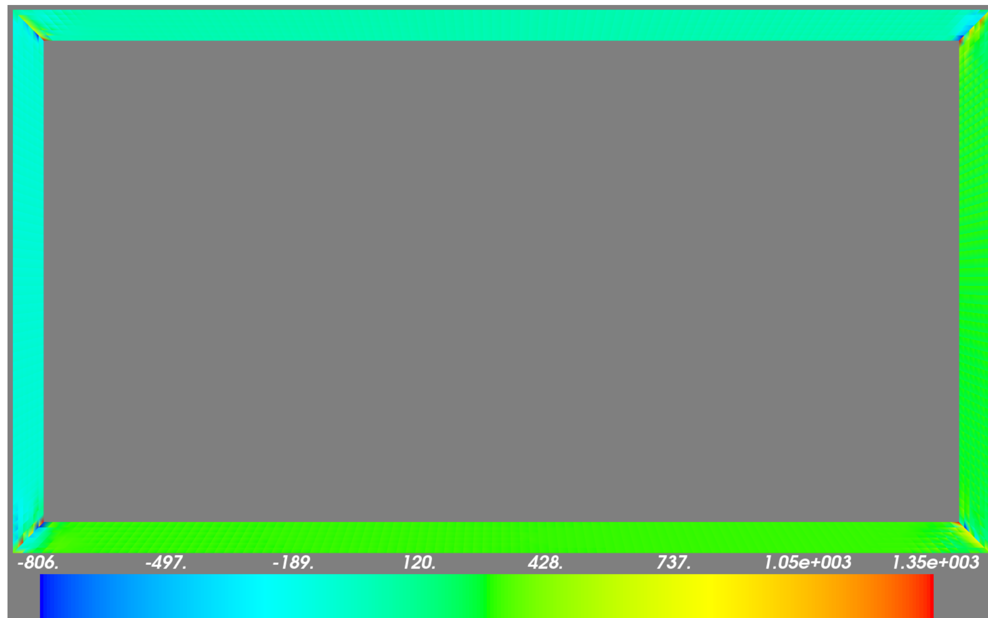


Figure 2.25: Layup 3 σ_{xy} Stress

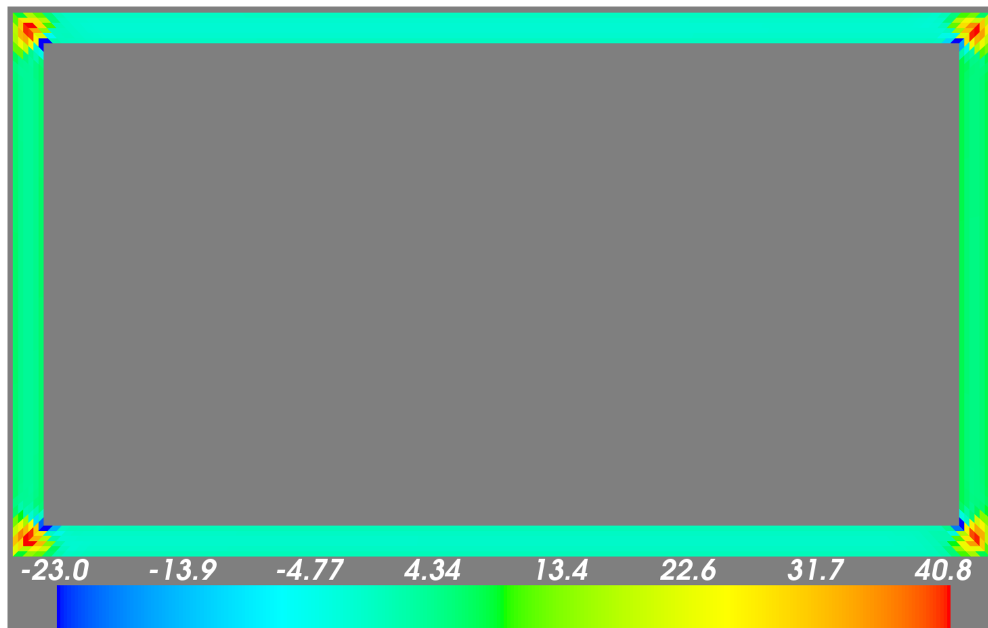


Figure 2.26: Layup 3 σ_{xy} AeroComBAT % Error

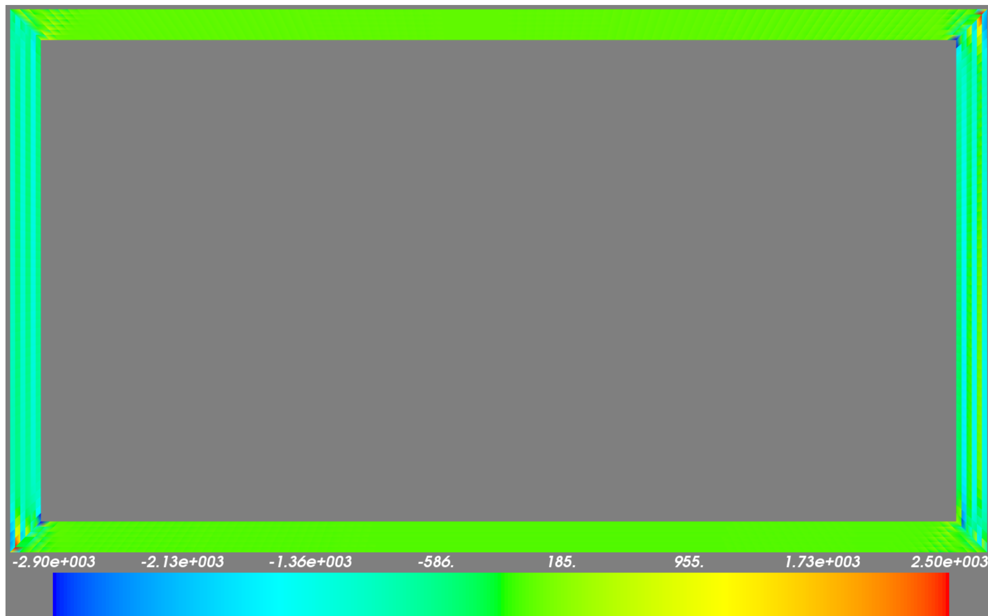


Figure 2.27: Layup 3 σ_{yy} Stress

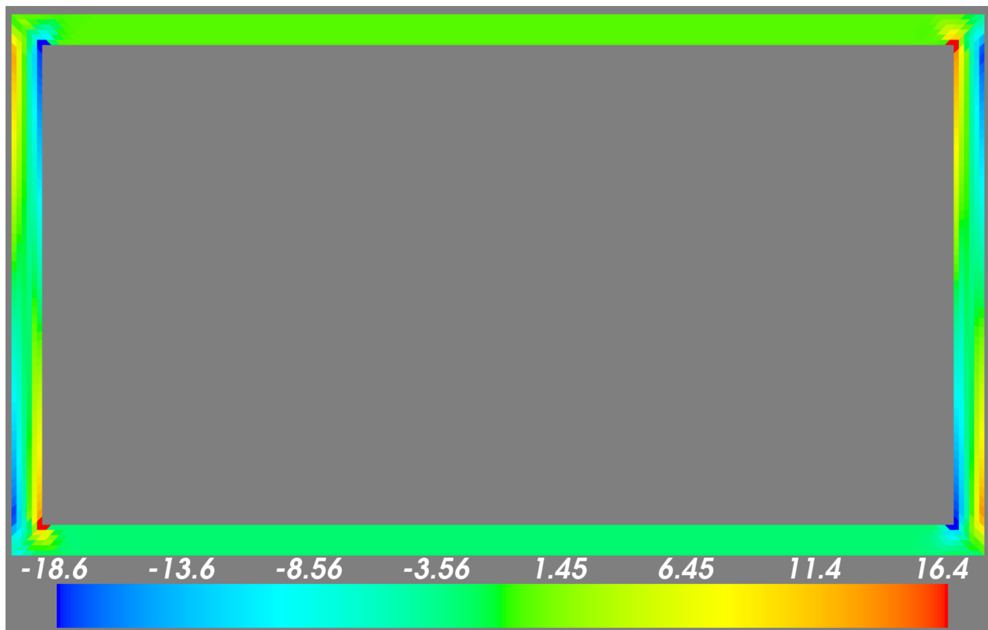


Figure 2.28: Layup 3 σ_{yy} AeroComBAT % Error

From Figures 2.23- 2.18, it is clear that AeroComBAT was able to capture

a significant portion of the intricate stress states due to the composite nature of the box beam. In particular, all features of the σ_{zz},σ_{xz}, and σ_{yz} stress states were captured. From Figures 2.24, 2.26, and 2.28 it is clear however that certain stress features are not accounted for. Looking at the features of the σ_{xx}, σ_{xy}, and σ_{yy} together, it appears as though there is in-plane bending occurring in the cross-section.

In order to narrow down under what conditions this in-plane bending occurs, a more simple solid beam model was generated. The new beam is geometrically identical to the previous beam, however uses an isotropic aluminum 6061-T6 using $E = 62.1$MPa and $\nu = 0.33$. After further investigation of the aluminum beam model, it became clear that this in-plane bending is due to the combination of a generalized beam torsional strain and warp restraint effects. Below is the deformed cross-section of sampled elements in the aluminum beam shown in figure 2.16 (contoured is the σ_{xx} stress state):

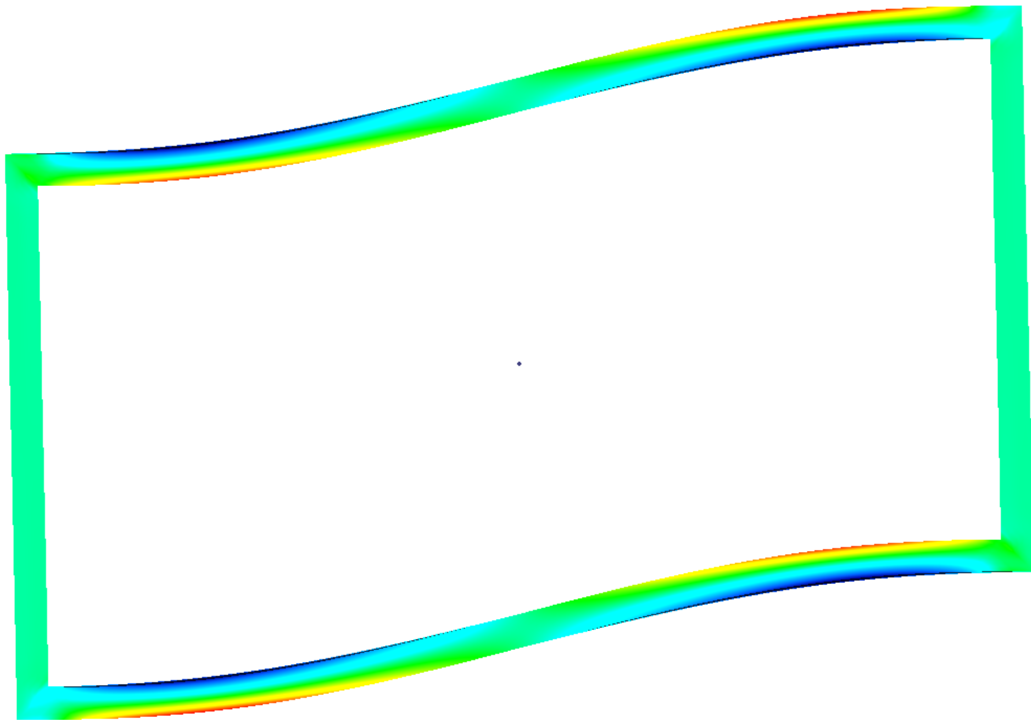


Figure 2.29: The x normal stress in the NASTRAN aluminum box beam under torsion

All of the cross-section's warping is in-plane cross-section warping. In con-

trast, the AeroComBAT warping displacement solution for the same problem can be seen in Figure 2.30

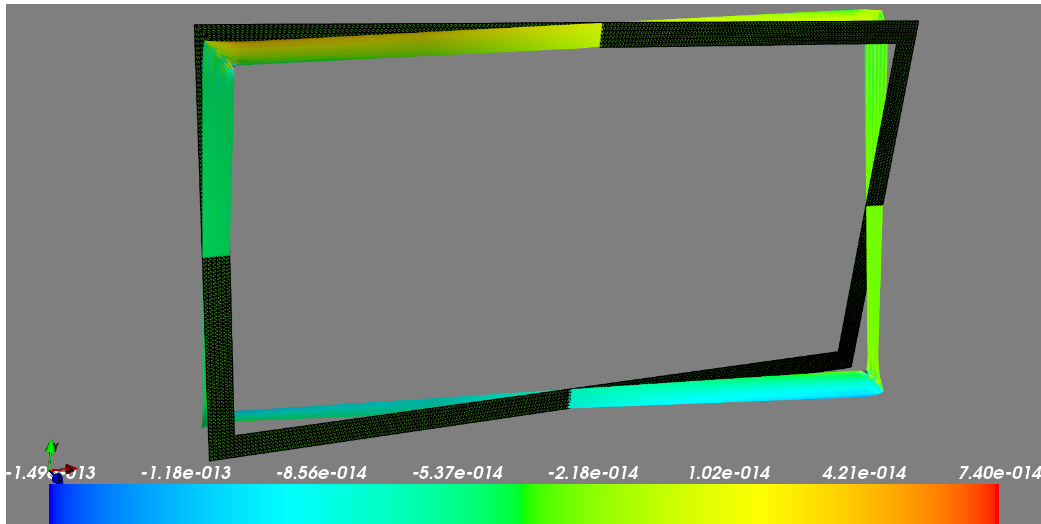


Figure 2.30: The x normal stress in the AeroComBAT aluminum box beam under torsion

For the simple aluminum beam, when a torque is applied at the tip, it should twist along its length and also exhibit out of plane warping, visible in figure 2.30. Displayed below in figures 2.31 and 2.32 are the warping ends of the NASTRAN aluminum beam.

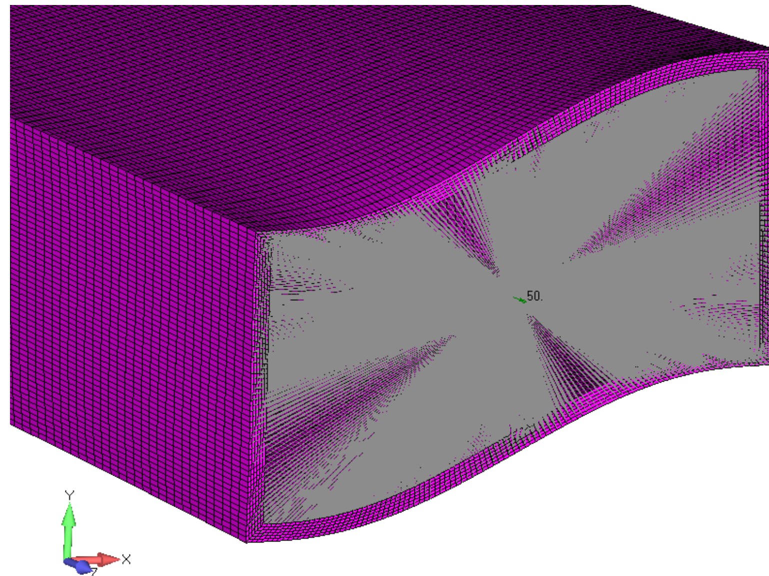


Figure 2.31: The warping at the tip of the beam where the torque is applied

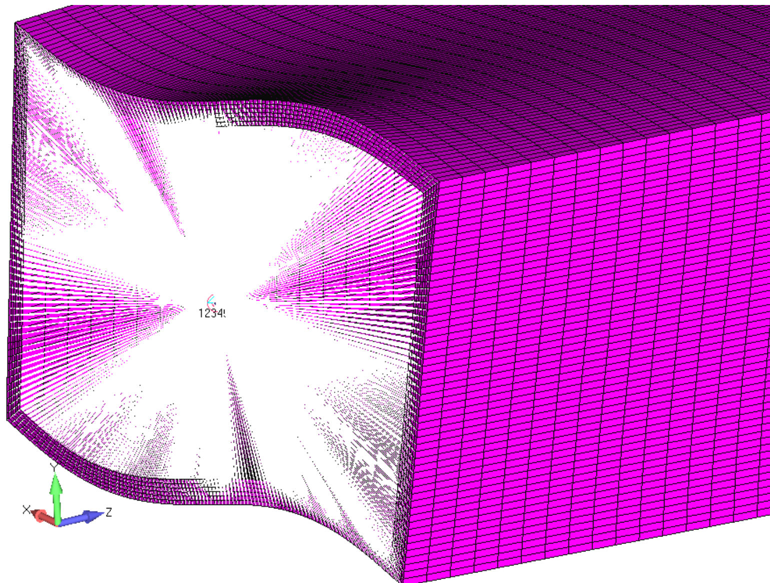


Figure 2.32: The warping at the end of the beam where the beam is constrained

From figures 2.31 and 2.32, it is clear that the cross-section warping NASTRAN produces is incorrect when RBE3 elements are used. It is likely that the in-plane bending is the result of the beam attempting to reconcile a gen-

eralized torsional strain and improperly enforced displacement fields at its ends.

Knowing when this in-plane bending occurs, the behavior of the original composite box beam can be explained. From Section 2.3, we know that this cross-section exhibits shear-torsion coupling. As such, when the tip loads are applied to the beam, a generalized torsional strain occurs. The improper behavior then occurs because the RBE3 elements are incapable of accurately resolving the torsional free-warping displacement fields at the ends.

2.5 Conclusion

The governing thesis of this work is that if this cross-sectional analysis can prove sufficiently accurate, it could help expand the design space in the early design process when iterating with shell models would be too costly. Chapter 2 has shown that not only is the cross-sectional analysis capable of capturing the cross-sectional stiffness matrix of a fully composite beam modeled with solid elements, but can also accurately recover the 3D stress-state within that cross-section due to any arbitrary loading. It is also clear however that in order to be fully confident in the results of the stress recovery it would be prudent to incorporate warp-restraint into the model in the future. It should also be noted that AeroComBAT was able to calculate the the cross-sectional stiffnesses and recover the 3D stresses in 28 seconds, whereas it took NASTRAN 21 minutes to do the same. This drastic difference in efficiency shows how effective a tool like AeroComBAT could be during the early design process, unlocking designs with complex layups and geometry that could never have otherwise been considered. It is important to keep in mind that in some senses these analysis are not comparable as you would most likely never create a solid model for this composite beam. Despite this fact, the time difference and the results that the cross-sectional analysis can produce illustrates the potential of this reduced order modeling method.

Chapter 3

Global Beam Behavior

3.1 Introduction

When evaluating beam models for structural analysis, two problems must be solved. The first is determining how the global beam behavior will define the local cross-sectional behavior. This is the cross-sectional analysis problem which Chapter 2 is dedicated to solving. The second problem is solving for the global beam behavior. Many different approaches have been taken in order to solve for the dominant beam physics of a structural model. Some of the major beam theories include the classical Euler-Bernoulli beam theory, the Timoshenko beam theory, the Geometrically Exact Beam Theory proposed by Reissner [21], as well as Hodges Fully Intrinsic Generalized Timoshenko Theory [14].

At this point it is important to restate the focus of the present work in order to convey why certain structural models were selected over others. One of the primary goals is to show that beam structural models still have a place in the design process if used in conjunction with an adequate cross-sectional analysis tool. As such, it was deemed that the structural model did not need to take material nonlinearity or geometric nonlinearity into account. It is possible that in future work, geometric linearities will be incorporated into the AeroComBAT API as this could allow for a more accurate representation of the physics present in significantly long and slender structures. The most simple beam formulation which was alluded to in Chapter 2 is the Euler-Bernoulli Beam Theory which many are familiar with. Recall that AeroComBAT's cross-sectional analysis package is capable

of recovering a fully populated 6x6 cross-sectional stiffness matrix including the shear stiffness terms of the beam. Since the shear effects are neglected in the Euler-Bernoulli beam theory, not only would the primary shear effects be neglected, but so would any possible shear coupling effects as well. Since these effects can be extremely common when incorporating composites into a beam design, the Euler-Bernoulli beam theory was deemed insufficient. Ultimately, the Timoshenko beam theory was selected as the structural model for the transverse loading of the beam. The governing differential equations for a beam running along the vector $\vec{v} = (0, 0, 1)$ loaded in the transverse x-direction are [20]:

$$\frac{d}{dz}\left(GA\kappa_x\left(-\psi + \frac{du}{dz}\right)\right) = -f_x \tag{3.1}$$

$$-\frac{d}{dz}\left(EI_{yy}\frac{d\psi}{dz}\right) + GA\kappa_x\left(\psi + \frac{du}{dz}\right) = m_y \tag{3.2}$$

where ψ is the total rotation the beam experiences (due to bending and shear), and κ_x is the beam's curvature about the x-axis. Note that this model assumes the cross-section is simple since the transverse z deflection and y rotation are not coupled with any other degree of freedom. For the axial behavior (both extension and torsion), the classical second order models were used. These equations can be seen below [20]:

$$\frac{d}{dz}\left(EA\frac{dw}{dz}\right) = p_z \tag{3.3}$$

$$\frac{d}{dz}\left(GJ\frac{d\phi}{dz}\right) = m_z \tag{3.4}$$

These models in themselves are not sufficient, as besides the transverse deflection and rotation, there is no way for the other degrees of freedom to couple. That said equations 3.1- 3.4 will provide an adequate foundation for the generalized beam finite elements used in the present work.

3.2 Linear Static Finite Element Formulation

In order derive the beam finite element formulation used to solve the behavior of the beam at the global level, the principle of virtual work is applied on a

beam using the models present in equations 3.1-3.4. The deformation of the beam is:

$$q(z)_{global} = \begin{Bmatrix} u(z) \\ v(z) \\ w(z) \\ \beta(z) \\ \psi(z) \\ \phi(z) \end{Bmatrix} \tag{3.5}$$

where u and v are the transverse beam displacements, w is the axial displacement, β and ψ are the transverse beam rotations, and ϕ is the torsional rotation. The deformation over the beam is approximated using a series of elements. Each of these elements use a series of piecewise shape functions to approximate the deformation over the domain of the element. These shape functions can be written in matrix form $[N]$:

$$[N] = \begin{bmatrix} u_{\{1\times l\}} & 0 & 0 & 0 & 0 & 0 \\ 0 & v_{\{1\times m\}} & 0 & 0 & 0 & 0 \\ 0 & 0 & w_{\{1\times n\}} & 0 & 0 & 0 \\ 0 & 0 & 0 & \beta_{\{1\times o\}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \psi_{\{1\times p\}} & 0 \\ 0 & 0 & 0 & 0 & 0 & \phi_{\{1\times r\}} \end{bmatrix} \tag{3.6}$$

where u, v, and w are row vectors of the mode shapes used for the displacement in the x, y, and z directions respectively. Similarly and β, ψ, and ϕ are row vectors of the shape functions used for the rotations about the x, y, and z axis respectively. A column vector of the corresponding generalized beam displacements and rotations can be written as:

$$\{q\}_e = \begin{Bmatrix} U_{\{l\times 1\}} \\ V_{\{m\times 1\}} \\ W_{\{n\times 1\}} \\ B_{\{o\times 1\}} \\ \Psi_{\{p\times 1\}} \\ \Phi_{\{r\times 1\}} \end{Bmatrix} \tag{3.7}$$

where U, V, W, B, Ψ, and Φ are column vectors of the nodal displacements and rotations along the beam element. Using equations 3.6 and 3.7, the beam element displacements and rotations can be expressed anywhere in the beam using the typical finite element formulation presented in equation 3.8:

$$q(z)_e = [N]\{q\}_e \tag{3.8}$$

In order to apply the principle of virtual work, an expression for the element's generalized beam strains must be found in terms of the element's shape functions and nodal values. The following strain-displacement relation matrix is defined, consistent with the selected theories for the different degrees of freedom:

$$[B] = \begin{bmatrix} \frac{d}{dz} & 0 & 0 & 0 & -1 & 0 \\ 0 & \frac{d}{dz} & 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{d}{dz} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{d}{dz} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{d}{dz} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{d}{dz} \end{bmatrix} \tag{3.9}$$

Using equations 3.6, 3.8, and 3.9, the element's generalized beam strains can be expressed as:

$$\{\epsilon\}_e = \left\{\epsilon_x \quad \epsilon_y \quad \epsilon_z \quad \kappa_x \quad \kappa_y \quad \kappa_z\right\}^T = [B][N]\{q\}_e \tag{3.10}$$

The internal virtual work is:

$$\delta W_{int} = \sum_{i=1}^{N} \left[\int_0^L \{\delta q\}_i^T [N]^T [B]^T [K]_s [B][N]\{q\}_i dz \right] \tag{3.11}$$

where $[K]_s$ is the beam cross-sectional stiffness matrix from Section 2.2.4, and N is the total number of elements used to approximate the beam. The external virtual work in the system can be expressed as:

$$\delta W_{ext} = \sum_{i=1}^{N} \left[\int_0^L \{\delta q\}_i^T [N]\{p\}_i dz \right] + \{\delta q\}_g^T \{P\} \tag{3.12}$$

where $\{q\}_g$ is the vector of the globally assembled beam nodal values. In addition $\{p\}_i$ is the distributed load over the element, and $\{P\}$ is the vector of loads that are applied to all of the global beam's nodes. They take the form:

$$\{p\}_i = \left\{p_x \quad p_y \quad p_z \quad m_x \quad m_y \quad m_z\right\}^T \tag{3.13}$$

$$\{P\} = \left\{P_{x_1} \quad P_{y_1} \quad P_{z_1} \quad M_{x_1} \quad M_{y_1} \quad M_{z_1} \quad \cdots \quad P_{x_N} \quad P_{y_N} \quad P_{z_N} \quad M_{x_N} \quad M_{y_N} \quad M_{z_N}\right\}^T \tag{3.14}$$

The total virtual work in the beam using equation 2.5 can be expressed as:

$$\delta W_{total} = \sum_{i=1}^{N} \left[\int_0^L \{\delta q\}_i^T [N] \{p\}_i dz - \int_0^L \{\delta q\}_i^T [N]^T [B]^T [K]_s [B][N] \{q\}_i dz \right] + \{\delta q\}^T \{P\} = 0 \tag{3.15}$$

In order for the variation of the total work to be equal to zero ignoring the trivial case where the variations in the displacements and rotations are zero, the following must be true:

$$[K]_g \{q\}_g = \{F\}_g + \{P\} \tag{3.16}$$

where $[K]_g$ and $\{F\}_g$ are the assembled global stiffness matrix and distributed force vector. The element stiffness matrix is:

$$[K]_e = \int_0^L [N]^T [B]^T [K]_s [B][N] dz \tag{3.17}$$

and the element distributed force vector is:

$$[F]_e = \int_0^L [N] \{p\} dz \tag{3.18}$$

Until this point, there has been no specific mention of what shape functions were used in the finite element formulation, although there are several viable options. For the torsion and axial degrees of freedom, two linear shape functions are sufficient to model each degree of freedom. Due to shear locking, additional consideration is required for the transverse bending and rotation degrees of freedom.

3.2.1 Overcoming Shear Locking

The true nature of shear locking is often poorly understood and is the go-to scapegoat when structural models behave incorrectly. Often shear locking introduces itself in models such as long and slender beams, models that are accurately handled by theories ignoring the additional shear deformation. It is not a problem with the model but with the discrete representation of the Timoshenko model [20]. With this numerical error in mind, consider a long and slender beam. The shear stiffness of this beam ($GA\kappa$) is significantly larger than the bending stiffness (EI) of the beam. If the mode shapes used are too coarse to correctly predict the behavior accurately, there will

be some error in both the transverse displacement and rotation. Since the magnitude of the shear stiffness is high and the displacement and shear strain are coupled, the small error in the shear strain propagates as a large error in the displacement.

For a more detailed example, consider a cantilevered beam with a moment applied at the tip. The solution of the transverse displacement of this beam takes the form $u(z) = z^2$:

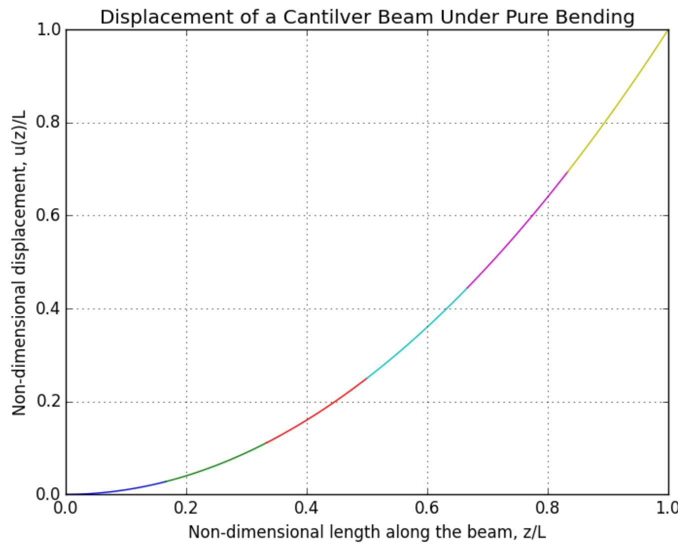


Figure 3.1: Non-dimensional displacement of a discretized cantilever beam in pure bending

The generalized beam shear strain can be written as:

$$\gamma_x = \frac{du}{dz} - \psi \tag{3.19}$$

The plot of beam generalized strain shows that on average over the elements, the shear strain for the beam is zero:

When calculating the energy due to the shear deformation, the shear strain is squared. Therefore even though the integral of the strain over the element is zero, the integral of the square of the strain is not, adding a fictitious amount of strain energy to the system. Even if this error is small, when it is multiplied by the shear stiffness $GA\kappa_x$ which has a large magnitude, what was a small error in the shear strain becomes a large error in the shear strain energy.

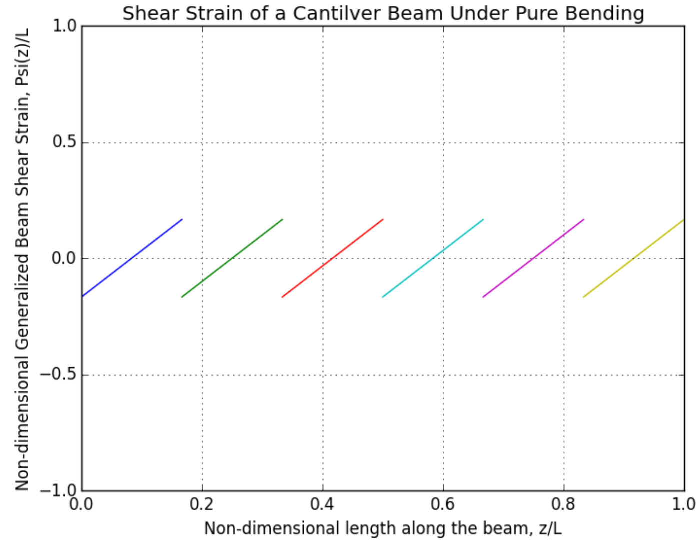


Figure 3.2: Non-dimensional shear strain of a discretized cantilever beam in pure bending

When a certain amount of work is done on the system, the system in turn gains an equal and opposite amount of potential strain energy. Due to the error in the strain displayed in figure 3.2, a certain amount of the external work must go to the fictitious shear strain energy, reducing the amount that can go towards the bending strain energy. As a result, the beam will shear incorrectly and exhibit a significant reduction in the transverse displacement. For further explanation on the shear locking phenomenon see Reddy [20].

Many strategies to overcome this numerical error have been constructed and the resulting leading solutions are either Reduced Integration Elements (RIE) or Consistent Integration Elements (CIE) [20]. With CIE, the Lagrange polynomials used for the transverse displacement ($u(z)$ in the example above) must be an order higher than that of the transverse rotation shape functions ($\gamma(z)$ in the example above). From the strain-displacement relation in equation 3.9, it is clear that $u(z)$ and $\gamma(z)$ must be differentiable once, and so at minimum linear shape functions are needed for both $u(z)$ and $\gamma(z)$. Since $u(z)$ must be an order higher polynomial then $\gamma(z)$, the result is that $u(z)$ can be expressed with quadratic shape functions and $\gamma(z)$ can be expressed with linear shape functions. Using this approach, the resulting beam element will be free of shear locking.

The other approach and that used by AeroComBAT is the RIE method. In this case, linear shape functions are used for both the transverse deflection and rotation degrees of freedom. If these are integrated exactly as was already mentioned, this would undoubtedly result in shear-locking elements. Rather than integrating the terms exactly however, only the shear terms ($GA\kappa\gamma^2$) will be integrated approximated with a single point Gauss Quadrature integration. Recall that for all of the beam degrees of freedom, two linear shape functions were used. As a result, it turns out that all of the other terms in the matrix can be evaluated exactly with a single Gauss point integration. Therefore the entire AeroComBAT stiffness matrix $[K_e]$ can be evaluated as:

$$[K]_e = w[N]^T[B]^T[K][B][N]\Big|_{z=z_{transf}(0)} J^{-1} \tag{3.20}$$

where $w = 2$ for a single Gauss Quadrature evaluation, $z_{transf} = \frac{h}{2}(\xi + 1)$ and is the function that maps ξ onto z, and $J = \frac{dz}{d\xi} = \frac{2}{h}$. Note that all terms of the distributed load vector in equations 3.18 must be integrated exactly. Using this formulation, AeroComBAT can accurately and efficiently calculate the displacements and rotations of any beam (slender or stubby) without exhibiting any shear locking.

3.2.2 Arbitrarily Oriented Beam

The finite element beam formulation has the capability to correctly calculate displacements and rotations for stubby and slender composite beams. Unfortunately without some additional massaging of the formulation, these elements are only valid if the beams run in the z-direction. Like with the cross-section stiffness matrix, a method to transform the beam element stiffness matrix must be achieved. For plane frame elements this is trivial and is extensively documented, and while the extension to 3D is not much more complex, there are few if any locations in which this is documented [1]. The most difficult step is to generate the rotation matrix to rotate the nodal degrees of freedom from the local element frame to the global frame. In order to uniquely define the local element frame, two vectors are needed. The first and most trivial is the unit vector pointing along the beam axis ($\hat{h}$). The second vector must be chosen by the user. This unit vector will uniquely identify the direction of the x-axis for the beam element in the global frame ($x_{\hat{local}}$). With these two vectors, the rotation matrix can be calculated as:

$$A = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} \tag{3.21}$$

where

$$\hat{w} = \hat{h} \tag{3.22}$$

$$\hat{v} = \frac{\hat{h} \times \hat{x_{local}}}{\|\hat{h} \times \hat{x_{local}}\|} \tag{3.23}$$

$$\hat{u} = \frac{\hat{v} \times \hat{w}}{\|\hat{v} \times \hat{w}\|} \tag{3.24}$$

Having calculated the transformation matrix A for a single point, the transformation matrix for all of the nodal displacements and rotations can be expressed as:

$$T = \begin{bmatrix} A & 0 & 0 & 0 \\ 0 & A & 0 & 0 \\ 0 & 0 & A & 0 \\ 0 & 0 & 0 & A \end{bmatrix} \tag{3.25}$$

The local element stiffness matrix can be written in the global frame using:

$$[K_e]_{global} = [T]^T [K_e][T] \tag{3.26}$$

Keep in mind that there are many ways in which the A rotation matrix could be calculated should a different approach be desired.

3.3 Normal Modes Finite Element Formulation

In order to conduct a dynamic flutter aeroelastic analysis, it is necessary to consider the free-vibration normal mode response of the beam as well. For the normal mode finite element formulation, the Lagrange's Equations are applied to the system:

$$\frac{\partial}{\partial t}\left(\frac{\partial(T-U)}{\partial \dot{q}_i}\right)-\frac{\partial(T-U)}{\partial q_i}=Q \tag{3.27}$$

where T is the kinetic energy in the beam, U is the potential energy in the beam, q_i and $\dot{q}_i$ are the generalized coordinates and the first time derivatives of the generalized coordinates respectively, and Q are the generalized forces. For a normal modes analysis, there are no forces, so Q goes to zero. The kinetic energy can be expressed as:

$$T=\frac{1}{2}\int_0^L\sum_{i=1}^{N}\{\dot{q}\}_i^T[N]^T[M][N]\{\dot{q}\}_i dz \tag{3.28}$$

The internal strain potential energy in the beam can be expressed as:

$$U=\frac{1}{2}\int_0^L\sum_{i=1}^{N}\{q\}_i^T[N]^T[B]^T[K]_s[B][N]\{q\}_i dz \tag{3.29}$$

Substituting the expressions for the kinetic and potential energy into 3.27, Lagranges equations become:

$$[M]_g\{\ddot{q}\}+[K]_g\{q\}=0 \tag{3.30}$$

where $[K]_g$ is the global stiffness matrix from Section 3.2 and $[M]_g$ is the global mass matrix. $[M]_g$ can be created using a typical finite element assembly of the element mass matrix $[M]_e$:

$$[M_e]=\int_0^h[N]^T[M]_s[N]dz \tag{3.31}$$

where $[M]_s$ is the cross-section mass matrix defined in Section 2.2.4. Note that the element mass matrix can also be transformed to the global frame using the approach from equations 3.26. Finally, assuming simple harmonic motion such that $\{q\}=\{\bar{q}\}e^{i\omega t}$, the free vibration equation can be expanded as:

$$-\omega^2[M]_g\{\bar{q}\}+[K]_g\{\bar{q}\}=0 \tag{3.32}$$

The resulting problem becomes a typical eigenvalue problem in which the eigenvalues are the squares of the natural frequencies, and the eigenvectors are the mode shapes associated with the corresponding natural frequency.

3.4 Timoshenko Beam Formulation Verification

In order to verify the global displacement and rotation behavior of the AeroComBAT beam model, a beam was generated with a length of 20 meters. The beam used a hollow box beam cross-section with a width and height of 1.798 and 1.0 meters respectively. The thickness from the outer mold line was 0.04 meters. The only modification to the cross-section was that an isotropic Aluminum 6061-T6 was used as the material instead of the AS4-3501-6 unidirectional carbon. The Young's Modulus used was $E = 71.7$ GPa with a Poisson's ratio of $\nu = 0.33$ and a density of $\rho = 2810kg/m^3$. To verify the static as well as normal mode solutions of this beam, an identical model was generated in NASTRAN using CBEAM elements. For both the static and normal mode case, the beam was fixed at its root. In addition, 40 evenly spaced elements were used along the length of the two beams.

3.4.1 Linear Static Verification

For the linear static case, analytical solutions of the governing differential equations including the traditional axial, torsional, and transverse Timoshenko bending were solved since the simple geometry and use of an isotropic material left resulting ODE's fairly easy to solve. In order to keep the solutions to the differential equations simple, a constant distributed load of $[p_x, p_y, p_z, t_z] = [1000N/m, 1000N/m, -1000N/m, 1000N]$ was considered over the domain of the beam. The error in both models were taken with respect to the analytical solutions to the differential equations. The error of the two models can be seen in the figures below.

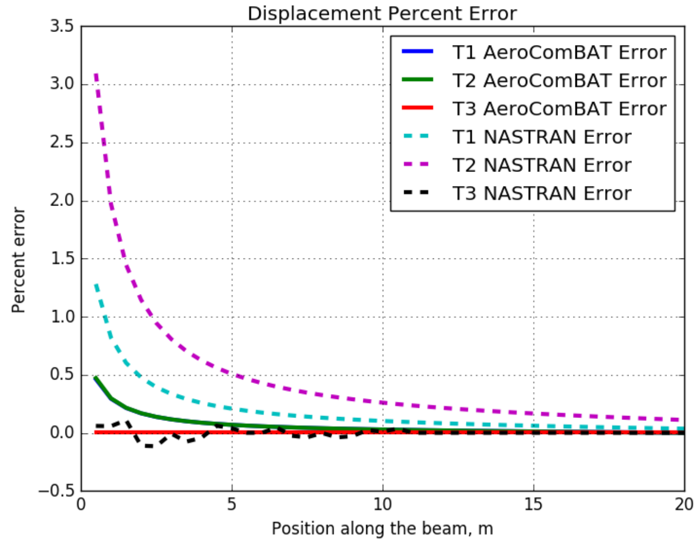


Figure 3.3: The finite element displacement error for NASTRAN and AeroComBAT

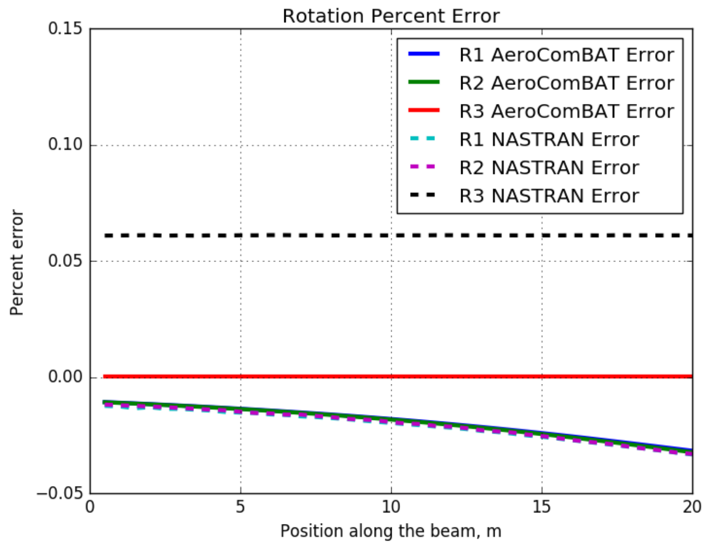


Figure 3.4: The finite element rotation error for NASTRAN and AeroComBAT

From figures 3.3 and 3.4 it is clear that not only have the displacements and rotations of the AeroComBAT beam sufficiently converged, but in gen-

eral the error in the displacements and rotations in the AeroComBAT beams are lower than those of their NASTRAN counter parts. The NASTRAN documentation indicates that the CBEAM element is shear deformable, which would imply that the additional shear flexibility should be accounted for, and should not be the source of the error. After further investigation, it was discovered that while the the shear reduction factors κ_x and κ_y as well as the torsion constant J were inaccurately calculated by the pre/post processer used, which was FEMAP. Once these factors were corrected with the corresponding values from AeroComBAT, the following results were produced:

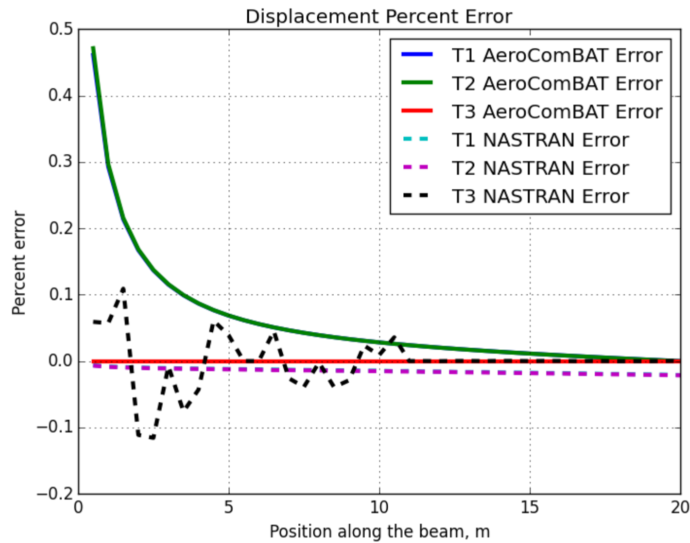


Figure 3.5: Displacement error for NASTRAN and AeroComBAT with corrected shear terms

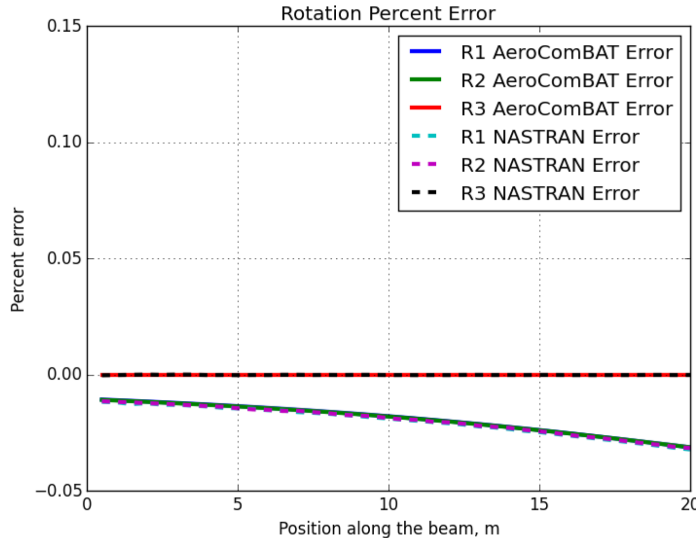


Figure 3.6: Rotation error for NASTRAN and AeroComBAT with corrected torsion term

With the updated shear and torsional stiffness terms, NASTRAN's CBEAM results are much more accurate for the transverse displacement, edging out AeroComBAT's error by a significant amount. One curious aspect of NASTRAN's CBEAM elements is that with the corrected stiffnesses, the transverse displacements appear to be more flexible than the analytical result by a small amount.

For a more for a more rigorous comparison of the global displacements and rotations, the composite beam from Section 2.4 is reused. The same boundary conditions are also used, being that a force of $[F_x, F_y, F_z] = [444N, 444N, 44444N]$ is applied at the tip while the other end of the beam is fixed. Two sets of constraints are considered for the nodes on the solid model. In the first, the nodal displacements at the end of the beam are fixed with a RBE2 elements such that no cross-section warping can occur. In addition, an RBE2 element is used to distribute the tip load throughout the cross-section. In the second case the end is fixed with a RBE3 element such that cross-section warping can occur. Similarly, an RBE3 element is used to distribute the tip load throughout the cross-section. Referring back to Section 2.4 the discovery was made that while RBE3 elements does allow cross-section warping, they enforce incorrect cross-section warping. Finally it should be noted that the generalized beam displacements and rotations for the solid NASTRAN model

were calculated using equations 2.40 and 2.41.

The comparison of the results between the AeroComBAT beam model and the NASTRAN solid model with warp restraint at the ends can be seen below in figures 3.7 and 3.8:

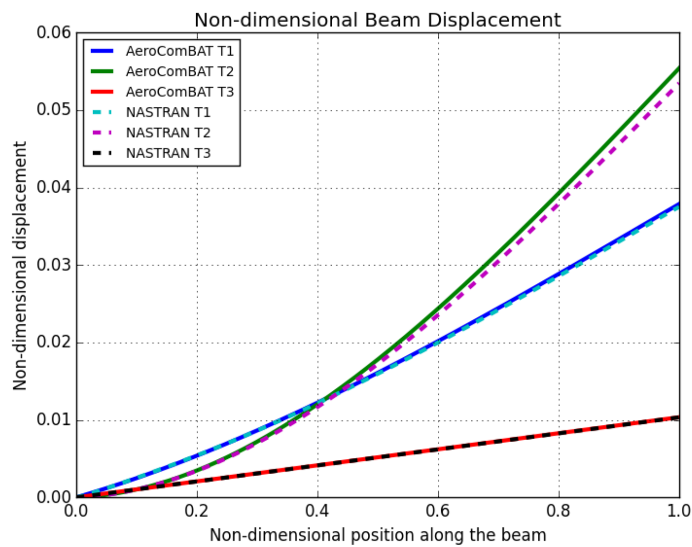


Figure 3.7: Displacement comparison with warp restraint

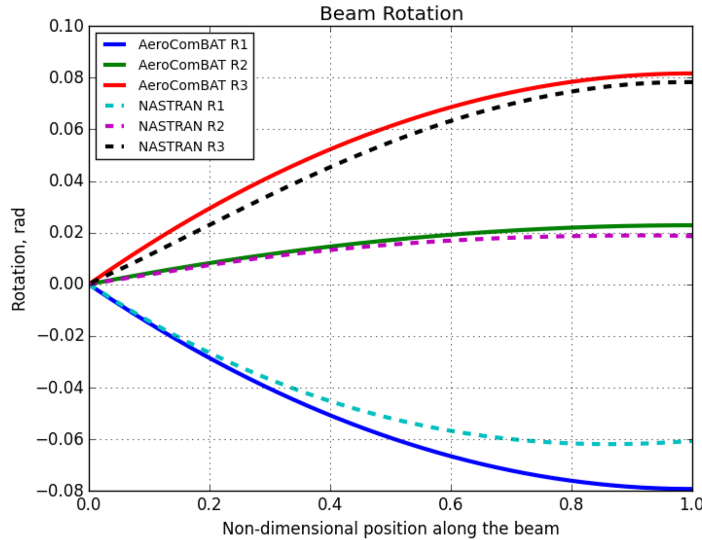


Figure 3.8: Rotation comparison with warp restraint

From figures 3.7 and 3.8, it is clear that the AeroComBAT has done a decent job at matching the displacements and rotations of the solid model. Since the displacements and rotations of the AeroComBAT model tend to be higher than the solid model, it implies that the AeroComBAT model is more flexible. Knowing that stiffness is added to the model due to the RBE2 imposed warp restraint at the tips, this result is not too surprising. The error in the rotations associated with the transverse displacements tend to be higher, however this would also tend to make sense since it was noted in subsection 3.2.1 that a small error in the displacements will propagate as a large error in the corresponding beam rotation.

The comparison of the results between the AeroComBAT beam model and the NASTRAN solid model with incorrectly enforced warping at the ends can be seen below in figures 3.9 and 3.10:

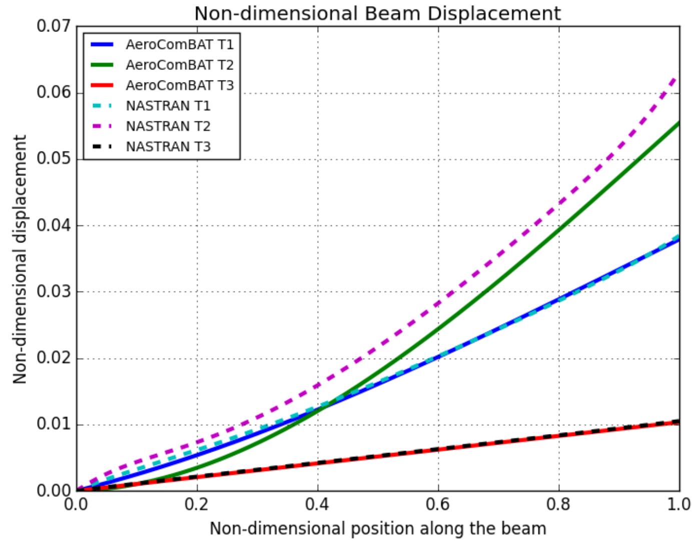


Figure 3.9: Displacement comparison with incorrectly enforced warping

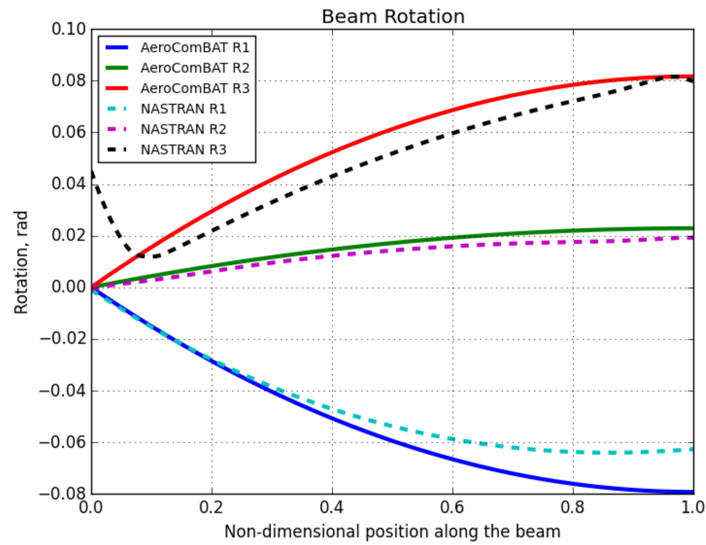


Figure 3.10: Rotation comparison with incorrectly enforced warping

To reiterate, it was shown in Section 2.4 that the warping at the ends of the beams is not correct, and as such there are still warping-enforced effects as can be clearly seen in figures 3.9 and 3.10. The AeroComBAT predictions are moderately close to the warping-enforced NASTRAN model, however it

is clear that the warping enforced on the ends of the solid model produce significant discrepancies. The results from figures 3.9 and 3.10 are puzzling. For some degrees of freedom the AeroComBAT model appears to be too stiff, while for others it appears to be to soft.

The comparison of the results between the AeroComBAT beam model, the solid model with warp restraint, and the solid model with a warping enforced conveys two clear concepts. In general AeroComBAT is capable of calculating the global displacements and rotations of a composite beam. Without the ability to capture warp-restraint effects, it is clear that AeroComBAT's accuracy can suffer when these effects are not negligible.

3.4.2 Dynamic Normal Modes Analysis

In order to verify the accuracy of the normal modes solution employed by AeroComBAT, the aluminum beam models first employed in subsection 3.4.1 are reused. Seen below in table 3.1 are the first 9 tabulated frequencies of the two models. Note, all frequencies are in hertz.

Table 3.1: Box beam verification layup schedules

Model	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7	ω_8	ω_9
NASTRAN	2.94	4.63	17.31	27.24	30.84	45.00	63.14	69.92	78.94
AeroComBAT	2.93	4.63	17.25	26.91	30.85	44.34	63.15	68.51	78.55

It is clear that at a first glance the AeroComBAT normal modes solution is sufficiently accurate. From subsection 3.4.1, it was shown that under the same loading, AeroComBAT and NASTRAN produce comparable results, and from the Table 3.1, it is clear that the differences in frequency are negligible.

Like with the linear static analysis, a more rigorous comparison to a solid model was undertaken. For the comparison of AeroComBAT to a solid model, the composite box beam models described in subsection 3.4.1 are reused. Due to the incorrect warping imposed by the RBE3 elements, both ends of the NASTRAN solid model will be constrained with RBE2 elements. The density of the $AS4/3501^*$ material used was $\rho = 18,915kg/m^3$. The tabulated natural frequency's of the first six modes can be seen below:

Table 3.2: Box beam verification layup schedules

Model	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6
NASTRAN	137.96	249.18	609.68	705.54	706.72	766.72
AeroComBAT	135.96	244.62	670.36	841.28	1119.17	1582.99

From table 3.2, it can be seen that after the first two natural frequencies the two models differ fairly greatly. Upon investigating the mode shapes of the NASTRAN model, the difference in results became clear: The dynamic motion of the cross-section was not considered with the AeroComBAT model. Below are the mode shapes corresponding to the NASTRAN ω_3 and ω_4 modes:

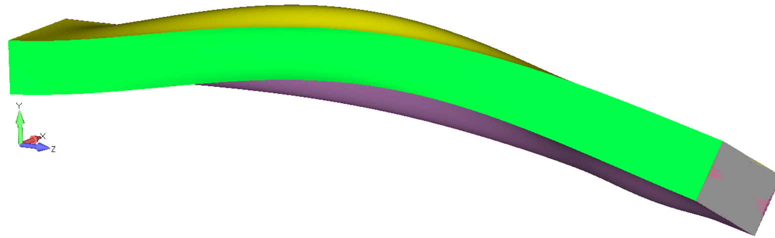


Figure 3.11: The third natural mode of the solid NASTRAN model exhibits a great amount of dynamic motion within the beam's cross-section

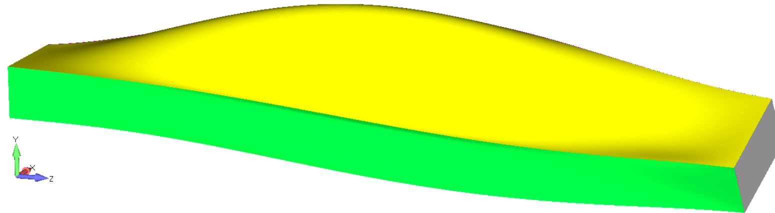


Figure 3.12: The third natural mode of the solid NASTRAN model exhibits a great amount of dynamic motion within the beam's cross-section

The mode shapes seen in figure 3.11 and 3.12 exhibit some of the same in-plane bending which was theorized to be artificially introduced by incorrect displacement boundary conditions imposed at the ends of the beam. It is possible that with the correct boundary conditions, the higher mode shapes where the torsional effects are non-negligible for this beam would match more closely. In order to further answer that question, a more significant investigation into the warp-restraint and the dynamic cross-section warping effects would need to be undertaken.

3.5 Conclusion

In this chapter, a simple Timoshenko beam formulation for small displacements and rotations was introduced, and the models efficacy was measured by comparing it against a NASTRAN CBEAM model and a NASTRAN solid CHEXA model. When compared to the CBEAM model, the AeroComBAT Timoshenko formulation proved adequate. Both the static and dynamic normal mode results all exhibited similar results. In fact for the linear linear static displacements and rotations, the results produced by AeroComBAT were slightly more accurate when compared to the analytical beam results. When compared with the solid model however, AeroComBAT had more difficulty. For the linear static case, AeroComBAT was able to match the global behavior of the solid composite beam when it had warp restrain imposed at both ends. AeroComBAT struggled even more so with the dynamic normal mode comparison for the full composite beam, only managing to capture the first two modes. From the aluminum box-beam case and the composite box beam case, it is clear that accuracy of AeroComBAT's results are dependent and in some cases sensitive to the displacement boundary conditions of the model, a sentiment that echoed by the results from Section 2.4.

Chapter 4

Unsteady Aerodynamic Model

4.1 Introduction

In order to consider the dynamic aeroelastic behavior of a structure, three contributions to this behavior must be considered: the inertial forces of the structure, the elastic response of the structure, and the aerodynamic forces acting on the structure. In Chapter 2 and Chapter 3, models for both the elastic response and the inertia of a beam structure were created. What remains is the need for an unsteady aerodynamic model. One of the earliest aerodynamic models considered was the strip theory model [13] [3]. This model takes the 2D aerodynamic solution for a wing and extends it throughout the length of the wing. In order to capture the unsteady nature of the aerodynamics, unsteady Theodorsen aerodynamics are often used in conjunction with strip theory. Unfortunately this method fails to take into account the 3D flow effects which can have a substantial impact on the loading of the wing. Furthermore when sweep must be considered an additional range of terms are required in order to account for an increased angle of attack due to transverse deflection of the wing [18].

The next advancement of the aerodynamic model came when Albano and Rodden [2] proposed the Doublet-Lattice method in 1969. This panel method uses a superposition of elementary flow solutions to capture the behavior of the fluid. Albano and Rodden were able to use the linearized potential flow equations using doublet line singularity elements to accurately capture the 3D unsteady pressure distributions for thin wings. The mathematics and history behind the development of the Doublet-Lattice method has been extensively

documented [2] [4] [17] [23] [15]. As such, the implementation of the method will be the focus of this chapter.

4.1.1 Brief Overview of the Kernel Function

The kernel function used to evaluate the doublet lattice method is at its core the evaluation of the elementary potential flow solutions used to solve the linearized unsteady potential flow equations seen below [7]:

$$(1 - M_\infty^2)\phi_{xx} + \phi_{yy} + \phi_{zz} - 2\left(\frac{M_\infty^2}{U_\infty}\right)\phi_{tt} = 0 \tag{4.1}$$

where the function $\phi(x, y, z, t)$ is the disturbance velocity potential such that:

$$\vec{u} = U_\infty \hat{i} + \nabla\phi(x, y, z, t) \tag{4.2}$$

and is derived from the Euler equations. Equation 4.1 can then be transformed into the frequency domain by assuming simple harmonic oscillations where:

$$\phi(x, y, z, t) = \bar{\phi}(x, y, z)e^{i\omega t} \tag{4.3}$$

substituting this assumption into equation 4.1 yields:

$$(1 - M_\infty^2)\bar{\phi}_{xx} + \bar{\phi}_{yy} + \bar{\phi}_{zz} - 2ikM_\infty^2\bar{\phi}_x + k^2M_\infty^2\bar{\phi} = 0 \tag{4.4}$$

where k is the reduced frequency. Elementary potential flow solutions can be superimposed to solve equation 4.4 while ensuring that their disturbances go to zero at infinity and that they also satisfy the non-penetration boundary condition on the lifting surfaces. For the full derivation, see Dowell [8]. In the following section, the implementation of the doublet lattice method is discussed.

4.2 Implementation of the Doublet Lattice Method

The doublet lattice method has been used extensively for the flutter analysis of wings (both planar and nonplanar systems) and yields reliable 3D pressure distributions at a relatively low cost. The first step in implementing this method is to discretize the wing. Bear in mind that one of the restrictions

involved with the doublet lattice method is that it requires that the chord lines always run parallel to the flow (x-axis). In addition unlike the vortex lattice method for steady aerodynamics, no panel may have a geometric angle of attack with the flow as seen in the figure below:

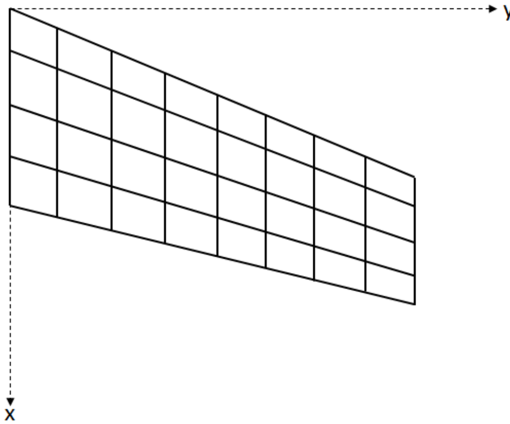


Figure 4.1: A lifting surface discretized for the doublet-lattice method

The geometry of the individual panels can also be seen below in figure 4.2.

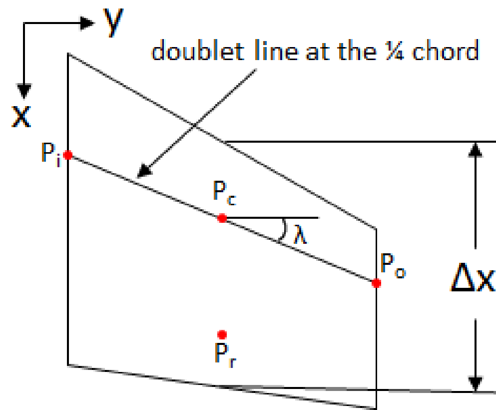


Figure 4.2: The detailed geometry of a single doublet panel

Let us first jump ahead to the end goal in order to understand where we are headed. The generalized forces due to the aerodynamics can be expressed as:

$$\{F\}_{AIC} = \frac{1}{2}\left(\frac{\rho}{\rho_0}\right)\rho_0 U^2 [B][D]^{-1}[W]\{h\} \tag{4.5}$$

In Equation 4.5, the density ratio term $\frac{\rho}{\rho_0}$ is there to facilitate flutter analysis at different altitudes where ρ_0 is the density at sea level. In addition, U is the free-stream velocity, and $\{h\}$ is a vector of the generalized coordinates of the structural model. $[W]$ (the substantial derivative matrix) is a matrix that maps the degrees of freedom of the structural model to the aerodynamic control points of the aerodynamic model. Using this matrix, the velocity in the structural model can be translated into a perceived downwash by the aerodynamic model. $[D]^{-1}$ is a matrix that relates the downwashes at the control points on panels to the non-dimensional pressure differences experienced by those panels. $[B]$ (the integration matrix) is a matrix that integrates the pressures of the panels and applies them as equivalent forces and moments back onto the structural model. The real challenge in the doublet lattice method is calculating the non-dimensional downwash-pressure relation matrix ($[D]$). Section 4.3 will layout how to solve for $[W]$ and $[B]$.

4.2.1 The Downwash-Pressure Relation

Referring back to the geometry of the panels used to discretize the wing in Figure 4.2, it can be seen the doublet line runs along the chord length of the panel at the quarter chord. The amplitude of the doublet strength of the panel is:

$$\frac{\bar{f}}{4\pi\rho}\int_l d\mu \tag{4.6}$$

where l is the length of the doublet line, $d\mu$ is the incremental length, ρ is the density of the flow and $\bar{f}$ is the magnitude of the doublet line strength. The downwash velocity induced at some point x_i due to the doublet line on the panel can be expressed with the Kernel function as:

$$\bar{w}(x_i) = \frac{\bar{f}}{4\pi\rho U^2}\int_l K(x_i, x(\mu), k, M)d\mu \tag{4.7}$$

where M is the Mach number of the flow and k is the reduced frequency which is defined below:

$$k = \frac{\omega b}{U} \tag{4.8}$$

where ω is the circular frequency of oscillations of the panel and b is the reference semi-chord. The effect of the single doublet panel on the downwash

at point x_i in equation 4.7 can be generalized to be the downwash due to N total panels:

$$\bar{w}(x_i) = \sum_{j=1}^{N} \frac{\bar{f}_j}{4\pi\rho U^2} \int_{l_j} K(x_i, x_j(\mu), k, M) d\mu \tag{4.9}$$

Furthermore the application of equation 4.9 N times results in N equations. These equations can then be used to solve for the N unknown doublet line strengths $\bar{f}_j$. Knowing that the non-dimensional pressure difference over any of the panels can be expressed as:

$$\bar{p_j} = \frac{\bar{f}_j}{\Delta x_j cos(\lambda_j)} \tag{4.10}$$

Using equation 4.10, equation 4.9 can be simplified to:

$$\bar{w}(x_i) = \sum_{j=1}^{N} \frac{\Delta x_j cos(\lambda_j)}{8\pi} \int_{l_j} K(x_i, x_j(\mu), k, M) d\mu \tag{4.11}$$

The contribution of each jth panel to the downwash at an ith location can become an entry D_{ij} in the $[D]$ where:

$$\{\bar{w}\} = [D]\{\bar{p}\} \tag{4.12}$$

and

$$D_{ij} = \frac{\Delta x_j cos(\lambda_j)}{8\pi} \int_{l_j} K(x_i, x_j(\mu), k, M) d\mu \tag{4.13}$$

Since the doublet lattice method can in general be applied to non-planar surfaces, it is imperative to create a new, local coordinate system defined at the sending point of sending source doublet panel such that:

$$\eta = y\cos(\gamma_{S_c}) + z\sin(\gamma_{S_c}) \qquad \xi = -y\sin(\gamma_{S_c}) + z\cos(\gamma_{S_c}) \tag{4.14}$$

where γ_{S_c} is the local dihedral of the wing at the sending point. Finally in order to simplify the integration process, the kernel function K will be expanded as:

$$K = \frac{\bar{K}}{r^2} = \frac{\bar{K}}{(\eta_0 - \eta)^2 + (\xi)^2} \tag{4.15}$$

Using the expansion of equation 4.15, the integral from equation 4.13 can be further expressed as:

$$I_{ij} = \int_{l_j} \frac{\bar{K}(x_i, x_j(\mu), k, M)}{(\eta_0 - \eta)^2 + (\xi)^2} cos(\lambda_j) d\mu \approx \int_{-e}^{e} \frac{A\eta^2 + B\eta + C}{(\eta_0 - \eta)^2 + (\xi)^2} \tag{4.16}$$

where:

$$e = \frac{1}{2} l_j cos(\lambda_j) \tag{4.17}$$

$$\begin{aligned} \eta_0 &= (y_R - y_{S_c})cos(\gamma_{S_c}) + (z_R - z_{S_c})sin(\gamma_{S_c}) \\ \xi_0 &= -(y_R - y_{S_c})sin(\gamma_{S_c}) + (z_R - z_{S_c})cos(\gamma_{S_c}) \end{aligned} \tag{4.18}$$

$$A = \frac{\bar{K}_i - 2\bar{K}_c + \bar{K}_o}{2e^2} \qquad B = \frac{\bar{K}_o - \bar{K}_i}{2e} \qquad C = \bar{K}_c \tag{4.19}$$

and $\bar{K}_i$, $\bar{K}_c$, and $\bar{K}_o$ are the evaluations of the modified kernel function at points S_i, S_c, and S_o respectively. Having approximated the modified kernel function as a quadratic function in η, the integral 4.16 can be evaluated as:

$$\begin{aligned} I_{ij} = \left(\left(\eta_0^2 - \xi_0^2\right) A + \eta_0 B + C\right) |\xi_0|^{-1} \tan^{-1}\left(\frac{2e|\xi_0|}{r_1^2 - e^2}\right) + \\ \left(\frac{B}{2} + \eta_0 A\right) ln\left(\frac{r_1^2 - 2\eta_0 e + e^2}{r_1^2 + 2\eta_0 e + e^2}\right) + 2eA \end{aligned} \tag{4.20}$$

where $r_1^2 = \eta_0^2 + \xi_0^2$. Notice from equation 4.18 that if there is no dihedral in the sending box (i.e. the sending box is parallel with the x-y plane), then the term $\xi_0 \to 0$ and $\xi_0^{-1} \to \infty$. For the planar case, the integration of equation 4.16 instead becomes:

$$I_{ij} = \left(\eta_0^2 A + \eta_0 B + C\right) \left(\frac{1}{\eta_0 - e} - \frac{1}{\eta_0 + e}\right) + \left(\frac{B}{2} + \eta_0 A\right) ln\left(\left(\frac{\eta_0 - e}{\eta_0 + e}\right)^2\right) + 2eA \tag{4.21}$$

Having successfully derived the a method to evaluate the integral, the D_{ij} element can now be expressed as:

$$D_{ij} = \frac{\Delta x_j cos(\lambda_j)}{8\pi} I_{ij} \tag{4.22}$$

While we no longer need to integrate the modified kernel function due to the quadratic approximation, we still need to evaluate the modified kernel function at the inboard, center, and outboard points for any given panel.

4.2.2 Evaluating the Modified Kernel Function

The kernel function relating the non-dimensional pressure acting on a panel to the downwash at another location is:

$$K = \frac{e^{-i\frac{kx_0}{b}}(K_1T_1 + K_2T_2)}{r^2} \tag{4.23}$$

The modified kernel function may be expressed as:

$$\bar{K} = Kr^2 = e^{-i\frac{kx_0}{b}}(K_1T_1 + K_2T_2) \tag{4.24}$$

where:

$$K_1 = I_1 + \frac{Mr_1e^{-ik1u1}}{R^2(1+u_1^2)^{1/2}} \tag{4.25}$$

$$K_2 = -3I_2 - \frac{ik_1M^2r_1^2e^{-ik1u1}}{R^2(1+u_1^2)^{1/2}} - \frac{Mr_1}{R}\left((1+u_1^2)\frac{\beta^2r_1^2}{R^2} + \frac{Mr_1u_1}{R} + 2\right)\frac{e^{-ik_1u_1}}{(1+u_1^2)^{3/2}} \tag{4.26}$$

$$T_1 = cos(\gamma_R - \gamma_{S_c}) \tag{4.27}$$

$$T_2 = \frac{(z_0cos(\gamma_R) - y_0sin(\gamma_R))(z_0cos(\gamma_{S_c}) - y_0sin(\gamma_{S_c}))}{r_1^2} \tag{4.28}$$

$$I_1 = \int_{u_1}^{\infty} \frac{e^{-ik1u}}{(1+u^2)^{3/2}}du \tag{4.29}$$

$$I_2 = \int_{u_1}^{\infty} \frac{e^{-ik1u}}{(1+u^2)^{5/2}}du \tag{4.30}$$

$$u1 = \frac{MR - x_0}{\beta^2 r_1} \tag{4.31}$$

$$\beta = \sqrt{1 - M^2} \tag{4.32}$$

$$k_1 = \frac{r_1k}{b} \tag{4.33}$$

$$R = \sqrt{x_0 + \beta^2r_1^2} \tag{4.34}$$

Evaluating equations 4.29 and 4.30 is still fairly problematic since they are in the form of indefinite integrals. As such, the integrations can be approximated as:

$$I_1(u_1, k_1) = e^{-ik_1u_1}\left(1 - \frac{u_1}{\sqrt{1+u_1^2}} - ik_1I_0(u_1, k_1)\right) \tag{4.35}$$

$$3I_2(u_1,k_1) = e^{-ik_1u_1}\left((2+ik_1u_1)\left(1-\frac{u_1}{\sqrt{1+u_1^2}}\right) - \frac{u_1}{(1+u_1^2)^{3/2}} - ik_1I_0(u_1,k_1) + k_1^2J_0(u_1,k_1)\right) \tag{4.36}$$

In this case, functions I_0 and J_0 are yet another pair of indefinite integrals, however they can be evaluated numerically as:

$$I_0(u_1,k_1) = \sum_{n=1}^{1} 1\frac{a_n e^{ncu_1}}{n^2c^2+k_1^2}(nc-ik) \tag{4.37}$$

$$J_0(u_1,k_1) = \sum_{n=1}^{1} 1\frac{a_n e^{ncu_1}}{(n^2c^2+k_1^2)^2}(n^2c^2-k^2+ncu_1(n^2c^2+k_1^2)-ik(2nc+u_1(n^2c^2+k^2))) \tag{4.38}$$

The constant $c = 0.372$ and the vector a is:

$$a = \begin{Bmatrix} 0.24186198 \\ 24.991079 \\ -111.59196 \\ 271.43549 \\ -305.75288 \\ -41.18363 \\ 545.98537 \\ -644.78155 \\ 328.72755 \\ -64.279511 \end{Bmatrix}^T \tag{4.39}$$

It is important to keep in mind that depending on whether the term u_1 is positive or negative, the approximations of $I_1(u_1,k_1)$ and $3I_2(u_1,k_1)$ are different. If $u_1 \geq 0$, then $I_1(u_1,k_1)$ and $3I_2(u_1,k_1)$ are evaluated with equations 4.35 and 4.36 respectively. Otherwise if $u_1 < 0$, then:

$$I_1(u_1,k_1) = 2Re(I_1(0,k_1)) - Re(I_1(-u_1,k_1)) + iIm(I_1(-u_1,k_1)) \tag{4.40}$$

$$3I_2(u_1,k_1) = 2Re(3I_2(0,k_1)) - Re(3I_2(-u_1,k_1)) + iIm(3I_2(-u_1,k_1)) \tag{4.41}$$

Finally refer back to equation 4.31 that when $r_1 \to 0$, then $u_1 \to \infty$. In this case, if $x_0 > 0$

$$\bar{K} = 2e^{-i\frac{kx_0}{b}} \tag{4.42}$$

Otherwise if $r_1 = 0$ and $x_0 < 0$, then:

$$\bar{K} = 0 \tag{4.43}$$

4.3 The Substantial Derivative and Integration Matrices

From a big picture perspective, what the $[D]$ matrix does is relate non-dimensional pressure differences acting over the surfaces of each panel to non-dimensional downwashes on those panels. This on it's own might not seem very useful, however consider $[D]^{-1}$ which is ultimately what is used to calculate the wings generalized aerodynamic forces. If the motion of the control points on the doublet panels can be related to the dynamic motion of the structural model, then this information can be used to determine the resulting downwash created by a particular dynamic motion. In essence this is what the substantial derivative matrix does. It is a way of linking the structural model to the aerodynamic panel model such that when the structure exhibits some dynamic motion, the aerodynamic model moves with it creating calculable downwashes on the panels control points. When used in conjunction, the $[W]$ and $[B]$ matrix serve the same purpose as the splines which are used extensively in common commercial packages, such as ZAERO™ [24]

4.3.1 The Substantial Derivative Matrix

In order to connect the motion of the structure to the motion of a panel control point, it is easiest to consider a chordwise strip of panels on the lifting surface. This can be seen in the figure below:

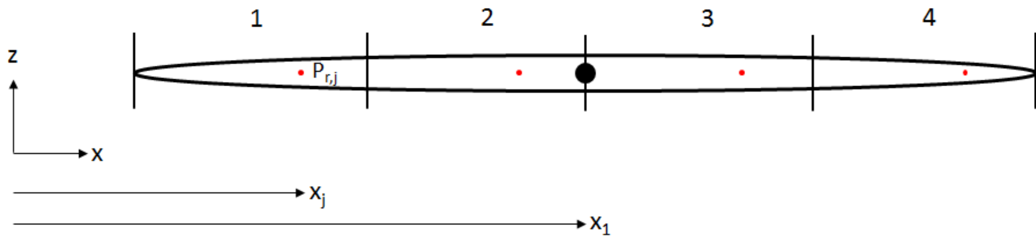


Figure 4.3: Relating the aerodynamic model to the structural model

For the jth panel, the downwash induced at a control point can be expressed as:

$$\bar{w}_j = ih\frac{k}{b} + \alpha + i\alpha\frac{k}{b}(x_j - x_1) \tag{4.44}$$

Translating the degrees of freedom of the structural model to the aerodynamic degrees of freedom, Equation 4.44 becomes:

$$\bar{w}_j = i(-w_n)\frac{k}{b} + (\gamma_n) + i(\gamma_n)\frac{k}{b}(x_j - x_1) \tag{4.45}$$

where w_n is the z displacement of the structural model at the n node, and γ_n is the rotation of the structural model about the y-axis at node n. Unless the aerodynamic panel control points match up perfectly with the nodes of the structural model, it is likely that most control points will lie in between elements. As such, the displacements and rotations can be interpolated from the nodes of the element. Equation 4.45 can be further expanded in matrix form as:

$$[W] = \begin{bmatrix} 0 & 0 & -i\frac{k}{b} & 0 & 1+i\frac{k}{b}(x_j-x_1) & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -i\frac{k}{b} & 0 & 1+i\frac{k}{b}(x_j-x_1) & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -i\frac{k}{b} & 0 & 1+i\frac{k}{b}(x_j-x_1) & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -i\frac{k}{b} & 0 & 1+i\frac{k}{b}(x_j-x_1) & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & -i\frac{k}{b} & 0 & 1+i\frac{k}{b}(x_j-x_1) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & -i\frac{k}{b} & 0 & 1+i\frac{k}{b}(x_j-x_1) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & -i\frac{k}{b} & 0 & 1+i\frac{k}{b}(x_j-x_1) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & -i\frac{k}{b} & 0 & 1+i\frac{k}{b}(x_j-x_1) & 0 \end{bmatrix} \tag{4.46}$$

In the matrix above, each row corresponds to a panel, or to be more precise the control point of a panel. Each column corresponds to the a displacement or rotation of a structural node.

4.3.2 The Integration Matrix

The integration matrix is a fairly simple concept at heart. For some jth panel with a pressure, the integration matrix simply calculates the force and moment generated by the panel on the structural model. Fortunately, having calculated the substantial derivative matrix $[W]$, calculating the integration matrix is simple:

$$[B] = \frac{b}{k} Im([W]^T)[A] \tag{4.47}$$

where $[A]$ is a diagonal square matrix of each of the panels areas. This concludes the doublet lattice method formulation.

4.4 Doublet Lattice Verification

In order to verify the capabilities of the doublet-lattice routine, one of the validation cases from reference [23] is considered. The wing in question is can be seen in figure 4.4, with a chord length of 0.4572 meters and a span of 0.697 meters (this gives the wing an aspect ratio of 3). The wing was discretized using 10 panels in the chordwise direction and 37 panels in the spanwise direction. It should be noted that the lighter gray box on the left hand side of the figure shows that symmetry was considered over the x-z plane.

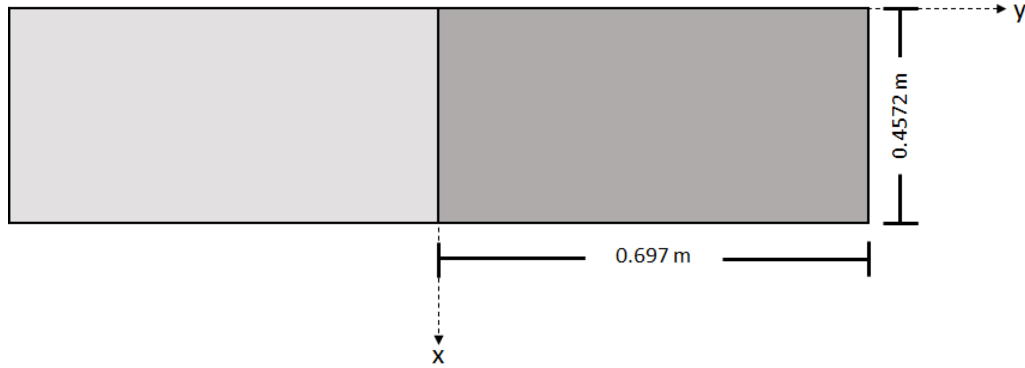


Figure 4.4: The geometry of the verification case

In the original study, the mach number $M = 0.24$ and reduced frequency $k = 0.47$ were considered in this analysis. The imaginary and real parts of the unsteady ΔC_p over the wing can be observed at the non-dimensional span length of $y/s = 0.9$. The non-dimensional bending mode used was [6]:

$$w = 0.18043(y/s) + 1.70255(y/s)^2 - 1.13688(y/s)^3 + 0.25387(y/s)^4 \quad (4.48)$$

The AeroComBAT results can be seen below in 4.5:

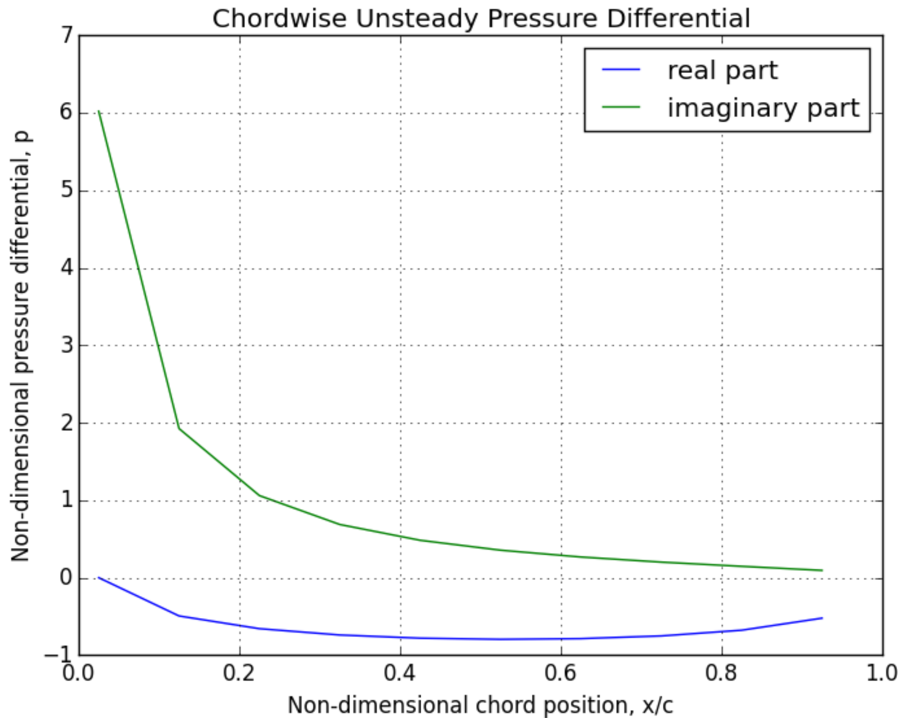


Figure 4.5: Chordwise loading of wing due to first bending mode

These results perfectly match the results achieved by [23] and [17].While this verification is by no means exhaustive, it is assumed that this match is sufficient for an already well established method. This concludes the verification of the implemented doublet lattice method.

4.5 Conclusion

The unsteady aerodynamic model first proposed by Rodden [23] is ideal model for a tool like AeroComBAT. It is detailed enough to capture 3D flow characteristics of planar and non-planar lifting surfaces without being nearly as resource intensive as comparable CFD solutions. One of the few drawbacks of the method is that is does a poor job at modeling the flow at $\omega = 0$. This can be corrected by replacing the steady doublet lattice solution with a vortex lattice solution instead. It also cannot be used to generate the

flow field of a rotor such as a helicopter blade of wind turbine blade. Despite some of its shortcomings, it is an accurate, robust, and efficient tool capable of complimenting the other efficient modeling aspects of AeroComBAT.

Chapter 5

Flutter Solution

5.1 Introduction and Background

THE primary goal when considering the dynamic aeroelastic behavior of a model is to determine the stability of the model. This instability is commonly known as flutter, and while there may be many different types of flutter (most often alluding to the need to use different aerodynamic models), they all generally refer to the same coupling between the aerodynamics and the structural dynamics of a model. Chapter 2 and Chapter 3 were devoted to formulating an accurate and efficient structural dynamic model for a composite beam, and Chapter 4 was devoted to formulating an accurate and efficient unsteady aerodynamic model for the lifting surface. This chapter will be devoted to determining how to most accurately and efficiently evaluate the dynamic stability of a wing model using the previously mentioned structural and aerodynamic formulations.

In flutter analysis there are 3 main types of solutions [13]. There is the p-method, the k-method, and the hybridized pk-method. In the p-method which is accepted as the most accurate of the solutions, the structural dynamic mode shapes are assumed to take the form:

$$\{q\} = \{\bar{q}\}e^{pt} \tag{5.1}$$

Using this assumption, the dynamic equations of motion of the structure can be expressed as:

$$\left(p^2[M] + [K] - \frac{\rho U^2}{2}[Q(p,M)]\right)\{\bar{q}\} = 0 \tag{5.2}$$

where $[M]$ is the generalized mass matrix, $[K]$ is the generalized stiffness matrix, $[Q(p)]$ is the matrix of unsteady aerodynamics influence coefficients and M is the Mach number. Notice that in this case, the aerodynamics of the model are dependent on p. This dependency on p makes capturing the aerodynamic effects difficult and computationally intensive. Equation 5.2 can be converted into an eigenvalue problem where the eigenvalues p are in general complex:

$$p = \omega(\gamma + i) \tag{5.3}$$

where γ is the rate of decay (or the damping in the model) and ω is the frequency. In contrast, the k method is much simpler. By taking the p-method formulation in equation 5.2 but assuming $p = i\omega$, assuming that the modes are always in simple harmonic motion, the equations of motion appear similarly as:

$$\left(-\omega^2[M] + (1 + ig)[K] - \omega^2 \frac{\rho b^2}{2k^2}[Q(ik, M)]\right)\{\bar{q}\} = 0 \tag{5.4}$$

In this case, the eigenvalues of the problem become:

$$\lambda = \frac{1 + ig}{\omega^2} \tag{5.5}$$

where ω is again the frequency at which the mode vibrates. The existence of the term g is somewhat controversial as pointed out by Hodges [13]. What equation 5.4 implies is that it is assumed that in order to maintain the simple harmonic motion assumption of the modes, an artificial structural damping g is introduced. In addition, it also assumes that the unsteady aerodynamics oscillate purely in simple harmonic motion. Despite these large assumptions, the k-method can often accurately predict the flutter velocity of a model. Due to these large assumptions however, it has been shown that the k-method can incorrectly capture the coupling between modes and even show the wrong mode going unstable [10]. Despite these inaccuracies that the k-method is susceptible to, many still use this method due to is extreme efficiency. Finally, a hybridized version of the two methods was proposed called the pk-method in which the the assumption of simple harmonic motion was neglected for the models structural dynamics as originally assumed by equation 5.1, however not for its aerodynamics:

$$\left(p^2[M] + [K] - \frac{\rho U^2}{2}[Q(k, M)]\right)\{\bar{q}\} = 0 \tag{5.6}$$

In this case, the eigenvalues like the p-method are of the form:

$$p = \omega(\gamma + i) \tag{5.7}$$

The pk-method was shown to be more accurate than the k-method, while more efficient than the p-method. This is why the pk-method was selected as the solution method for AeroComBAT. Revisiting equation 5.6, it can be shown that the there is an incongruity between the structural dynamics and the aerodynamics. For a given Mach number and reduced frequency, the AICs can be generated and the eigenvalues of the problem can be solved. For the pk-method, the frequency about which the aerodynamics fluctuate is controlled by the reduced frequency which is just a non-dimensional frequency. In contrast the vibration of the structure is solved for in the eigenvalue solution of equation 5.7. We have no guarantee that when we solve for the eigenvalues, the structural frequency matches the dimensionalized reduced frequency used in generating the AICs. As such, an iterative process is needed for each trial velocity at which the flutter problem is solved. This leads to high inefficiency in the pk-solution and is the main computational expense difference between the k and pk-methods. Using a strategy first proposed by Dale Pitt in 1999 [19], this iterative process can be effectively removed from the problem.

Using a finite element formulation can be fairly computationally expensive since the approach described above would still solve for every eigenvalue and eigenvector possible, rather than the low frequency modes involved in the aeroelastic response. As such, AeroComBAT uses the free-vibration mode shapes of the first few modes to reduce the order of the mass, stiffness, and AIC matrices:

$$[\bar{M}] = [\Phi]^T[M][\Phi] \qquad [\bar{K}] = [\Phi]^T[K][\Phi] \qquad [Q(\bar{k}, M)] = [\Phi]^T[Q(k, M)][\Phi] \tag{5.8}$$

where $[\Phi]$ is a matrix of the free vibration mode shape column vectors.

5.1.1 The Non-Iterative PK-Method

The reduced frequency equation 4.8 can be re-written such that the corresponding aerodynamic circular frequency is:

$$\omega_{aero} = \frac{kU}{b} \tag{5.9}$$

For the traditional pk-method, the work flow is as follows:

1. Calculate all of the [Q(k,M)] AIC matrices for a set of combinations of k and M

2. For every free-stream velocity U

    (a) Assume an initial k value guess and interpolate a new $[Q(k, M)]$ bases on the reference AIC matrices generated in step 1.

    (b) For every flutter mode

        i. Solve the flutter eigenvalue problem

        ii. If the dynamic frequency ω_{root} doesn't correspond to the reduced frequency (in other words equal ω_{aero}), interpolate a new [Q(k,M)]

        iii. Repeat step i and ii until $\omega_{root} = \omega_{aero}$

As it can be seen from the procedure above, having to iterate for every mode until $\omega_{root} = \omega_{aero}$ is very computationally expensive. As such Pitt introduced the non-iterative pk-method. The non-iterative pk-method starts out very similarly. First all of the AIC matricies are generated, and then the system starts iterating over every desired velocity. For a given velocity and k value, the circular frequency is fixed as demonstrated in equation 5.10. Rather than re-interpolating [Q(k,M)] until $\omega_{root} = \omega_{aero}$, all of the p eigenvalues are saved. Then for every mode at a given free-stream velocity, the true ω_{root} of the system is interpolated such that $\omega_{aero} = \omega_{root}$. Having calculated the true ω_{root} at airspeed U, the γ_{root} and mode shapes can also be interpolated as a function of ω_{aero}. The diagram below shows an example plot of ω_{aero} plotted verses all of the different modes ω_{root}.

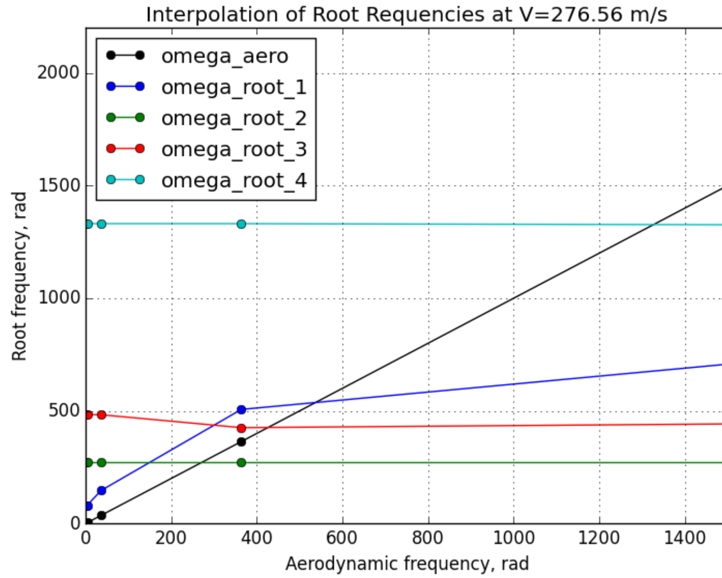


Figure 5.1: The interpolation of the flutter point frequencies at a velocity

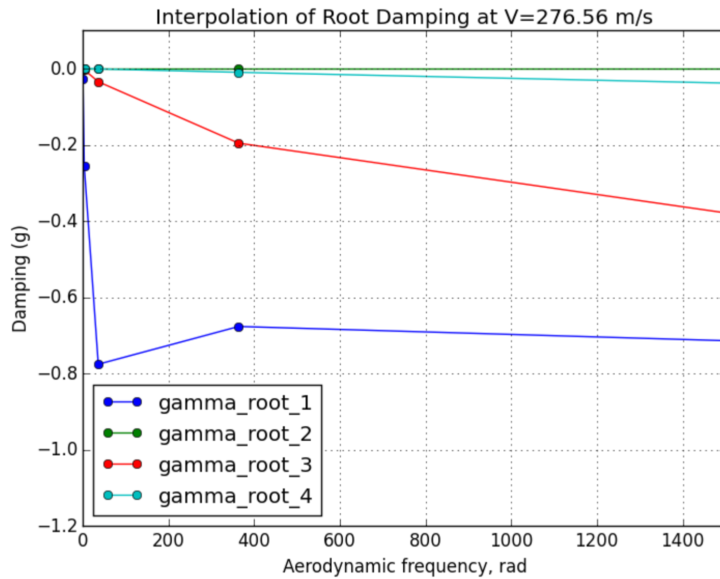


Figure 5.2: The interpolation of the flutter point damping at a velocity

5.1.2 Mode Tracking

In order to ensure that the flutter modes can be accurately tracked, a modal assurance criteria is used as shown below [16]:

$$MAC_{ij} = \frac{\left|\{\bar{q}\}_0^T\{\bar{q}\}_1^*\right|^2}{\left(\{\bar{q}\}_0^T\{\bar{q}\}_0^*\right)\left(\{\bar{q}\}_1^T\{\bar{q}\}_1^*\right)} \tag{5.10}$$

where $\{\bar{q}\}_0$ is the initial reference mode shape, $\{\bar{q}\}_1$ is the perturbed mode shape, and * is the complex conjugate operator. Using equation 5.10, AeroComBAT is capable of tracking mode shapes even when their respective eigenvalues become very close in value.

5.2 PK-Method Verification

In order to verify the accuracy of the pk-method implementation, a cantilevered aluminum wing was considered. The overall dimensions of the wing can be seen in figure 5.3. The light gray region signifies the planform view of the aerodynamic model. On top of that in the darker gray is the planform view of the structural model, with it's reference axis denoted by the red line located at the wing's semichord. In order to verify the accuracy of the results, a NASTRAN model of the beam was generated using plate elements. In the doublet-lattice discretization of the wing for both models, 10 boxes were used in the chordwise direction and 72 boxes were used in the spanwise distribution.

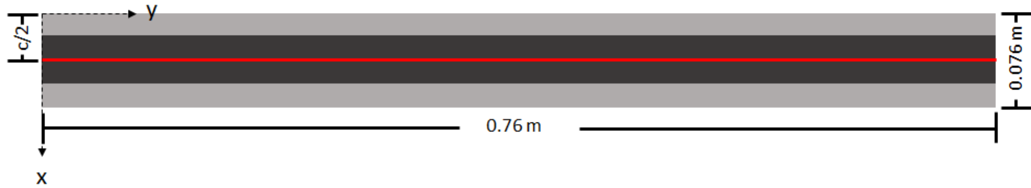


Figure 5.3: Planform view of the wing geometry

The structural model uses a thick-walled box cross-section visible below in figure 5.4. The material used by the structure is an Aluminum with Young's Modulus $E = 68.9$ GPa, Poisson's ratio $\nu = 0.33$, and density

of $\rho = 5400kg/m^3$. Note that the density of Aluminum is normally $\rho = 2700kg/m^3$. In order to encourage a lower flutter speed, the density was increased effectively adding uniformly distributed non-structural mass.

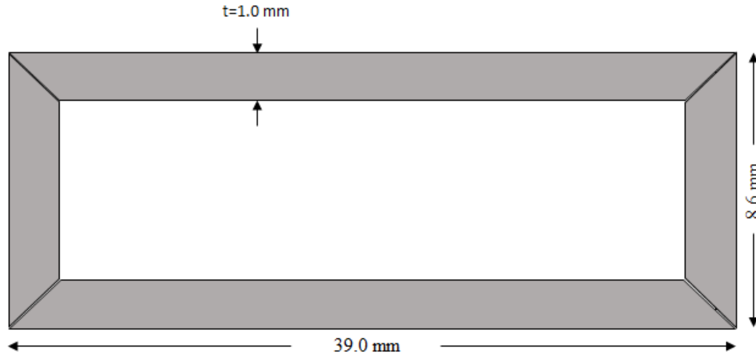


Figure 5.4: Cross-section of the aluminum beam

For this case study, 6 modes are considered. In order to ensure that the structural model of the two codes initially exhibit similar elastic and inertial behavior, the normal mode frequencies are compared, and can be seen below in table 5.1:

Table 5.1: Normal Mode Frequency Comparison

Model	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6
NASTRAN	12.45	43.77	77.26	212.85	269.88	348.05
AeroComBAT	12.43	43.74	77.01	211.87	269.63	351.14

The frequency comparison shows that the structural models compare well. This would imply that the the largest sources of error between the two flutter solutions will be in the differences in the unsteady aerodynamic models themselvs and in how the pk-solution is achieved. For both models, an air density of $\rho_{SL} = 1.225kg/m^3$ was considered, at free-stream airspeeds of $1 \leq U_\infty \leq 342$ meters per second and $M = 0$. The results from the flutter analysis for both codes can be seen below. Note that the solid lines represent the AeroComBAT modes while the dotted lines represent the NASTRAN results:

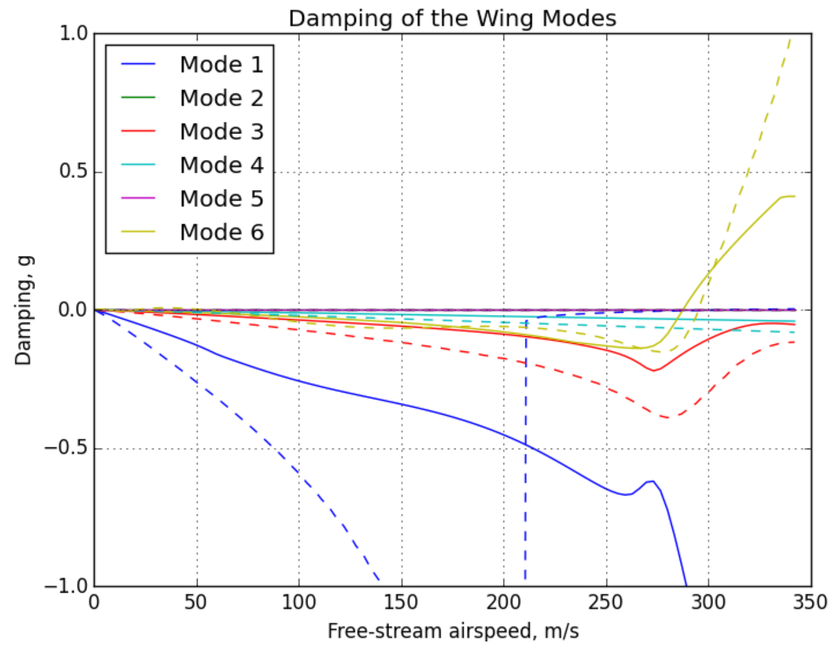


Figure 5.5: Damping of flutter modes vs airspeed

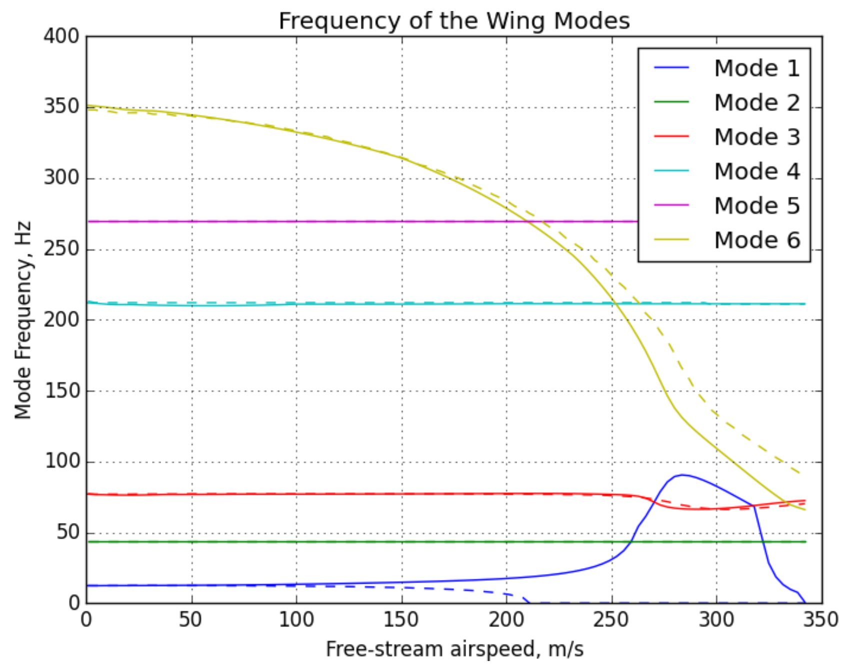


Figure 5.6: Frequency of flutter modes vs airspeed

Comparing the two plots it is clear that there are some non-negligible discrepancies. The most obvious can be seen in figure 5.6. From the NASTRAN results, the first mode (which is the first bending mode) is shown to diverge at $U = 211m/s$. AeroComBAT on the other hand indicates the code doesn't diverge until roughly $U = 342m/s$. Hodges [13] provides a simple closed form solution that can be used to solve for the dynamic divergence pressure assuming 2D strip theory aerodynamics. Fortunately since the aspect ratio of the aerodynamic surface is 10, the strip theory assumption is fairly sound. This equation takes the form:

$$q_d = \frac{GJ}{ecC_{l_\alpha}} \frac{\pi}{2L}^{2} \tag{5.11}$$

Using this equation, the flutter speed for the current example is $273.776m/s$, which is roughly halfway inbetween the two predictions. It is unclear why NASTRAN and AeroComBAT produce such drastically different results, however for this problem the actual divergence speed should be fairly close to that produced by the simple strip-theory analytical model that is equation 5.11. A possible explanation for the inconsistency in the first divergence mode between the analytical strip-theory result and that of AeroComBAT could come from the unsteady aerodynamic model used by the later. The steady aerodynamic contribution of the doublet panels at $\omega = 0$ has been acknowledged as inaccurate. Often when using the doublet lattice method, the steady DLM contribution is subtracted away. What remains is added to the steady solution achieved by using a vortex-lattice method. Since AeroComBAT fails to capture this behavior correctly, this might be an explanation for the resulting behavior.

From the damping plot, it is clear both codes predicted the sixth mode to flutter. The predicted flutter speeds for AeroComBAT and NASTRAN are $U = 287m/s$ and $U = 295m/s$ respectively. This difference in predicted flutter speeds might be explained by the interaction of the modes. Since the modes interact with each other, any large error in the first mode (i.e. failing to correctly model divergence) may propagate to the other modes, changing the predicted flutter speed. Lacking the time to incorporate an additional steady potential flow model such as the vortex-lattice method, the resulting difference in the two models is viewed as acceptable.

5.3 Parametric Composite Study

Having verified the ability of the present work to accurately predict the flutter speed of a wing, two case studies are considered in order to show the effects that composite stiffness tailoring can have on a structures dynamic aeroelastic response. The same wing from Section 5.2 will be reused, only with a slightly different cross-section. The new cross-section can be seen in figure 5.7.

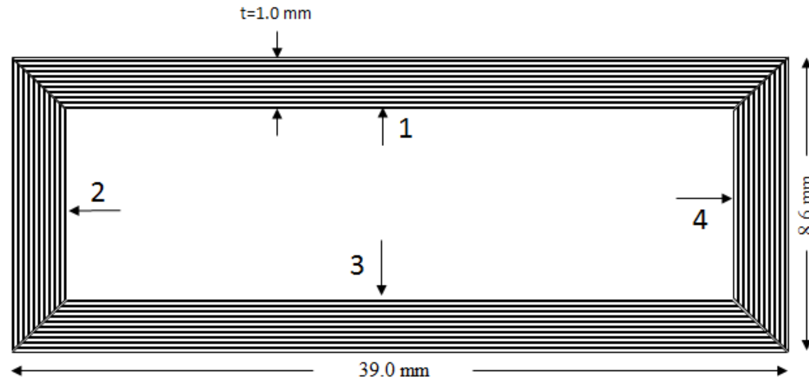


Figure 5.7: Composite cross-section for flutter box

Rather than using solid aluminum, this cross-section will be composed of four laminates. The material to be used is the $AS4/3501-6$ from table 2.10. Two cases will be considered. In the first case, a circumferentially uniform stiffness (CUS) configuration will be considered. In the second case, a circumferentially asymmetric (CAS) configuration will be be used on laminates 1 and 3. These layup schedules can be seen below in table 5.2

Table 5.2: Layup schedules for parametric box beam studies

Case	Material	Laminate 1	Laminate 2	Laminate 3	Laminate 4
Case 1	$AS4/3501-6^*$	$[\theta]_4$	$[\theta]_4$	$[\theta]_4$	$[\theta]_4$
Case 2	$AS4/3501-6^*$	$[\theta]_4$	$[0]_4$	$[-\theta]_4$	$[0]_4$

For case 1, θ was varied from 0 to 90 degrees, and the flutter speeds for each fiber orientation were recorded. The results of case 1 can be seen in figure 5.8:

The flutter boundary curve is agrees fairly well with similar studies done by Patil [18]. The only discrepancy that appears is the discontinuity between

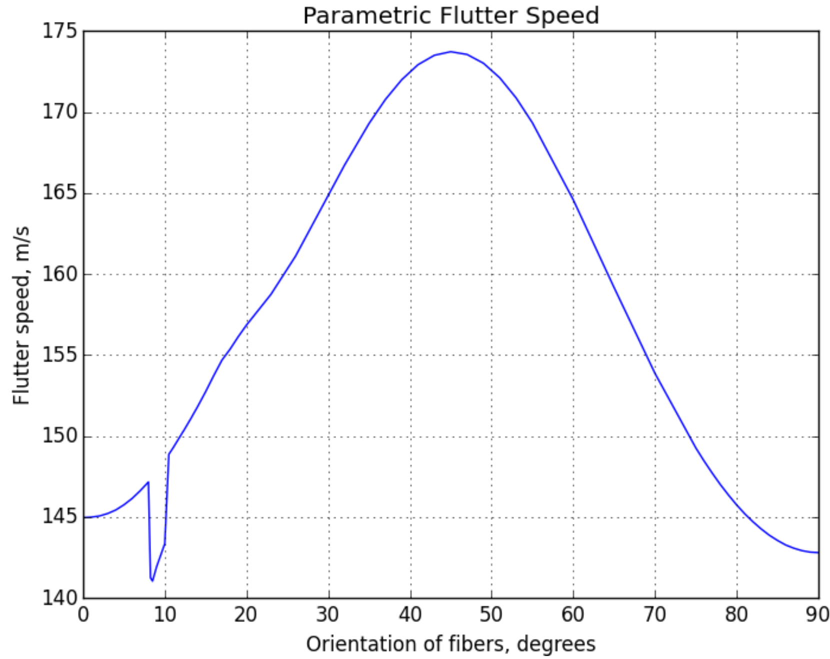


Figure 5.8: Parametric flutter speeds for CUS box beam

$8 \leq \theta \leq 10.5$. The reason for this discontinuity is that a different mode temporarily went unstable at a lower airspeed. That mode is only unstable for a fairly small θ range. Otherwise the entire curve is fairly smooth as expected. While not shown, it can be noted that this behavior is symmetric about the $x = 0$, which is why the flutter speeds from $-90 \leq \theta \leq 0$ are not shown. As θ approaches 45 degrees, two things are occurring. Since fibers are rotated away from the primary bending axis, the bending stiffnesses in both the x and the z directions decreases. For example, at $\theta = 0$, bending stiffness for bending in the z direction is 1.68e2 Pa. At $\theta = 45$, the bending stiffness is 2.39e1 Pa. In contrast, at $\theta = 0$ the torsional stiffness is 2.25e1 and at $\theta = 45$ increases to a value of 4.65e1 Pa. It is likely that the increase in the flutter speed is driven by the increased torsional rigidity. It is likely that the divergence speed would exhibit a similar pattern since for an unswept wing, divergence is purely dictated by torsional effects. Unfortunately as was remarked in Section 5.2, the present work cannot capture static aeroelastic effects.

The flutter results from Case 2 appear slightly more erratic in nature. The results of the flutter speeds can be seen in figure 5.9.

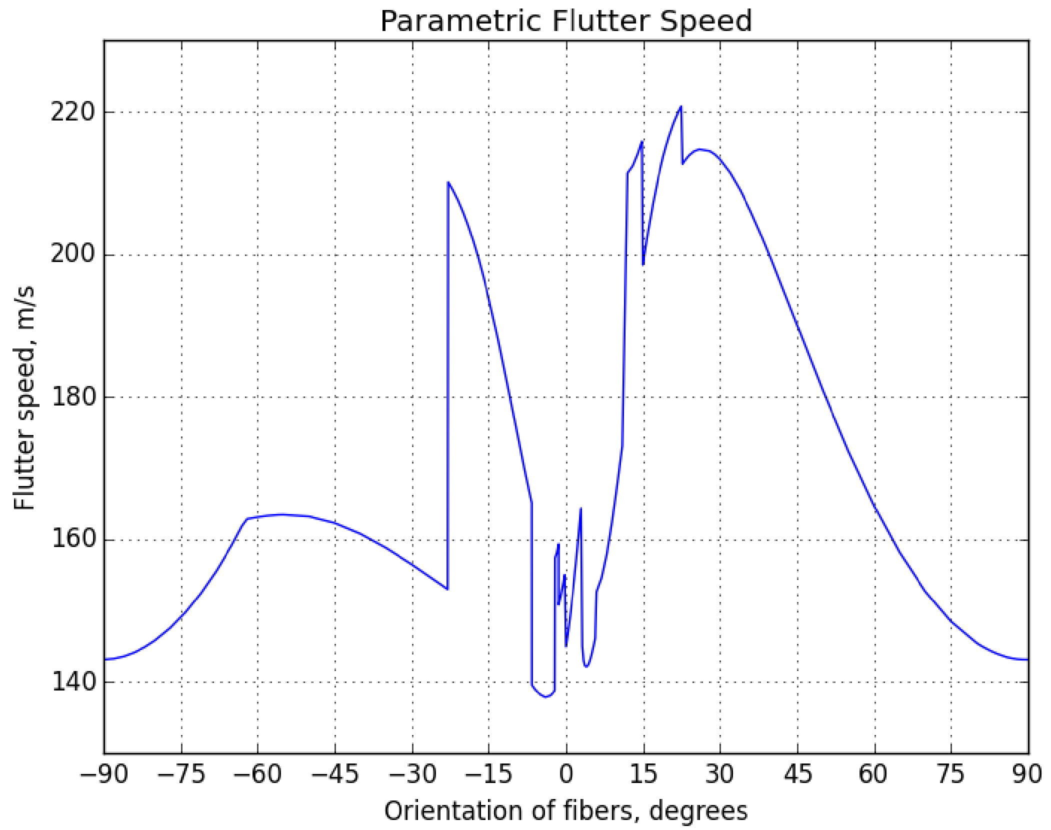


Figure 5.9: Parametric flutter speeds for CAS box beam

The first notable discontinuity appears at $\theta = -23$ degrees. To paint a clearer picture of the flutter behavior at this point, the frequency and damping plots for $\theta = -23$ and $\theta = -22.9$ degrees are included below.

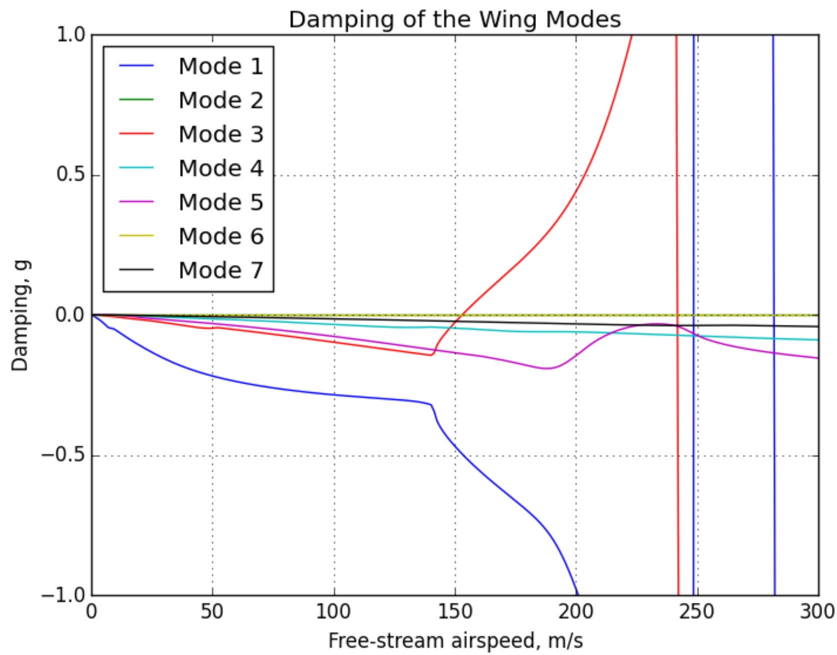


Figure 5.10: Modal damping at $\theta = -23$ degrees

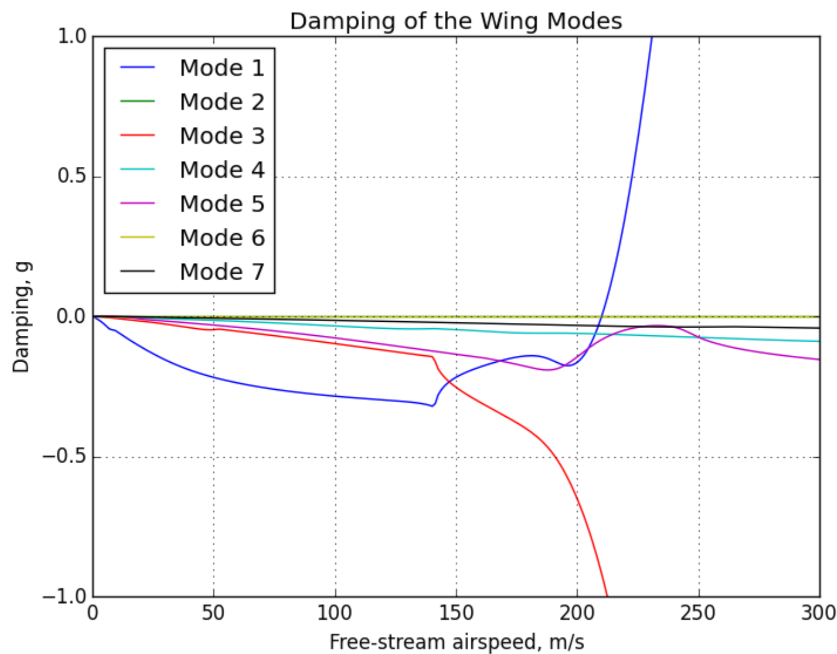


Figure 5.11: Modal damping at $\theta = -22.9$ degrees

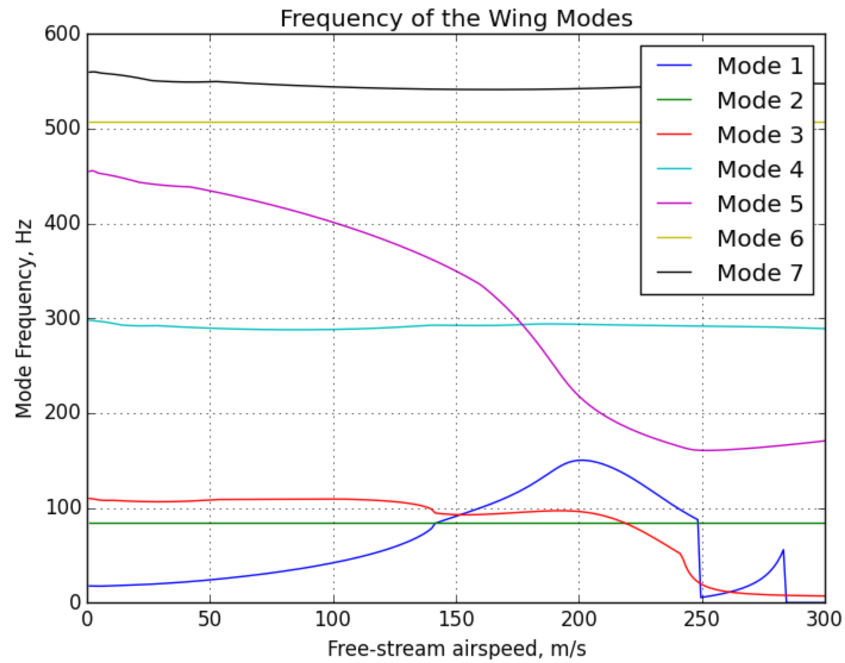


Figure 5.12: Modal damping at $\theta = -23$ degrees

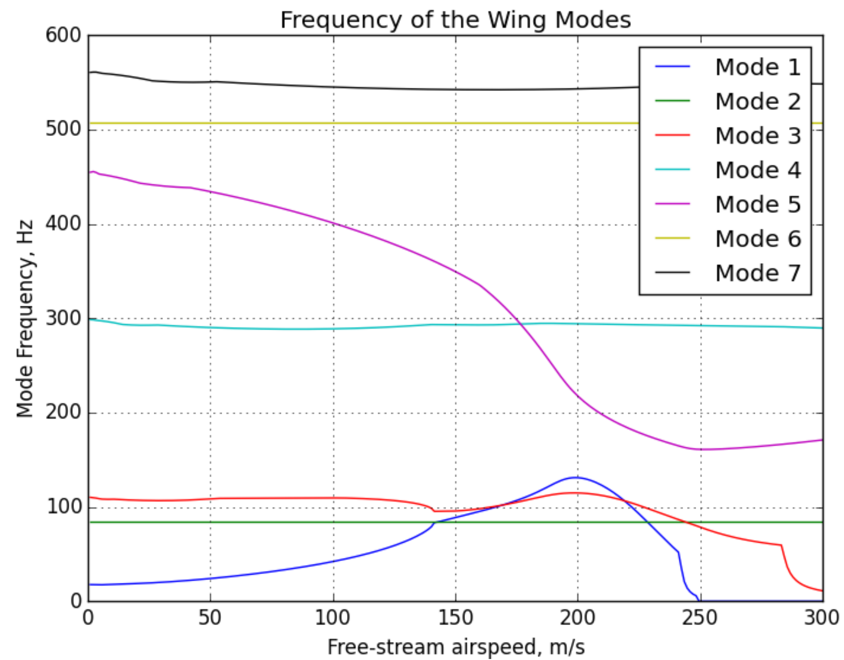


Figure 5.13: Modal damping at $\theta = -22.9$ degrees

It is clear from the damping plots that as θ increases, the red mode stops being unstable and instead the blue mode goes unstable. From the frequency figures, it appears as though as θ increases, the first mode is pulled less towards the fifth mode. As a result, it appears to be able to interact with the third (previously unstable) mode more.

The part of the parametric flutter curve from figure 5.9 has significant discontinuities near $\theta = 0$. Specifically, it occurs between the bounds of $-6.65 \leq \theta \leq 22.75$ degrees. Many additional points were added in this region in order to capture the complex curves as much as possible. Many of these jumps appear because either a mode with a lower flutter speed began to grow unstable, or as in the previous example mode interaction cause discontinuous behavior in the damping. It is quite possible that the non-looping pk-method (NLPK) in AeroComBAT could be exacerbating if not causing some of the numerous discontinuities in this region. Without a comparison model or additional flutter solution approaches however, this is mere speculation.

Chapter 6

Conclusion

6.1 Summary

Composite structures have great potential, and based on their increased use it is clear that industry is already making decisions based on their benefits. Composites have the potential to create complex mechanical behaviors, but taking all of those behaviors into account drastically slows down the early design process. Instead, structural models in the initial design phase are rudimentary, greatly oversimplifying the mechanical behavior of the composite structures. In doing so, any possibility for beneficial stiffness couplings are removed. From Section 5.3, it was shown that even small angle changes in the fiber orientations for cross-sections can lead to significant changes in the dynamic aeroelastic effects, and other works have shown this is true for the static case as well. In addition it is likely that the simplified tools oversimplify the stress-fields and report inaccurate stresses within the laminates.

So how does the present work seek to fix these issues? It does this by intertwining various efficient tools together. At the surface level, the novelty of this work might be obscure. All of the tools utilized in this work were previously developed by other leaders in their respective fields, and no new improvements were added to them individually. Despite these facts, the underlying implications of this work are innovative. It shows that the use of lower fidelity models in concert with one another create a unique tool, allowing engineers to feasibly explore what otherwise would have been too costly to consider. Whether through manual iteration or through the use of

an optimization scheme, accurate information such as dynamic aeroelastic stability and 3D stress fields can be captured for complex composite structures allowing an engineer to quickly and accurately sift through complex composite designs quickly. In addition, there is no other open source tool which shares AeroComBAT's capabilities: cross-sectional analysis, 3D stress recovery, finite element Timoshenko beam analysis, and dynamic aeroelastic stability. AeroComBAT is not complete. There are still many improvements that could be made to the tool and its models. However as it stands, it is a well fleshed out proof-of-concept tool capable of predicting the behavior of composite structures.

6.2 Future Work

There are many improvements that could be made to AeroComBAT to make it more effective and robust. One of the more restrictive features of Giavotto's cross-sectional analysis formulation is that to date there is no way of enforcing warp-restraint into the model. While no boundary condition is completely fixed, many are relatively rigid enough to consider them such, and the results from Section 2.4 and subsection 3.4.1 showed that these effects can have a great effect. As such, adding the capability to consider warp restraint would be valuable.

As useful as it is, the doublet lattice method is still too restrictive. It neglects the thickness effects and is incapable of modeling rotating flow fields such as those perceived by helicopter blades and wind turbines. Furthermore on its own, it lacks the ability to accurately model quasi-steady aerodynamics. It is likely that other similar panel methods could be used to efficiently model steady and unsteady aerodynamics, however it is possible that a similar approach that was taken with the cross-sectional structural analysis could be applied as well.

Another future improvement to AeroComBAT would be a greater focus on stability and nonlinear behavior. As it stands, AeroComBAT has no way of modeling laminate buckling which can be a significant failure mode when using composites. Since AeroComBAT is focused on the early design phase, this could likely be achieved by implementing a Ritz method buckling model. In order to incorporate the nonlinear beam behavior which can be important effects in HALE aircraft, a geometrically exact beam theory could be implemented

Bibliography

[1] Nastran cofe. `https://github.com/vtpasquale/NASTRAN_CoFE`. Accessed: 2015-05-30.

[2] Edward Albano and William P. Rodden. A doublet-lattice method for calculating lift distributions on oscillating surfaces in subsonic flows. *American Institute of Aeronautics and Astronautics*, 7:279–285, 1969.

[3] Raymond L. Bisplinghoff, Holt Ashley, and Robert L. Halfman. *Aeroelasticity*. Dover Publications, Inc., 1996.

[4] Max Blair. A compilation of the mathematics leading to the doublet lattice method. Technical report, Analysis and Optimization Branch, Structures Division, Air Force Wright Laboratory, Wright-Patterson Air Force Base, Ohio, 1992.

[5] Jos Pedro Albergaria Amaral Blasques, Mathias Stolpe, Christian Berggreen, and Kim Branner. *Optimal Design of Laminated Composite Beams*. PhD thesis, 2011.

[6] Enrique Mata Bueso. Unsteady Aerodynamic Vortex Lattice of Moving Aircraft. Master's thesis, Aeronautical and Vehicle engineering department, KTH, Kungliga Tekniska Hogskolan, Stockholm, Sweden, 2011.

[7] Luciano Amaury dos Santos, Roberto Gil Annes da Silva, Breno Moura Castro, Adolfo Gomes Marto, and Antonio Carlos Ponce Alonso. A planar doublet-lattice code for teaching and research in aeroelasticity. *Proceedings of COBEM*, 2005.

[8] Earl Dowell. *A Modern Course in Aeroelasticity*. Springer Publications, Inc., 2015.

[9] V. Giavotto, M. Borri, P. Mantegazza, G. Ghiringghelli, V. Carmaschi, G.C. Maffiolu, and F. Mussi. Anisotropic beam theory and applications. *Composite Structures*, 16:403–413, 1983.

[10] H. J. Hassig. An approximate true damping solution of the flutter equation by determinant iteration. *Journal of Aircraft*, 8:885–889, 1971.

[11] D. H. Hodges, A.R. Atilgan, C. E. S. Cesnik, and M. V. Fulton. On a simplified strain energy function for geometrically non-linear behavior of anisotropic beams. *Composites Engineering*, 2:513–526, 1992.

[12] D. H. Hodges and B. Popescu. On asymptotically correct timoshenko-like anisotropic beam theory. *International Journal of Solids and Structures*, 37:535–558, 2000.

[13] Dewey H. Hodges. *Introduction to Structural Dynamics and Aeroelasticity*. Cambridge University Press, 2002.

[14] Dewey H. Hodges. *Nonlinear Composite Beam Theory*. American Institute of Aeronautics and Astronautics, Inc., 2006.

[15] T. P. Kalman, W. P. Rodden, and J. P. Giesing. Aerodynamic influence coefficients by the doublet lattice method for interfering nonplanar lifting surfaces oscillating in a subsonic flow, part 1. Technical report, Douglas Aircraft Division, McDonnell Douglas Corporation, California, 1969.

[16] Miroslav Pastor, Michal Binda, and Tomas Harcarik. Modal assurance criterion. *Elsevier Ltd. Selection*, 2012.

[17] Mayuresh Patil. Calculation of aerodynamic forces on an oscillating wing for flutter analysis. Technical report, Aerospace Engineering Department, Indian Institute of Technology, Bombay, 1993.

[18] Mayuresh J. Patil. Aeroelastic tailoring of composite box beams. *American Institute of Aeronautics and Astronautics*, 1997.

[19] Dale Pitt. A new non-iterative p-k match point flutter solution. *40th AIAA/ASME/ASCE/AHS/ASC Structuers, Structural Dynamics, and Materials Conference (St. Louis, MO, April 1999)*, 1999.

[20] J. N. Reddy. *An Introduction to the Finite Element Method*. McGraw Hill, 2006.

[21] E. Reissner. On one-dimensional, large-displacement, finite-strain beam theory. *Studies in applied mathematics*, 52:87–95, 1973.

[22] Irving H. Shames and Clive L. Dym. *Energy and Finite Element Methods in Structural Mechanics*. Taylor and Francis Books, Inc., 2003.

[23] B. Stahl, T. P. Kalman, Jp. P. Giesing, and W. P. Rodden. Aerodynamic influence coefficients for oscillating planar lifting surfaces by the doublet lattice method for subsonic flows including quasi-steady fuselage interference. Technical report, Douglas Aircraft Division, McDonnell Douglas Corporation, California, 1968.

[24] Zona Technology, Inc. *Zaero Theoretical Manual*.

AeroComBAT API Documentation

AeroComBAT Documentation

Release 1.0

Ben Names

April 17, 2016

AeroComBAT is a python API for composite, aeroelastic structures. The physics of this package have been heavily verified against codes such as NABSA, VABS, BECAS, and NX NASTRAN.

CHAPTER

ONE

AEROCOMBAT INTRODUCTION

AeroComBAT (Aeroelastic Composite Beam Analysis Tool) is a python API intended to allow users to efficiently models composite beam structures.

Authors Ben Names

Version 1.0 of 2016/04/16

CHAPTER TWO

VERSION 1.0

Capabilities

- Simple classical lamination theory analysis
- Cross-sectional analysis of composite beams
- **3D Composite Timoshenko (shear deformable) beam analysis**
  - Linear static analysis
  - Normal mode analysis
  - Dynamic Aeroelastic Stability (Flutter) Analysis

CHAPTER

THREE

INSTALLATION INSTRUCTIONS

First of all it is strongly recomended that the user first install the Anaconda python distribution from Continuum analytics here.

Ensure that once this is installed, numpy and numba are installed. Finally, in the command prompt or terminal, run:

Distributed load function example:

```
conda install mayavi
```

Mayavi is the 3D visualization engine currently used by AeroComBAT. Note that in some cases installing mayavi has been found to downgrade numpy. This is not necessary, so try to update you numpy installation.

CHAPTER

FOUR

TUTORIALS

For for AeroComBAT Tutorials, see AeroComBAT Tutorials.

CHAPTER

FIVE

TABLE OF CONTENTS:

5.1 AeroComBAT Tutorials

The following are tutorials intended convey how best to use AeroComBAT.

5.1.1 Tutorial 1 - Material Library and Classical Lamination Theory

This tutorial is intended to expose the user to the material library class as well as creating CLT laminates with the laminate class. Note that these laminate objects are integral to meshing the cross-section (XSect) objects.

```
# =============================================================================
# AEROCOMBAT TUTORIAL 1 - MATERIAL lIBRARY AND CLASSICAL LAMINATION THEORY
# =============================================================================

# IMPORT SYSTEM PACKAGES
# ======================
import sys
import os
# Append the root to the system path
sys.path.append(os.path.abspath('..'))

# IMPORT AEROCOMBAT CLASSES
# =========================
from AeroComBAT.Structures import MaterialLib, Laminate
from AeroComBAT.Utilities import RotationHelper
from AeroComBAT.tabulate import tabulate
import numpy as np


# MATERIAL lIBRARY VALIDATION
# ===========================
# Generate Empty Material Library
matlib = MaterialLib()
# Add a graphite orthotropic material
matlib.addMat(1, 'Graphite-Polymer Composite ortho', 'ortho', \
                [155.0e9, 12.1e9, 12.1e9, .458, .248, .248, 3.2e9, 4.4e9,\
                4.4e9, 1.7e3], .15e-3)
# Add a graphite transversely isotropic material
matlib.addMat(2, 'Graphite-Polymer Composite', 'trans_iso', \
                [155.0e9, 12.1e9, .458, .248, 4.4e9, 1.7e3], .15e-3)
# Add a glass transversely isotropic material
matlib.addMat(3, 'Glass-Polymer Composite', 'trans_iso', \
                [50.0e9, 15.2e9, .428, .254, 4.7e9, 1.2e3], .15e-3)
```

```
# Add a T300 transversely isotropic material
matlib.addMat(4, 'T300/5208', 'trans_iso', \
                [181.0e9, 10.3e9, .458, .28, 7.17e9, 1.8e3], .15e-3)
# Add a aluminum isotropic material
matlib.addMat(5, 'AL-2050', 'iso',[75.8, 0.33, 2.7e3], .15e-3)
# Add a rotated T300 transversely isotropic material
matlib.addMat(6, 'T300/5208', 'trans_iso', \
                [181.0e9, 10.3e9, .458, .28, 7.17e9, 1.8e3], .15e-3,th = [0.,45.,0.])
# Print a summary of the mat
matlib.printSummary()
# Get the material associated with MID 1
mat1 = matlib.getMat(1)
# Get the compliance matrix of the material mat1
Smat1 = mat1.Smat
# Get the stiffness matrix of the material mat1 and round for accuracy
Cmat1 = np.around(mat1.Cmat/1e9,decimals=2)
Cmat1[0,0] = np.around(Cmat1[0,0])
# The following matricies are the correct compliance and stiffness matricies
Smat1Test = np.array([[6.45e-12,-1.6e-12,-1.6e-12,0.,0.,0.],\
                      [-1.6e-12,82.6e-12,-37.9e-12,0.,0.,0.],\
                      [-1.6e-12,-37.9e-12,82.6e-12,0.,0.,0.],\
                      [0.,0.,0.,312e-12,0.,0.],\
                      [0.,0.,0.,0.,227e-12,0.],\
                      [0.,0.,0.,0.,0.,227e-12]])
Cmat1Test = np.array([[158e9,5.64e9,5.64e9,0.,0.,0.],\
                      [5.64e9,15.51e9,7.21e9,0.,0.,0.],\
                      [5.64e9,7.21e9,15.51e9,0.,0.,0.],\
                      [0.,0.,0.,3.2e9,0.,0.],\
                      [0.,0.,0.,0.,4.4e9,0.],\
                      [0.,0.,0.,0.,0.,4.4e9]])/1e9
# Check to make sure the calculated values are correct
np.testing.assert_array_almost_equal(Smat1,Smat1Test,decimal=12)
np.testing.assert_array_almost_equal(Cmat1,Cmat1Test,decimal=12)


# MATERIAL PROPERTY ROTATION HELPER VALIDATION
# ============================================
# Create a rotation helper object
rh = RotationHelper()
# Create an array of x-y-z rotations
th = [0.,45.,0.]
# Initialize a stiffness matrix
C = np.array([[1.8403e11,5.4101e9,5.4101e9,0.,0.,0.],\
              [5.4101e9,1.31931e10,6.12866e9,0.,0.,0.],\
              [5.4101e9,6.12866e9,1.31931e10,0.,0.,0.],\
              [0.,0.,0.,5.21455e9,0.,0.],\
              [0.,0.,0.,0.,7.17e9,0.],\
              [0.,0.,0.,0.,0.,7.17e9]])
# Convert it into a compliance matrix
S = np.linalg.inv(C)
# Rotate the compliance matrix
Sp = rh.transformCompl(S,th,xsect=True)
# Convert it back to a stiffness matrix
Cp = np.linalg.inv(Sp)
print('The rotated stiffness matrix:')
print(tabulate(np.around(Cp-C,decimals=3),tablefmt="fancy_grid"))

# ===============================================================================
```

```
# CLT VALIDATION
# ==============================================================================
# Initialize the number of plies per each orientation
n_i = [1,1,1,1]
# Initialize the materials to be used at each orientation
m_i = [4,4,4,4]
# Initialize the angle orientations for the plies
th = [30,-30,0,45]
# Create a laminate with default orientations (for 4 orientations, this will
# default to th_defalt = [0,45,90,-45])
lam1 = Laminate(n_i,m_i,matlib)
# Print a summary of laminate 1
print('Laminate 1 summary:')
lam1.printSummary(decimals=3)
# Create a laminate with default orientations (for more or less than 4
# orientations, th_default = [0]*len(n_i))
lam2 = Laminate(n_i+n_i,m_i+m_i,matlib)
# Print summary of laminate 2
print('Laminate 2 summary:')
lam2.printSummary(decimals=3)
# Create a laminate using the above rotation orientations
lam3 = Laminate(n_i,m_i,matlib,th=th)
# Print Summary of laminate 3
print('Laminate 3 summary:')
lam3.printSummary(decimals=3)
```

5.1.2 Tutorial 2 - CQUADX and Airfoil Classes

This tutorial is intended to expose the user to the CQUADX and airfoil classes.

```
# ==============================================================================
# AEROCOMBAT TUTORIAL 2 - CQUADX AND AIRFOIL
# ==============================================================================

# IMPORT SYSTEM PACKAGES
# ======================
import sys
import os

sys.path.append(os.path.abspath('..'))

# IMPORT AEROCOMBAT CLASSES
# =========================
from AeroComBAT.Structures import Node, MaterialLib, CQUADX
from AeroComBAT.Aerodynamics import Airfoil

# IMPORT NUMPY MODULES
# ====================
import numpy as np
import matplotlib.pyplot as plt

# Material Info
mat_lib = MaterialLib()
# Add an aluminum isotropic material
mat_lib.addMat(1, 'AL-2050', 'iso',[75.8, 0.33, 2.7e3], .15e-3)
```

```
# CQUADX 2D ELEMENT CREATION
# ==========================
# Create a node 1 object
n1 = Node(1,[0.,0.,0.])
# Create a node 2 object
n2 = Node(2,[2.,0.,0.])
# Create a node 3 object
n3 = Node(3,[2.,3.,0.])
# Create a node 4 object
n4 = Node(4,[0.,5.,0.])
# Create a CQUADX element
elem1 = CQUADX(1,[n1,n2,n3,n4],1,mat_lib)
# Print a summary of the element
elem1.printSummary(nodes=True)

# AIRFOIL OUTER MOLD LINE VALIDATION
# ==================================
# Initialize a chord length of 1
c = 1.
# Create an airfoil object with a 'box' profile
af1 = Airfoil(c,name='box')
# Generate a set of non-dimensional x-coordinates
x = np.linspace(-.5,.5,50)
# Create the upper and lower box airfoil curves
xu,yu,xl,yl = af1.points(x)
# Create a matplotlib figure
plt.figure(num=1)
plt.plot(xu,yu)
plt.hold(True)
plt.plot(xl,yl)
plt.axes().set_aspect('equal', 'datalim')
plt.xlabel('x coordinate along the airfoil')
plt.ylabel('y coordinate along the airfoil')
plt.title('Box airfoil profile')
plt.hold(False)

# Create a NACA2412 airfoil profile
af2 = Airfoil(c,name='NACA2412')
# Generate a set of non-dimensional x-coordinates
x = np.linspace(0,1.,500)
# Create the upper and lower airfoil curves
xu,yu,xl,yl = af2.points(x)
# Create a matplotlib figure
plt.figure(num=2)
plt.plot(xu,yu)
plt.hold(True)
plt.plot(xl,yl)
plt.hold(False)
plt.axes().set_aspect('equal', 'datalim')
```

5.1.3 Tutorial 3 - Cross-Section Meshing and Analysis

This is the first extensive tutorial. It exposes the user to meshing several different cross-section types, all with varying complexities.

```
# =============================================================================
# AEROCOMBAT TUTORIAL 3 - Using XSect Objects
# =============================================================================

# IMPORT SYSTEM PACKAGES
# ======================
import sys
import os

sys.path.append(os.path.abspath('..'))

# IMPORT AEROCOMBAT CLASSES
# =========================
from AeroComBAT.Structures import MaterialLib, Laminate, XSect
from AeroComBAT.Aerodynamics import Airfoil

# IMPORT NUMPY MODULES
# ====================
import numpy as np


# ADD MATERIALS TO THE MATERIAL LIBRARY
# =====================================
# Create a material library object
matLib = MaterialLib()
# Add material property from Hodges 1999 Asymptotically correct anisotropic
# beam theory (Imperial)
matLib.addMat(1,'AS43501-6','trans_iso',[20.6e6,1.42e6,.34,.3,.87e6,.1],.005)
# Add material property from Hodges 1999 Asymptotically correct anisotropic
# beam theory (Imperial)
matLib.addMat(2,'AS43501-6*','trans_iso',[20.6e6,1.42e6,.34,.42,.87e6,.1],.005)
# Add an aluminum material (SI)
matLib.addMat(3,'AL','iso',[71.7e9,.33,2810],.005)

# CREATE A LAMINATE CROSS-SECTION
# ===============================
# Create a box airfoil object. Note that when creating an airfoil object, only
# the chord length is used. As such, it doesn't truly matter if the airfoil
# has an airfoil profile or a box profile. In this case we will just give it a
# box profile.
# Initialize the chord length
c1 = 1.
# Initialize the non-dimensional starting and stopping points of the cross-
# section. These bounds when dimensionalized will determine the overall
# dimesions of the cross-section. Therefore the total width of the laminate is:
# xdim[1]*c1-xdim[0]*x. In this case, the total width is 2!
xdim1 = [-1.,1.]
af1 = Airfoil(c1,name='box')
# Create a layup schedule for the laminate. In this case, we will select a
# layup schedule of [0_2/45/90/3]_s
th_1 = [0,45,90]
n_1 = [2,1,3]
m_1 = [1,1,1]
# Notice how the orientations are stored in the 'th_1' array, the subscripts are
# stored in the 'n_1' array, and the material information is held in 'm_1'.
# Create the laminate object:
lam1 = Laminate(n_1,m_1,matLib,th=th_1,sym=True)
# In order to make a cross-section, we must add all of the laminates to be used
```

```python
# to an array:
laminates1 = [lam1]
# We now have all the information necessary to make a laminate beam cross-
# section:
xsect1 = XSect(1,af1,xdim1,laminates1,matLib,typeXSect='laminate',meshSize=2)
# With the cross-section object initialized, let's run the cross-sectional
# analysis to get cross-section stiffnesses, etc.
xsect1.xSectionAnalysis()
# Let's see what our rigid cross-section looks like when plotted in 3D:
xsect1.plotRigid(mesh=True)
# Note that while it might look like the cross-section is made of triangular
# elements, it's actually made of quadrilaterals. This is an artifact of how
# the visualizer mayavi works. Let's get a summary of the cross-section's
# stiffnesses, ect.
xsect1.printSummary(stiffMat=True)
# Notice that from the command line output, all of the important cross-
# sectional geometric properties are located at the origin. By observing the
# cross-section stiffness matrix, it can be seen by the 1,3 entry that there
# is shear-axial coupling. From the non-zero 4,6 entry, we can also tell that
# the cross-section has bending-torsion coupling.
# We can apply a force to the face of this cross-section at the reference axis
# (which in this case is at x,y = 0,0) and see what the stresses look like. In
# this case we'll apply [Fx,Fy,Fz,Mx,My,Mz]=[0.,0.,0.,100.,0.,0.] as if the
# beam belonging to this cross-section were in pure bending.
force1 = np.array([0.,100.,0.,10.,0.,0.])
xsect1.calcWarpEffects(force=force1)
# Having applied the force, let's see what the sigma_zz (normal stresses of the
# beam) look like
xsect1.plotWarped(figName='Laminate Sigma_33 Stress',warpScale=10,\
    contour='sig_33',colorbar=True)
# Let's look at the sigma_13 stress state now since we know there is torsion
# coupling:
xsect1.plotWarped(figName='Laminate Sigma_13 Stress',warpScale=10,\
    contour='sig_13',colorbar=True)
# Notice the increased stress in two of the plies? Recall which ones those are?
# Those plies are the 45 degree plies which are currently taking the shear!

# CREATE A LAMIANTE CROSS-SECTION WITH A DIFFERENT REFERENCE AXIS
# ==============================================================
# The cross-section we just made happened to have all of it's geometrical
# locations (mass center, shear center, tension center) at the origin, which
# is where we applied our force resultant. Suppose we wanted to give the cross-
# section a different reference axis. We can do this by executing the
# xSectionAnalysis method again:
ref_ax = [.5,0.]
# This will move the reference axis (location where we apply forces and
# moments) to x=0.5, y=0. Carrying out the cross-sectional analysis again, we
# get:
xsect1.xSectionAnalysis(ref_ax=ref_ax)
# Let's see how the cross-section's stiffness matrix has changed:
xsect1.printSummary(stiffMat=True)

# Let's not apply the same force resultant and see what the stresses look like:
xsect1.calcWarpEffects(force=force1)
xsect1.plotWarped(figName='Laminate Sigma_33 Stress New Reference Axis',\
    warpScale=10,contour='sig_33',colorbar=True)
xsect1.plotWarped(figName='Laminate Sigma_13 Stress New Reference Axis',\
    warpScale=10,contour='sig_13',colorbar=True)
```

```python
# Notice how the stress resultants are fairly different once we moved the
# reference axis.

# CREATE A WRAPPED RECTANGULAR BOX-BEAM CROSS-SECTION
# ====================================================
# Layup 1 Box beam (0.5 x 0.923 in^2 box with laminate schedule [0]_6)

# Let's make a slightly more complex cross-section. Using the 'rectBox' key
# word and four laminates, we can make a cross-section box-beam. In the next
# case, we are going to make a cross-section from Hodges 1999 Asymptotically
# correct anisotropic beam theory paper. We will do the most simple box-beam
# in this case, which is the "Layup 1" case:
# Establish the chord length
c2 = 0.53
# Establish the non-dimensional starting and stopping points of the cross-
# section.
xdim2 = [-.953/(c2*2),.953/(c2*2)]
# This can be confirmed by plotting, but it should be noted that for the
# 'rectBox' routine, the mesh y-coordinates will go from -c2/2 -> c2/2, and the
# x-coordinates will go from xdim2[0]*c -> xdim2[1]*c. Therefore the box's
# overall dimesions should be from 0.53 in x 0.953 in. Next we will generate
# the airfoil box:
af2 = Airfoil(c2,name='box')
# Now let's make all of the laminate objects we will need for the box beam. In
# this case it's 4:
n_i_Lay1 = [6]
m_i_Lay1 = [2]
th_Lay1 = [0.]
lam1_Lay1 = Laminate(n_i_Lay1, m_i_Lay1, matLib, th=th_Lay1)
lam2_Lay1 = Laminate(n_i_Lay1, m_i_Lay1, matLib, th=th_Lay1)
lam3_Lay1 = Laminate(n_i_Lay1, m_i_Lay1, matLib, th=th_Lay1)
lam4_Lay1 = Laminate(n_i_Lay1, m_i_Lay1, matLib, th=th_Lay1)
# Assemble the laminates into an array. Refer back to the documentation to
# remind yourself of stacking sequence direction, etc. It should be noted that
# the first laminate is at the top of the box cross-section. The next is the
# left laminate, and so on in a counter-clockwise direction.
laminates_Lay1 = [lam1_Lay1,lam2_Lay1,lam3_Lay1,lam4_Lay1]
# Create the cross-section vector:
xsect_Lay1 = XSect(2,af2,xdim2,laminates_Lay1,matLib,typeXSect='rectBox',\
    meshSize=2)
# Create the cross-section object. Note that since we aren't specifying the
# reference axis, it will automatically be set at the shear center. Since this
# cross-section is simple, this will still be at the origin.
xsect_Lay1.xSectionAnalysis()
# Let's look at the stiffness matrix:
xsect_Lay1.printSummary(stiffMat=True)
# Since this cross-section is simple, we can analytically calculate some of the
# simpler parameters. For example, the 3,3 entry is just E1*A. Similarly, the
# bending stiffnesses of the cross-section are just E1*I_xx and E1*I_yy. Try
# calculating them on your own to verify this! Note that this will only work
# since all of the fibers have a 0 degree orientation. Let's apply a load and
# see how it behaves!
# Force Resultant Vector:
force2 = np.array([0.,0.,0.,0.,0.,100.])
# Calculate the effects of the force resultant
xsect_Lay1.calcWarpEffects(force=force2)
# Plot the normal stress
xsect_Lay1.plotWarped(figName='Layup 1 Box Beam Sigma_33 Stress',\
```

```
    warpScale=100,contour='sig_33',colorbar=True)
# Now the shear 13 stress!
xsect_Lay1.plotWarped(figName='Layup 1 Box Beam Sigma_13 Stress',\
    warpScale=100,contour='sig_13',colorbar=True)

# Look at the differences in magnitudes of the stress between the two plots.
# Notice anything? There is virtually no normal stress, but A LOT of shear
# stress. This makes sense though since we only applied a torque to the cross-
# section. Note that this warping profile is vero common for any box type cross
# section. Let's try one more slightly more complex shape:

# NACA 2412 BOX BEAM
# ==================

# Now let's mesh a NACA2412 box beam. We will use the last of the supported
# meshing routines for this. This is the less restrictive than the 'rectBox'
# routine, and has different laminate mesh interfaces. This time we will also
# make a slightly more interesting mesh using unbalanced and unsymetric
# laminates. First let's initialize the airfoil shape:
# Initialize a chord length of four inches
c3 = 4.
# Initialize the non-dimesional locations for the airfoil points to be
# generated:
xdim3 = [.15,.7]
# Create the airfoil object:
af3 = Airfoil(c3,name='NACA2412')
# Create the laminates to make up the cross-section
n_i_1 = [1,1,1,1,1,1]
m_i_1 = [2,2,2,2,2,2]
th_1 = [-15,-15,-15,-15,-15,-15]
lam1 = Laminate(n_i_1, m_i_1, matLib, th=th_1)
n_i_2 = [1,1,1,1,1,1]
m_i_2 = [2,2,2,2,2,2]
th_2 = [15,-15,15,-15,15,-15]
lam2 = Laminate(n_i_2, m_i_2, matLib, th=th_2)
n_i_3 = [1,1,1,1,1,1]
m_i_3 = [2,2,2,2,2,2]
th_3 = [15,15,15,15,15,15]
lam3 = Laminate(n_i_3, m_i_3, matLib, th=th_3)
n_i_4 = [1,1,1,1,1,1]
m_i_4 = [2,2,2,2,2,2]
th_4 = [-15,15,-15,15,-15,15]
lam4 = Laminate(n_i_4, m_i_4, matLib, th=th_4)
# Organize the laminates into an array
laminates_Lay3 = [lam1,lam2,lam3,lam4]
# Create the cross-section object and mesh it
xsect_Lay3 = XSect(4,af3,xdim3,laminates_Lay3,matLib,typeXSect='box',meshSize=2)
# Run the cross-sectional analysis. Since this is an airfoil and for this,
# symmetric airfoils the AC is at the 1/c chord, we will put the reference axis
# here
xsect_Lay3.xSectionAnalysis(ref_ax=[0.25*c3,0.])
# Let's see what the rigid cross-section looks like:
xsect_Lay3.plotRigid()
# Print the stiffness matrix
xsect_Lay3.printSummary(stiffMat=True)
# Create an applied force vector. For a wing shape such as this, let's apply a
# semi-realistic set of loads:
force3 = np.array([10.,100.,0.,10.,1.,0.])
```

```
# Calculate the force resultant effects
xsect_Lay3.calcWarpEffects(force=force3)
# This time let's plot the max principle stress:
xsect_Lay3.plotWarped(figName='NACA2412 Box Beam Max Principle Stress',\
    warpScale=10,contour='MaxPrin',colorbar=True)
```

5.1.4 Tutorial 4 - Using Superbeams to Conduct Simple Beam Analysis

This tutorial exposes users to conducting beam analysis using the SuperBeam class. It also exposes users to the Model class for the first time.

```
# =============================================================================
# AEROCOMBAT TUTORIAL 4 - Using Super Beams to Conduct Analysis
# =============================================================================

# IMPORT SYSTEM PACKAGES
# ======================
import sys
import os

sys.path.append(os.path.abspath('..'))

# IMPORT AEROCOMBAT CLASSES
# =========================
from AeroComBAT.Structures import MaterialLib, Laminate, XSect, SuperBeam
from AeroComBAT.Aerodynamics import Airfoil
from AeroComBAT.FEM import Model

# IMPORT NUMPY MODULES
# ====================
import numpy as np
import mayavi.mlab as mlab

# ADD MATERIALS TO THE MATERIAL LIBRARY
# =====================================
# Create a material library object
matLib = MaterialLib()
# Add material property from Hodges 1999 Asymptotically correct anisotropic
# beam theory (Imperial)
matLib.addMat(1,'AS43501-6','trans_iso',[20.6e6,1.42e6,.34,.3,.87e6,.1],.005)
# Add material property from Hodges 1999 Asymptotically correct anisotropic
# beam theory (Imperial)
matLib.addMat(2,'AS43501-6*','trans_iso',[20.6e6,1.42e6,.34,.42,.87e6,.1],.005)
# Add an aluminum material (SI)
matLib.addMat(3,'AL','iso',[71.7e9,.33,2810],.005)

# CREATE A WRAPPED RECTANGULAR BOX-BEAM CROSS-SECTION
# ===================================================
# Layup 1 Box beam (0.5 x 0.923 in^2 box with laminate schedule [0]_6)

# Before we make a beam, we must first make the cross-section of that beam. We
# are going to start with a cross-section we used in the third tutorial.
c2 = 0.53
# Establish the non-dimensional starting and stopping points of the cross-
# section.
xdim2 = [-.953/(c2*2),.953/(c2*2)]
# Generate the airfoil box:
```

```python
af2 = Airfoil(c2,name='box')
# Now let's make all of the laminate objects we will need for the box beam. In
# this case it's 4:
n_i_Lay1 = [6]
m_i_Lay1 = [2]
th_Lay1 = [0.]
lam1_Lay1 = Laminate(n_i_Lay1, m_i_Lay1, matLib, th=th_Lay1)
lam2_Lay1 = Laminate(n_i_Lay1, m_i_Lay1, matLib, th=th_Lay1)
lam3_Lay1 = Laminate(n_i_Lay1, m_i_Lay1, matLib, th=th_Lay1)
lam4_Lay1 = Laminate(n_i_Lay1, m_i_Lay1, matLib, th=th_Lay1)
# Assemble the laminates into an array.
laminates_Lay1 = [lam1_Lay1,lam2_Lay1,lam3_Lay1,lam4_Lay1]
# Create the cross-section vector:
xsect_Lay1 = XSect(2,af2,xdim2,laminates_Lay1,matLib,typeXSect='rectBox',\
    meshSize=2)
# Create the cross-section object.
xsect_Lay1.xSectionAnalysis()
# Having created the cross-section, we can now generate a superbeam. A
# superbeam is just a collection of beam elements. In other words, a superbeam
# is just there to fascilitate beam meshing and other pre/post processing
# benefits. In order to make a superbeam, we need to initialize a few things.
# First, let's initialize the starting and stopping location of the beam:
x1 = np.array([0,0,0])
x2 = np.array([0,0,4])
# Initialize a superbeam ID
SBID = 1
# Next we need to initialize the number of elements the superbeam should mesh:
noe = 20
# Now let's make the superbeam
sbeam1 = SuperBeam(SBID,x1,x2,xsect_Lay1,noe)
# In order to analyze this beam, we'll need to add it to a finite element
# model. First let's make a finite element model!
model = Model()
# Easy right? Now let's add the superbeam to the model.
model.addElements([sbeam1])
# Now that our beam is loaded into the FEM, let's visualize it!
model.plotRigidModel(numXSects=8)
# First let's constrain the beam at it's root. Since we only added one
# superbeam and we never specified a starting node ID, we know that the first
# node ID is actually 1! So when we constrain the model, we can just select:
model.applyConstraints(1,'fix')
# There are two supported keywords for constraints, either 'fix' or 'pinned'.
# If the user wanted to apply a different constraint, they can just enter the
# degrees of freedom to constrain on the model. This can be done by supplying
# an array such as: [1,2,3,5]. Now let's apply a load. We will make two load
# cases. In the first case, we are going to apply a simple tip load:
load1 = {21:np.array([100.,100.,0.,0.,0.,100.])}
# We can also create a function for a distributed load:
def load2(x):
   vx = (1/10)*10*x[2]**2-7*x[2]-2.1
   vy = 10*x[2]**2-7*x[2]
   pz = 0
   mx = 0
   my = 0
   tz = (10*x[2]**2-7*x[2])/10+3*x[0]**2
   return np.array([vx,vy,pz,mx,my,tz])
# Ok now let's add these loads to the model:
model.applyLoads(1,F=load1)
```

```
# Notice that when I applied a tip load, I did it using the argument 'F'. When
# we apply a distributed load function, we use the argument 'f' instead.
model.applyLoads(2,f=load2,allElems=True)
# Now with constraints and loads, we can run a static analysis! Let's run the
# first load case.
model.staticAnalysis(1,analysis_name='tip load')
# Let's see what results we get:
model.plotDeformedModel(analysis_name='tip load',figName='Tip Load Analysis',\
    numXSects=8,contour='VonMis',contLim=[0,1e5],warpScale=50,displScale=10)
# Now let's try analyzing the distributed load:
model.staticAnalysis(2,analysis_name='distributed load')
# Let's see what results we get for the distributed load:
model.plotDeformedModel(analysis_name='distributed load',\
    figName='Distributed Load Analysis',numXSects=8,contour='VonMis',\
    contLim=[0,1e5],warpScale=50,displScale=10)
# Really quickly, let's discuss some of the keywords used in the analysis and
# plotting. The analysis_name designates the name where the results should be
# stored for the beam. Therefore if you want to keep multiple results stored
# at once you can give it a name. Otherwise this will always result to the
# default name. figName is just the name of the MayaVi figure, and numXSects
# designates how many cross-sections should be plotted. Note that the more
# cross-sections are plotted, the slower the plotting process will be. Both
# contour, contLim and warpScale were discussed in Tutorial 3. displScale is
# the scalling factor applied to the beam displacements and rotations.
# We can also run a normal modes analysis on the beam as well: having already
# applied the constraints, we can run:
model.normalModesAnalysis(analysis_name='normal modes')
# For now, the frequencies can be accessed via the model attribute 'freqs'
Frequencies = model.freqs
# Let's plot the first mode:
model.plotDeformedModel(analysis_name='normal modes',\
    figName='Normal Modes 1',numXSects=10,contour='none',\
    warpScale=1,displScale=2,mode=1)
# How about the second?
model.plotDeformedModel(analysis_name='normal modes',\
    figName='Normal Modes 2',numXSects=10,contour='none',\
    warpScale=1,displScale=5,mode=2)
# How about the third? It happens to be the torsional mode.
model.plotDeformedModel(analysis_name='normal modes',\
    figName='Normal Modes 3',numXSects=10,contour='none',\
    warpScale=1,displScale=1,mode=3)
```

5.1.5 Tutorial 5 - Creating a Wing and Conducting a Flutter Analysis

This tutorial exposes users to creating a wing and conducting flutter analysis on it. Note that every analysis a superbeam can conduct can also be conducted on a wing.

```
# =============================================================================
# AEROCOMBAT TUTORIAL 4 - Using Super Beams to Conduct Analysis
# =============================================================================

# IMPORT SYSTEM PACKAGES
# ======================
import sys
import os
sys.path.append(os.path.abspath('..'))
```

```
# IMPORT NUMPY PACKAGES
# =====================
import numpy as np
import matplotlib.pyplot as plt

# IMPORT AEROCOMBAT CLASSES
# =========================
from AeroComBAT.Structures import MaterialLib
from AeroComBAT.AircraftParts import Wing
from AeroComBAT.FEM import Model

# ADD MATERIALS TO THE MATERIAL LIBRARY
# =====================================
# Create a material library object
matLib = MaterialLib()
# Add an aluminum material (SI)
matLib.addMat(1,'AL','iso',[68.9e9,.33,2700*2],.00025)
# Add an soft material material (SI)
matLib.addMat(2,'Weak_mat','iso',[100,.33,10],.00025)
# Add material property from Hodges 1999 Asymptotically correct anisotropic
# beam theory (SI)
matLib.addMat(3,'AS43501-6*','trans_iso',[142e9,9.8e9,.34,.42,6e9,20000],0.0005)

# CREATE THE WING
# ===============
# Define the chord length of the model
c = .076
# Define the chord length (this will be the hight of the box beam)
ctip = 0.0076+.001
# Since the wing isn't tapered
croot = 0.0076+.001
# Define the non-dimensional starting point of the cross-section
x1 = -0.039/croot/2
# Define the non-dimensional ending point of the cross-section
x2 = 0.039/croot/2
# Define the span of the beam
span = 0.76
# Define the starting and stopping point of the wing structure
p1 = np.array([c/2,0.,0.])
p2 = np.array([c/2,span,0.])
# Define the non-dimesional locations of the ribs in the wing.
Y_rib = np.linspace(0.,1.,2)

# Initilize the layup schedule for the cross-section.
n_ply = [4,4,4,4]
# Initilize the material ID corresponding to an orientation in the
# cross-section
m_ply = [1,1,1,1]
# Initilize the orientations used in the box beam.
th_ply = [0,0,0,0]

# Define the number of orientations used in each laminate. In this case, we'll
# just use one.
n_orients = 1
# Define the number of laminates per cross-section. In this case since we are
# using a box beam, it will be 4.
n_lams = 4
# Define the type of cross-section. In this case it'll be the 'rectBox' mesh
```

```
typeXSect = 'rectBox'
# Define the number of elements per unit length are to be used. The structure
# will have then 120*span beam elements.
noe_dens = 120
# Define the aditional vector required to define the orientation of the beam.
# In this case, we'll have it pointing down the length of the wing.
chordVec=np.array([1.,0.,0.])
# Create the wing object. For more information about some of the input
# parameters see the AeroComBAT documentation
wing1 = Wing(1,p1,p2,croot,ctip,x1,x2,Y_rib,n_ply,m_ply,matLib,name='box',\
    noe=noe_dens,chordVec=chordVec,ref_ax='shearCntr',n_orients=n_orients,\
    n_lams=n_lams,typeXSect=typeXSect,meshSize=2,th_ply=th_ply)
# This is an optional step for ease of programming. Since only one wing section
# was created and the wing isn't tapered, there is only one superbeam which
# contains all of the beam elements in the wing.
sbeam1 = wing1.wingSects[0].SuperBeams[0]

# ADD A LIFTING SURFCE TO THE WING
# ================================
# Define the root leading edge location
x1 = np.array([0,0.,0.])
# Define the root trailing edge location
x2 = np.array([c,0.,0.])
# Define the tip trailing edge location
x3 = np.array([c,span,0.])
# Define the tip leading edge location
x4 = np.array([0,span,0.])
# Determine the number of boxes to be used in the spanwise direction
nspan = 36*2
# Determine the number of boxes to be used in the chordwise direction
nchord = 10
# Add the lifting surface to the wing.
wing1.addLiftingSurface(1,x1,x2,x3,x4,nspan,nchord)

# MAKE THE FINITE ELEMENT MODEL (FEM)
# ===================================
model  = Model()
# Add the aircraft wing to the model
model.addAircraftParts([wing1])
# Apply the constraint for the model
model.applyConstraints(0,'fix')
# Plot the rigid wing in the finite elment model
model.plotRigidModel(numXSects=10)
# Conduct a normal modes analysis. Since the normal mode shapes are used in the
# flutter analysis it is good to run this ahead of time to make sure you are
# selecting enough mode shapes to include any relevant torsional and bending
# mode shapes.
model.normalModesAnalysis()
# Save the frequencies of the modal analysis
freqs = model.freqs

# IMPORT NASTRAN RESULTS
# ======================
# The same analysis was conducted on a plate model in NX NASTRAN to verify the
# results produced by AeroComBAT
NASTRAN = np.genfromtxt('NASTRANFlutterResults.csv', delimiter=',')
UNAST = NASTRAN[:,0]
Damp1 = NASTRAN[:,1]
```

```
Freq1 = NASTRAN[:,2]
Damp2 = NASTRAN[:,3]
Freq2 = NASTRAN[:,4]
Damp3 = NASTRAN[:,5]
Freq3 = NASTRAN[:,6]
Damp4 = NASTRAN[:,7]
Freq4 = NASTRAN[:,8]
Damp5 = NASTRAN[:,9]
Freq5 = NASTRAN[:,10]
Damp6 = NASTRAN[:,11]
Freq6 = NASTRAN[:,12]

# CONDUCT FLUTTER ANALYSIS
# ========================
# Whenever a flutter analysis is conducted, several quantities need to be
# defined. The first is an array of free-stream airspeeds:
U_vec = np.linspace(1,342,100)
# In addition, a vector of trial reduced frequencies must be initialized. Keep
# in mind that the because the non-looping pk method is used, a wide range of
# reduced frequencies may need to be used.
kr_vec = np.array([0.,1e-06,1e-04,.001,.01,.05,.1,.5,1.,5.,10.,50])*10
# A vector of mach numbers must also be used. These should be kept close to
# The suspected flutter mach number. If mach numbers greater than 0.7 are used,
# is is likely doublet lattice method is no-longer valid.
M_vec = [0.]*len(kr_vec)
# Initialize the sea level density
rho_0 = 1.225
# Determine the number of modes to be used
nmodes = 6
# Run the flutter analysis. Depending on how many reduced frequencies and
# velocities sampled, this could take anywhere from a 30 seconds to 2 minutes.
model.flutterAnalysis(U_vec,kr_vec,M_vec,c,rho_0,nmodes,symxz=True,g=0.0)

# POST-PROCESS THE FLUTTER RESULTS
# ================================
# Note that in these figures, the dashed lines are the NASTRAN results, whereas
# the solid lines are the AeroComBAT results.
cvec = ['b','g','r','c','m','y']
plt.figure(1)
plt.hold(True)
for PID, point in model.flutterPoints.iteritems():
    plt.plot(U_vec,point.gamma,color=cvec[PID],label='Mode '+str(PID+1))
plt.legend(loc=2)
plt.ylim([-1,1])
plt.plot(UNAST,Damp1,str(cvec[0])+'--',label='NASTRAN Mode 1')
plt.plot(UNAST,Damp2,str(cvec[1])+'--',label='NASTRAN Mode 2')
plt.plot(UNAST,Damp3,str(cvec[2])+'--',label='NASTRAN Mode 3')
plt.plot(UNAST,Damp4,str(cvec[3])+'--',label='NASTRAN Mode 4')
plt.plot(UNAST,Damp5,str(cvec[4])+'--',label='NASTRAN Mode 5')
plt.plot(UNAST,Damp6,str(cvec[5])+'--',label='NASTRAN Mode 6')
plt.title('Damping of the Wing Modes')
plt.xlabel('Free-stream airspeed, m/s')
plt.ylabel('Damping, g')
plt.grid(True)
plt.hold(False)

plt.figure(2)
plt.hold(True)
```

```
for PID, point in model.flutterPoints.iteritems():
    plt.plot(U_vec,point.omega,color = cvec[PID],label='Mode '+str(PID+1))
plt.legend(loc=1)
plt.plot(UNAST,Freq1,str(cvec[0])+'--',label='NASTRAN Mode 1')
plt.plot(UNAST,Freq2,str(cvec[1])+'--',label='NASTRAN Mode 2')
plt.plot(UNAST,Freq3,str(cvec[2])+'--',label='NASTRAN Mode 3')
plt.plot(UNAST,Freq4,str(cvec[3])+'--',label='NASTRAN Mode 4')
plt.plot(UNAST,Freq5,str(cvec[4])+'--',label='NASTRAN Mode 5')
plt.plot(UNAST,Freq6,str(cvec[5])+'--',label='NASTRAN Mode 6')
plt.title('Frequency of the Wing Modes')
plt.xlabel('Free-stream airspeed, m/s')
plt.ylabel('Mode Frequency, Hz')
plt.grid(True)
plt.hold(False)

# The following figures demonstrate how the damping and frequencies are
# interpolated.
Uind = 80
point1 = model.flutterPoints[0]
point2 = model.flutterPoints[1]
point3 = model.flutterPoints[2]
point4 = model.flutterPoints[3]
omegaAeros = point1.omegaAeroDict[U_vec[Uind]]
omegaRoots1 = point1.omegaRootDict[U_vec[Uind]]
omegaRoots2 = point2.omegaRootDict[U_vec[Uind]]
omegaRoots3 = point3.omegaRootDict[U_vec[Uind]]
omegaRoots4 = point4.omegaRootDict[U_vec[Uind]]
gammas1 = point1.gammaDict[U_vec[Uind]]
gammas2 = point2.gammaDict[U_vec[Uind]]
gammas3 = point3.gammaDict[U_vec[Uind]]
gammas4 = point4.gammaDict[U_vec[Uind]]

plt.figure(3)
plt.hold(True)
plt.plot(omegaAeros,omegaAeros,str(cvec[0])+'o-',label='omega_aero')
plt.plot(omegaAeros,omegaRoots1,str(cvec[1])+'o-',label='omega_root_1')
plt.plot(omegaAeros,omegaRoots2,str(cvec[2])+'o-',label='omega_root_2')
plt.plot(omegaAeros,omegaRoots3,str(cvec[3])+'o-',label='omega_root_3')
plt.plot(omegaAeros,omegaRoots4,str(cvec[4])+'o-',label='omega_root_4')
plt.legend(loc=2)
plt.ylim([0,20000])
plt.xlim([0,25000])
plt.xlabel('Aerodynamic frequency, rad')
plt.ylabel('Root frequency, rad')
plt.title('Interpolation of Root Requencies at V=%4.2f m/s'%(U_vec[Uind]))
plt.grid(True)
plt.hold(False)

plt.figure(4)
plt.hold(True)
plt.plot(omegaAeros,gammas1,str(cvec[1])+'o-',label='gamma_root_1')
plt.plot(omegaAeros,gammas2,str(cvec[2])+'o-',label='gamma_root_2')
plt.plot(omegaAeros,gammas3,str(cvec[3])+'o-',label='gamma_root_3')
plt.plot(omegaAeros,gammas4,str(cvec[4])+'o-',label='gamma_root_4')
plt.legend(loc=3)
plt.ylim([-1,.1])
plt.xlim([0,25000])
plt.xlabel('Aerodynamic frequency, rad')
```

```
plt.ylabel('Damping (g)')
plt.title('Interpolation of Root Damping at V=%4.2f m/s'%(U_vec[Uind]))
plt.grid(True)
plt.hold(False)
```

5.2 FEM Interface Module

This module contains a basic environment for conducting finite element analysis.

The primary purpose of this library is to fascilitate the creation of a FEM within the AeroComBAT package.

SUMARRY OF THE CLASSES

- ***Model*: The Model class has two main purposes. The first is that it is meant** to serve as an organizational class. Once an aircraft part has been loaded into the model by using the addAircraftPart() method, the aircraft part can be loaded and constrained by the user. Once all parts have been loaded into the model and all loads and constraints have been applied, the user can choose to execute the plotRigidModel() method to visualize the model and make sure it accurately represents their problem. If the model appears as it should, the user can elect to run a static, buckling, normal mode, static aeroelastic, or dynamic flutter analysis.
- ***LoadSet*: This class is used to fascilitate the created of many loads that** cal be individually applied to a finite element model. Typically this class is not explicitly used. Instead the are created by the applyLoads method of the Model class.
- ***FlutterPoint*: Primarily as a way to fascilitate the interpolation of** flutter results generated from the flutterAnalysis method of Model.

Note: Currently the only avaliable part in the AeroComBAT package are wing parts, however this is likely to change as parts such as masses, fuselages and other types of aircraft parts are added.

5.2.1 MODEL

class `AeroComBAT.FEM.`**`Model`**

Creates a Model which is used to organize and analyze FEM.

The primary used of Model objects are to organize FEM's and analyze them. The Model object doesn't create any finite elements. Instead, it loads aircraft parts which contain various types of finite element structural models as well as aerodynamic models. The type of model will depend on the type of aircraft part added. Once all of the models are created and added to the model object, the model object will serve as the analysis primary interface used to manipulate the generated model.

Attributes

- *Kg (DOFxDOF np.array[float])*: This is the global stiffness matrix.
- ***Kgr ((DOF-CON)x(DOF-CON) np.array[float])*: This is the global reduced** stiffness matrix. In other words, the global stiffness matrix with the rows and columns corresponding to the constraints (CON) removed.
- *Fg (DOFx1 np.array[float])*: The global force vector.
- ***Fgr ((DOF-CON)x1 np.array[float])*: The global reduced force vector. In** other words, the global force vector with the rows corresponding to the constraints (CON) removed.
- *Mg (DOFxDOF np.array[float])*: The global mass matrix.

- ***Mgr ((DOF-CON)x(DOF-CON) np.array[float])*: The global reduced mass** matrix. In other words, the global mass matrix with the rows and columns corresponding to the constraints (CON) removed.
- ***Qg (DOFx1 np.array[float])*: The global force boundary condition vector.** This is where all of the nodal loads are stored before the system is assembled.
- ***nids (Array[int])*: This array contains all of the node IDs used within** the model.
- ***nodeDict (dict[NID,node])*: This dictionary is a mapping of the node IDs** used within the model to the corresponding node objects.
- ***elems (Array[obj])*: This array contains all of the element objects used** in the model.
- ***const (dict[NID,Array[DOF])*: This dictionary is a mapping of the node** IDs constrained and the corresponding degrees of freedom that are constrained.
- ***parts (dict[PID, part])*: This dictionary is a mapping of part ID's** (PID) and the aircraft part objects that are added to the model. Currently the only suported parts are wings.
- ***loads (dict[LID,int])*: This dictionary is a mapping of the load ID** (LID) and the load set objects.
- ***aeroBox (dict[PANID,panel])*: This dictionary is a mapping of the** aerodynamic panel ID's (PANID) and the aerodynamic panel objects used in the flutter analysis.
- ***SuperBeams (array[obj])*: This array contains all of the superbeam's** added to the model through addElements. In otherwords, this superbeam object is without an associated part.
- ***u (dict[str,1xDOF np.array[float]])*: This dictionary maps analysis** names to displacement results for a static analysis.
- ***freqs (1x(DOF-CON) np.array[float])*: This is a 1D array which holds the** frequencies of a normal modes analysis.

Methods

- *addElements*: A method to add individual elements to the model.
- ***addAircraftParts*: A method to add an Aircraft part to the model. This** is a much more effective method than addElements as when a part is added, the model can utilize all of the organizational and post processing methods built into the part.
- ***resetPointLoads*: A convenient way to reset all of the nodal loads in** the model to zero.
- ***resetResults*: A convenient way to clear the results in all of the** elements from a previous analysis. This method is subject to change as the way in which results are stored is likely to change.
- ***applyLoads*: A method to apply nodal loads as well as distributed loads** to a range of elements, all of the elements in a part, or all of the elements in the model.
- *applyConstraints*: A method to apply nodal constraints to the model.
- *staticAnalysis*: A method which conducts a linear static analysis.
- ***normalModesAnalysis*: A method which conducts a normal modes analysis on** the model.
- ***flutterAnalysis*: A method which conducts a linearized flutter pk-method** analysis on the model.
- *plotRigidModel*: A method to plot and visualize the model.
- ***plotDeformedModel*: A method to plot and visualize the results from an** analysis on the model.

`addAircraftParts`(*parts*)

A method to add an array of aircraft parts to the model.

This method is a more robust version of addElements. Provided an array of part objects, this method will add the parts to the model. This includes adding all of the elements and nodes to the model, as well as a few other pieces of information. In addition, if a wing has aerodynamic panels associated with it, these will also be added to the model.

Args

- *parts (Array[obj])*: An array of part objects.

Returns

- None

addElements (*elemarray*)

A method to add elements to the model.

Provided an array of elements, this method can add those elements to the model for analysis. This is a rather rudementary method as the post processing methods utilized by the parts are not at the users disposal for the elements added to the model in this way.

Args

- ***elemarray (Array[obj])*: Adds all of the elements in the array to** the model.

Returns

- None

Note: Currently supported elements include: SuperBeam, Tbeam.

---

applyConstraints (*NID*, *const*)

A method for applying nodal constraints to the model.

This method is the primary method for applying nodal constraints to the model.

Args

- *NID (int)*: The node ID of the node to be constrained.
- ***const (str, np.array[int])*: const can either take the form of a** string in order to take advantage of the two most common constraints being 'pin' or 'fix'. If a different constraint needs to be applied, const could also be a numpy array listing the DOF (integers 1-6) to be constrained.

Returns

- None

Note: When constraining nodes, only 0 displacement and rotation

---

constraints are currently supported.

applyLoads (*LID*, ***kwargs*)

A method to apply nodal and distributed loads to the model.

This method allows the user to apply nodal loads to nodes and distributed loads to elements within the model.

Args

- ***f (func)*: A function which, provided the provided a length 3 numpy** array representing a point in space, calculates the distributed load value at that point. See an example below:
- ***F (dict[NID,1x6 np.array[float]])*: A dictionary mapping a node ID** to the loads to be applied at that node ID.
- ***allElems (bool)*: A boolean value used to easily load all of the** elements which have been added to the model.
- ***PIDs (Array[int])*: An array containing part ID's, signifying that** all elements used by that part should be loaded.
- ***eids (Array[int])*: An array containing all of the element ID's** corresponding to all of the elements which should be loaded.

Returns

- None

Distributed load function example:

```
def f(x):
   vx = (1/10)*10*x[2]**2-7*x[2]-2.1
   vy = 10*x[2]**2-7*x[2]
   pz = 0
   mx = 0
   my = 0
   tz = (10*x[2]**2-7*x[2])/10+3*x[0]**2
   return np.array([vx,vy,pz,mx,my,tz])
```

Nodal load dictionary example:

```
F[NID] = np.array([Qx,Qy,P,Mx,My,T])
```

flutterAnalysis(*U_vec*, *kr_vec*, *M_vec*, *b*, *rho_0*, *nModes*, ***kwargs*)

Conducts a flutter analysis.

This method calculates the flutter modes and damping provided velocities, reduced frequencies, Mach numbers, and the reference semi-chord.

Args

- ***U_vec (1xN np.array[float])*: A vector of trial velocities where the** damping and frequency of all of the respective mode shapes will be calculated.
- ***kr_vec (1xM np.array[float])*: A vector of reduced frequencies for** which the AIC's will be calculated. The minimum possible value can be 0.
- ***M_vec (1xM np.array[float])*: A vector of mach numbers at which the** AIC's will be calculated. Currently interpolating results by Mach number aren't possible. As such, select mach numbers to be close to the suspected instability.
- *b (float)*: The reference semi-chord.
- *rho_0 (float)*: The reference density at sea level.
- ***nmodes (int)*: The number of modes to be considered for the flutter** analysis. For a composite cantilevered wing, 6 modes should usually be sufficient.

- *g (float)*: **A proportional structural damping term. Acceptable** ranges of g can be approximated between 0. and 0.05.
- *symxz (bool)*: **A boolean value indicating whether the aerodynamics** should be mirrored over the xz-plane.
- *rho_rat (1xN np.array[float])*: **An array of density ratios to allow** for flutter calculations at different altitudes.
- *analysis_name (str)*: **The string name to be associated with this** analysis. By default, this is chosen to be 'analysis_untitled'.

Returns

- None

Note: Currently static aeroelastic instability (divergence) cannot

---

be captured by AeroComBAT.

`normalModesAnalysis`(***kwargs*)

Conducts normal mode analysis.

This method conducts normal mode analysis on the model. This will calculate all of the unknown frequency eigenvalues and eigenvectors for the model, which can be plotted later.

Args

- *analysis_name (str)*: **The string name to be associated with this** analysis. By default, this is chosen to be 'analysis_untitled'.

Returns

- None

Note: There are internal loads that are calculated and stored within the model elements, however be aware that these loads are meaningless and are only retained as a means to display cross section warping.

`plotDeformedModel`(***kwargs*)

Plots the deformed model.

This method plots the deformed model results for a given analysis in the mayavi environement.

Args

- *analysis_name (str)*: The string identifier of the analysis.
- *figName (str)*: **The name of the figure. This is 'Rigid Model' by** default.
- *clr (1x3 tuple(int))*: **The color tuple or RGB values to be used for** plotting the reference axis for all beam elements. By default this color is black.
- *numXSects (int)*: **The number of cross-sections desired to be plotted** for all wing sections. The default is 2.
- *contour (str)*: **A string keyword to determine what analysis should** be plotted.
- *contLim (1x2 Array[float])*: **An array containing the lower and upper** contour limits.

- ***warpScale (float)*: The scaling factor used to magnify the cross** section warping displacement factor.
- ***displScale (float)*: The scaling fator used to magnify the beam** element displacements and rotations.
- ***mode (int)*: If the analysis name refers to a modal analysis, mode** refers to which mode from that analysis should be plotted.

Returns

- mayavi figure

`plotRigidModel`(***kwargs*)

Plots the rigid model.

This method plots the rigid model in the mayavi environement.

Args

- ***figName (str)*: The name of the figure. This is 'Rigid Model' by** default.
- ***clr (1x3 tuple(int))*: The color tuple or RGB values to be used for** plotting the reference axis for all beam elements. By default this color is black.
- ***numXSects (int)*: The number of cross-sections desired to be plotted** for all wing sections. The default is 2.

Returns

- mayavi figure

`resetPointLoads`()

A method to reset the point loads applied to the model.

This is a good method to reset the nodal loads applied to a model. This method will be useful when attempting to apply a series different analysis.

Args

- None

Returns

- None

`resetResults`()

A method to reset the results in a model.

This is a good method to reset the results in the model from a given analysis. This method will be useful when attempting to apply a series different analysis.

Args

- None

Returns

- None

staticAnalysis(*LID*, ***kwargs*)

Linear static analysis.

This method conducts a linear static analysis on the model. This will calculate all of the unknown displacements in the model, and save not only dispalcements, but also internal forces and moments in all of the beam elements.

Args

- ***LID (int)*: The ID corresponding to the load set to be applied to** the model.
- ***analysis_name (str)*: The string name to be associated with this** analysis. By default, this is chosen to be 'analysis_untitled'.

Returns

- None

5.2.2 LOAD SET

class AeroComBAT.FEM.**LoadSet**(*LID*)

Creates a Model which is used to organize and analyze FEM.

The primary use of LoadSet is to fascilitate the application of many different complex loads to a finite element model.

Attributes

- *LID (int)*: The integer identifier for the load set object.
- ***pointLoads (dict[pointLoads[NID,F])*: A dictionary mapping applied point** loads to the node ID's of the node where the load is applied.
- ***distributedLoads (dict[EID,f])*: A dictionary mapping the distributed** load vector to the element ID of the element where the load is applied.

Methods

- ***__init__*: The constructor of the class. This method initializes the** dictionaries used by the loads
- *addPointLoad*: Adds point loads to the pointLoads dictionary attribute.
- ***addDictibutedLoad*: Adds distributed loads to the distributedLoads** dictionary attribute.

addDistributedLoad(*f*, *eid*)

Initialized the load set ibject.

This method is a simple constructor for the load set object.

Args

- *LID (int)*: The integer ID linked with the load set object.

Returns

- None

addPointLoad(*F*, *NID*)

Initialized the load set ibject.

This method is a simple constructor for the load set object.

Args

- *LID (int)*: The integer ID linked with the load set object.

Returns

- None

5.2.3 FLUTTER POINT

class AeroComBAT.FEM.**FlutterPoint**(*FPID*, *U_vec*, *nModes*)

Creates a flutter point object.

The primary purpose for the flutter point class is to allow for easier post processing of the data from the flutter modes.

Attributes

- ***FPID (int)*: The integer identifier associated with the flutter point** object.
- ***U_vec (1xN np.array[float])*: A vector of the velocities where the** flutter point frequency and damping have been solved.
- ***omegaAeroDict(dict[U,array[float])*: This dictionary maps velocities to** the aerodynamic frequencies used to generate the AIC matricies.
- ***omegaRootDict(dict[U,array[float])*: This dictionary maps velocities to** the root frequencies of the flutter mode solution for particular reduced frequencies.
- ***gammaDict(dict[U,array[float])*: This dictionary maps velocities to** the root damping of the flutter mode solution for particular reduced frequencies.
- ***gammaDict(dict[U,array[float])*: This dictionary maps velocities to** the root mode shape of the flutter mode solution for particular reduced frequencies.
- ***omega (array[float])*: An array of floats which are the** flutter mode frequencies corresponding to the velocities in U_vec.
- ***gamma (array[float])*: An array of floats which are the** flutter mode damping values corresponding to the velocities in U_vec.
- ***shape (array[MxN np.array[float]])*: An MxL numpy array which** contain the eigenvector solutions of the flutter mode. The values in the eigenvectors are the coefficient weighting factors for the normal mode shapes.

Methods

- ***__init__*: The constructor of the class. This method initializes the** attributes of the model, as well as the flutter
- ***saveSol*: Saves solutions to the flutter equation for the particular** mode.
- ***interpOmegaRoot*: Interpolates the flutter mode frequency, damping and** mode shapes for the different velocities.

__init__(*FPID*, *U_vec*, *nModes*)

Creates a flutter point object.

This is the constructor for the flutter point object.

Args

- *FPID (int)*: The integer ID linked with the flutter point object.
- ***U_vec (1xN np.array[float])*: An array of velocities where the** flutter problem will be solved.
- ***nModes (int)*: The number of modes that are used for the flutter** solution.

Returns

- None

interpOmegaRoot()

Interpolates correct dynamic frequencies and damping.

From the data saved using the saveSol method, this method interpolates the correct dynamic frequencies and damping for the different flutter velocities.

Args

- None

Returns

- None

saveSol(*U*, *omega_aero*, *omega_root*, *gamma_root*, *shape*)

Saves data from the flutter solutions.

This method saves the damping, frequencies and mode shapes for the different flutter velocities and reduced frequencies.

Args

- *U (float)*: The flutter velocity of the data.
- ***omega_aero (float)*: The aerodynamic frequency corresponding to the** reduced frequency.
- ***omega_root (float)*: The root frequency corresponding to the** flutter solution of the particular aerodynamic frequency.
- *gamma_root (float)*: The root damping of the flutter solution
- ***shape (1xM np.array[float])*: The mode shape of the flutter** solution.

Returns

- None

5.3 Aircraft Parts Module

This module contains a library of classes devoted to modeling aircraft parts.

The main purpose of this library is to model various types of aircraft parts. Currently only wing objects are suported, however in the future it is possible that fuselages as well as other parts will be added.

SUMARRY OF THE CLASSES

- ***Wing*: Creates a wing aircraft. This wing is capable of modeling the** structures of the aircraft wing as well as the aerodynamics. The structures are modeled with a combination of beam models currently, however it is likely that Ritz method laminates will also incorporated for buckling prediction purposes. The aerodynamics are currently modeled with potential flow doublet panels.

5.3.1 Wing

class `AeroComBAT.AircraftParts.`**`Wing`**(*PID*, *p1*, *p2*, *croot*, *ctip*, *x0_spar*, *xf_spar*, *Y_rib*, *n_ply*, *m_ply*, *mat_lib*, ***kwargs*)

`__init__`(*PID*, *p1*, *p2*, *croot*, *ctip*, *x0_spar*, *xf_spar*, *Y_rib*, *n_ply*, *m_ply*, *mat_lib*, ***kwargs*)

Creates a wing object.

This object represents a wing and contains both structural and aerodynamic models.

Args

- *PID (int)*: The integer ID linked to this part.
- *p1 (1x3 np.array[float])*: The initial x,y,z coordinates of the wing.
- *p2 (1x3 np.array[float]*: The final x,y,z coordinates of the wing.
- *croot (float)*: The root chord length.
- *ctip (float)*: The tip chord length.
- ***x0_spar (float)*: The non-dimensional starting location of the cross** section.
- ***xf_spar (float)*: The non-dimensional ending location of the cross** section.
- ***Y_rib (1xN Array[float])*: The non-dimensional rib locations within** the wing. This dimension is primarily used to create wing-sections which primarily define the buckling span's for laminate objects.
- ***n_ply (1xM Array[int])*: An array of integers specifying the number** plies to be used in the model. Each integer refers to the number of plies to be used for at a given orientation.
- ***m_ply (1xM Array[int])*: An array of integers specifying the** material ID to be used for the corresponding number of plies in n_ply at a given orientation.
- ***th_ply (1xM Array[int])*: An array of floats specifying the** degree orientations of the plies used by the lamiantes in the model.
- ***mat_lib (obj)*: A material library containing all of the material** objets to be used in the model.
- ***name (str)*: The name of the airfoil section to be used for cross** section generation.
- *wing_SNID (int)*: The first node ID associated with the wing.
- *wing_SEID (int)*: The first beam element ID associated with the wing.
- *wing_SSBID (int)*: The first superbeam ID associated with the wing.

- *SXID (int)*: The starting cross-section ID used by the wing.
- ***noe (float)*: The number of beam elements to be used in the wing per** unit length.
- ***n_orients (int)*: The number of fiber orientations to be used in** each laminate.
- ***n_lams (int)*: The number of laminates required to mesh the desired** cross-section.
- ***meshSize (float)*: The maximum aspect ratio a 2D element may have in** the cross-section.
- *ref_ax (str)*: The reference axis to be loaded in the wing.
- ***chordVec (1x3 np.array[float])*: This numpy array is used to orient** the cross-section in 3D space. It corresponds to the local unit x- vector in the cross-section, expressed in the global frame.
- ***typeXSect (str)*: The type of cross-section to be used by the wing** structure. Currently the suported typed are 'boxbeam', 'laminate', and 'rectBoxBeam'. See the meshing class in the structures module for more details.

Returns

- None

`addLiftingSurface`(*SID*, *x1*, *x2*, *x3*, *x4*, *nspan*, *nchord*)

Adds a potential flow lifting surface to the model.

This method adds a potential flow panel aerodynamic model to the wing part. The x1,x2,x3, and x4 points correspond to the root leading edge, root trailing edge, tip trailing edge, and tip leading edge of the wing respectively. Currently the only suported types of panels are doublet- lattice panels to be used for unsteady aerodynamic models.

Args

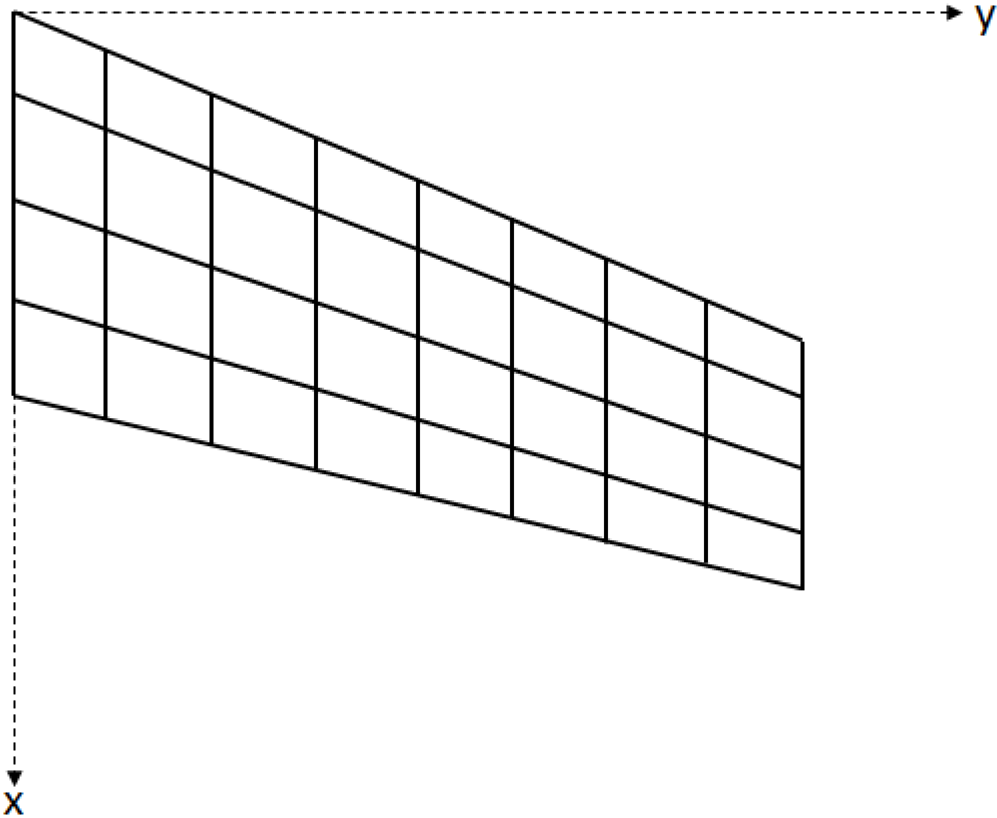
- ***SID (int)*: The lifting surface integer identifier corresponding to** the lifting surface.
- ***x1 (1x3 numpy array)*: The point in 3D space corresponding to the** root leading edge point of the lifting surface.
- ***x2 (1x3 numpy array)*: The point in 3D space corresponding to the** root trailing edge point of the lifting surface.
- ***x3 (1x3 numpy array)*: The point in 3D space corresponding to the** tip trailing edge point of the lifting surface.
- ***x4 (1x3 numpy array)*: The point in 3D space corresponding to the** tip leading edge point of the lifting surface.
- ***nspan (int)*: The number of boxes to be used in the spanwise** direction.
- ***nchord (int)*: The number of boxes to be used in the chordwise** direction.

Returns

- None

Note: Mutliple surfaces could be added to the wing part.

> **Warning:** In order to use the doublet lattice method, the chord lines of the lifting surface must run in the x-direction, and there can be no geometric angles of attack present. The geometry of a general wing can be seen in the figure below:



plotRigidWing(***kwargs*)

Plots the rigid wing.

This method plots the rigid model of the wing. This includes the reference axis of the beams, the cross-sections of the beams, and the lifting surfaces that make up the wing. This is an excellent check to perform before adding the part to a FEM model.

Args

- ***figName (str)*: The name of the MayaVi figure. 'Rigid Wing' by** default.
- ***numXSects (int)*: The number of cross-sections that each wing** section will display. By default it is 2.
- ***color (1x3 tuple(int))*: This is a length 3 tuple to be used as the** color of the beam reference axes. Black by default.

Returns

- None

5.4 Structures Module

This module contains a library of classes devoted to structural analysis.

The primary purpose of this library is to fascilitate the ROM (reduced order modeling) of structures that can simplified to beams. The real power of this library comes from it's the XSect class. This class can create and analyze a cross-

section, allowing the user to accurately model a nonhomogeneous (made of multiple materials) anisotropic (materials that behave anisotropically such as composites) complex cross-sections.

It should be noted that classes are ordered by model complexity. The further down the structures.py library, the more complex the objects, often requiring multiple of their predecessors. For example, the CQUADX class requires four node objects and a material object.

SUMARRY OF THE CLASSES

- *Node*: Creates a node object with 3D position.
- *Material*: Creates a material object, generating the 3D constitutive relations.
- ***MicroMechanics*: Class to fascilitate the calculation of composite stiffnesses** using micro-mechanical models where fibers are long and continuous.
- ***CQUADX*: Creates a 2D linear quadrilateral element, mainly used to fascilitate cross-sectional analysis, this class could be** such that they could also be used to create plate or laminate element objects as well.
- ***MaterialLib*: Creates a material library object meant to hold many material** objects.
- *Ply*: Creates ply objects which are used in the building of a laminate object.
- ***Laminate*: Creates laminate objects which could be used for CLT (classical** lamination theory) analysis as well as to be used in building a beam cross-section.
- ***XSect*: Creates a cross-section object which can be used in the ROM of a beam** with a non-homogeneous anisotropic cross-section. Currently only supports simple box beam cross-section (i.e., four laminates joined together to form a box), however outer mold lines can take the shape of airfoil profiles. See the Airfoil class in AircraftParts.py for more info.
- *TBeam*: Creates a single Timoshenko beam object for FEA.
- ***SuperBeam*: Creates a super beam object. This class is mainly used to automate** the creation of many connected TBeam objects to be used late for FEA.
- ***WingSection*: A class which creates and holds many super beams, each of which** could have different cross-sections. It also helps to dimensionalize plates for simple closed-form composite buckling load aproximations.

Note: Currently the inclusion of thermal strains are not supported for any structural model.

5.4.1 NODE

class `AeroComBAT.Structures.`**`Node`**(*NID*, *x*)

Creates a node object.

Node objects could be used in any finite element implementation.

Attributes

- *NID (int)*: The integer identifier given to the object.
- ***x (Array[float])*: An array containing the 3 x-y-z coordinates of the** node.
- ***summary (str)*: A string which is a tabulated respresentation and** summary of the important attributes of the object.

Methods

- *printSummary*: **This method prints out basic information about the node** object, such as it's node ID and it's x-y-z coordinates

`__init__`(*NID*, *x*)
Initializes the node object.

Args

- *nid (int)*: The desired integer node ID
- *x (Array[float])*: The position of the node in 3D space.

Returns

- None

`printSummary`()
Prints basic information about the node.

The printSummary method prints out basic node attributes in an organized fashion. This includes the node ID and x-y-z global coordinates.

Args

- None

Returns

- A printed table including the node ID and it's coordinates

5.4.2 MATERIAL

class `AeroComBAT.Structures.`**`Material`**(*MID*, *name*, *matType*, *mat_constants*, *mat_t*, ***kwargs*)
creates a linear elastic material object.

This class creates a material object which can be stored within a material library object. The material can be in general orthotropic.

Attributes

- *name (str)*: A name for the material.
- *MID (int)*: An integer identifier for the material.
- ***matType (str)*: A string expressing what type of material it is.** Currently, the supported materials are isotropic, transversely isotropic, and orthotropic.
- ***summary (str)*: A string which is a tabulated respresentation and** summary of the important attributes of the object.
- ***t (float)*: A single float which represents the thickness of a ply if** the material is to be used in a composite.
- ***rho (float)*: A single float which represents the density of the** materials.
- ***Smat (6x6 numpy Array[float])*: A numpy array representing the** compliance matrix in the fiber coordinate system.*
- ***Cmat (6x6 numpy Array[float])*: A numpy array representing the** stiffness matrix in the fiber coordinate system.*

Methods

- ***printSummary*: This method prints out basic information about the** material, including the type, the material constants, material thickness, as well as the tabulated stiffness or compliance matricies if requested.

Note: The CQUADX element assumes that the fibers are oriented along the (1,0,0) in the global coordinate system.

`__init__` (*MID*, *name*, *matType*, *mat_constants*, *mat_t*, ***kwargs*)

Creates a material object

The main purpose of this class is assembling the constitutive relations. Regardless of the analysis

Args

- *MID (int)*: Material ID.
- *name (str)*: Name of the material.
- ***matType (str)*: The type of the material. Supported material types** are "iso", "trans_iso", and "ortho".
- ***mat_constants (1xX Array[Float])*: The requisite number of material** constants required for any structural analysis. Note, this array includes the material density. For example, an isotropic material needs 2 elastic material constants, so the total length of mat_constants would be 3, 2 elastic constants and the density.
- *mat_t (float)*: The thickness of 1-ply of the material
- ***th (1x3 Array[float])*: The angles about which the material can be** rotated when it is initialized. In degrees.

Returns

- None

Note: While this class supports material direction rotations, it is more robust to simply let the CQUADX and Mesher class handle all material rotations.

`printSummary` (***kwargs*)

Prints a tabulated summary of the material.

This method prints out basic information about the material, including the type, the material constants, material thickness, as well as the tabulated stiffness or compliance matricies if requested.

Args

- ***compliance (str)*: A boolean input to signify if the compliance** matrix should be printed.
- ***stiffness (str)*: A boolean input to signify if the stiffness matrix** should be printed.

Returns

- **String print out containing the material name, as well as material** constants and other defining material attributes. If requested this includes the material stiffness and compliance matricies.

5.4.3 CQUADX

class AeroComBAT.Structures.**CQUADX**(*EID*, *nodes*, *MID*, *matLib*, ***kwargs*)

Creates a linear, 2D 4 node quadrilateral element object.

The main purpose of this class is to assist in the cross-sectional analysis of a beam, however it COULD be modified to serve as an element for 2D plate or laminate FE analysis.

Attributes

- *type (str)*: A string designating it a CQUADX element.
- ***xsect (bool)*: States whether the element is to be used in cross-** sectional analysis.
- ***th (1x3 Array[float])*: Array containing the Euler-angles expressing how** the element constitutive relations should be rotated from the material fiber frame to the global CSYS. In degrees.
- *EID (int)*: An integer identifier for the CQUADX element.
- ***MID (int)*: An integer refrencing the material ID used for the** constitutive relations.
- ***NIDs (1x4 Array[int])*: Contains the integer node identifiers for the** node objects used to create the element.
- ***nodes (1x4 Array[obj])*: Contains the properly ordered nodes objects** used to create the element.
- ***xs (1x4 np.array[float])*: Array containing the x-coordinates of the** nodes used in the element
- ***ys (1x4 np.array[float])*: Array containing the y-coordinates of the** nodes used in the element
- *rho (float)*: Density of the material used in the element.
- *mass (float)*: Mass per unit length (or thickness) of the element.
- ***U (12x1 np.array[float])*: This column vector contains the CQUADXs** 3 DOF (x-y-z) displacements in the local xsect CSYS due to cross- section warping effects.
- ***Eps (6x4 np.array[float])*: A matrix containing the 3D strain state** within the CQUADX element.
- ***Sig (6x4 np.array[float])*: A matrix containing the 3D stress state** within the CQUADX element.

Methods

- ***x*: Calculates the local xsect x-coordinate provided the desired master** coordinates eta and xi.
- ***y*: Calculates the local xsect y-coordinate provided the desired master** coordinates eta and xi.
- ***J*: Calculates the jacobian of the element provided the desired master** coordinates eta and xi.
- ***resetResults*: Initializes the displacement (U), strain (Eps), and** stress (Sig) attributes of the element.
- ***getDeformed*: Provided an analysis has been conducted, this method** returns 3 2x2 np.array[float] containing the element warped displacements in the local xsect CSYS.
- ***getStressState*: Provided an analysis has been conducted, this method** returns 3 2x2 np.array[float] containing the element stress at four points. The 3D stress state is processed to return the Von-Mises or Maximum Principal stress state.
- ***printSummary*: Prints out a tabulated form of the element ID, as well** as the node ID's referenced by the element.

J(*eta*, *xi*)

Calculates the jacobian at a point in the element.

This method calculates the jacobian at a local point within the element provided the master coordinates eta and xi.

Args

•*eta (float)*: The eta coordinate in the master coordinate domain.*

•*xi (float)*: The xi coordinate in the master coordinate domain.*

Returns

•***Jmat (3x3 np.array[float])*: The stress-resultant transformation** array.

Note: Xi and eta can both vary between -1 and 1 respectively.

`__init__`(*EID*, *nodes*, *MID*, *matLib*, ***kwargs*)

Initializes the element.

Args

•*EID (int)*: An integer identifier for the CQUADX element.

•***nodes (1x4 Array[obj])*: Contains the properly ordered nodes objects** used to create the element.

•***MID (int)*: An integer refrencing the material ID used for the** constitutive relations.

•***matLib (obj)*: A material library object containing a dictionary** with the material corresponding to the provided MID.

•***xsect (bool)*: A boolean to determine whether this quad element is** to be used for cross-sectional analysis. Defualt value is True.

•***th (1x3 Array[float])*: Array containing the Euler-angles expressing** how the element constitutive relations should be rotated from the material fiber frame to the global CSYS. In degrees.

Returns

•None

Note: The reference coordinate system for cross-sectional analysis is a

---

local coordinate system in which the x and y axes are planer with the element, and the z-axis is perpendicular to the plane of the element.

`getDeformed`(***kwargs*)

Returns the warping displacement of the element.

Provided an analysis has been conducted, this method returns 3 2x2 np.array[float] containing the element warped displacements in the local xsect CSYS.

Args

•***warpScale (float)*: A multiplicative scaling factor intended to** exagerate the warping displacement within the cross-section.

Returns

•***xdef (2x2 np.array[float])*: warped x-coordinates at the four corner** points.

- ***ydef (2x2 np.array[float])*: warped y-coordinates at the four corner** points.
- ***zdef (2x2 np.array[float])*: warped z-coordinates at the four corner** points.

getStressState(*crit='VonMis'*)
Returns the stress state of the element.

Provided an analysis has been conducted, this method returns a 2x2 np.array[float] containing the element the 3D stress state at the four guass points by default.*

Args

- ***crit (str)*: Determines what criteria is used to evaluate the 3D** stress state at the sample points within the element. By default the Von Mises stress is returned. Currently supported options include: Von Mises ('VonMis'), maximum principle stress ('MaxPrin'), the minimum principle stress ('MinPrin'), and the local cross-section stress states 'sig_xx' where the subindeces can go from 1-3. The keyword 'none' is also an option.

Returns

- ***sigData (2x2 np.array[float])*: The stress state evaluated at four** points within the CQUADX element.

Note: The XSect method calcWarpEffects is what determines where strain

---

and stresses are sampled. By default it samples this information at the Guass points where the stress/strain will be most accurate.

printSummary(*nodes=False*)
A method for printing a summary of the CQUADX element.

Prints out a tabulated form of the element ID, as well as the node ID's referenced by the element.

Args

- None

Returns

- ***summary (str)*: Prints the tabulated EID, node IDs and material IDs** associated with the CQUADX element.

resetResults()
Resets stress, strain and warping displacement results.

Method is mainly intended to prevent results for one analysis or sampling location in the matrix to effect the results in another.

Args

- None

Returns

- None

x (*eta*, *xi*)
Calculate the x-coordinate within the element.

Calculates the local xsect x-coordinate provided the desired master coordinates eta and xi.

Args

- *eta (float)*: The eta coordinate in the master coordinate domain.*
- *xi (float)*: The xi coordinate in the master coordinate domain.*

Returns

- *x (float)*: The x-coordinate within the element.

Note: Xi and eta can both vary between -1 and 1 respectively.

y (*eta*, *xi*)
Calculate the y-coordinate within the element.

Calculates the local xsect y-coordinate provided the desired master coordinates eta and xi.

Args

- *eta (float)*: The eta coordinate in the master coordinate domain.*
- *xi (float)*: The xi coordinate in the master coordinate domain.*

Returns

- ‘y (float)’: The y-coordinate within the element.

Note: Xi and eta can both vary between -1 and 1 respectively.

5.4.4 MATERIAL LIBRARY

class `AeroComBAT.Structures.`**`MaterialLib`**
Creates a material library object.

This material library holds the materials to be used for any type of analysis. Furthermore, it can be used to generate new material objects to be automatically stored within it. See the Material class for suported material types.

Attributes

- ***matDict (dict)*: A dictionary which stores material objects as the** values with the MIDs as the associated keys.

Methods

- *addMat*: Adds a material to the MaterialLib object dictionary.
- *getMat*: Returns a material object provided an MID
- ***printSummary*: Prints a summary of all of the materials held within the** matDict dictionary.

__init__()

Initialize MaterialLib object.

The initialization method is mainly used to initialize a dictionary which houses material objects.

Args

- None

Returns

- None

addMat(*MID*, *mat_name*, *mat_type*, *mat_constants*, *mat_t*, ***kwargs*)

Add a material to the MaterialLib object.

This is the primary method of the class, used to create new material obects and then add them to the library for later use.

Args

- *MID (int)*: Material ID.
- *name (str)*: Name of the material.
- ***matType (str)*: The type of the material. Supported material types** are "iso", "trans_iso", and "ortho".
- ***mat_constants (1xX Array[Float])*: The requisite number of material** constants required for any structural analysis. Note, this array includes the material density. For example, an isotropic material needs 2 elastic material constants, so the total length of mat_constants would be 3, 2 elastic constants and the density.
- *mat_t (float)*: The thickness of 1-ply of the material
- ***th (1x3 Array[float])*: The angles about which the material can be** rotated when it is initialized. In degrees.
- ***overwrite (bool)*: Input used in order to define whether the** material being added can overwrite another material already held by the material library with the same MID.

Returns

- None

getMat(*MID*)

Method that returns a material from the material libary

Args

- *MID (int)*: The ID of the material which is desired

Returns

- '(obj): A material object associated with the key MID

printSummary()

Prints summary of all Materials in MaterialLib

A method used to print out tabulated summary of all of the materials held within the material library object.

Args

- None

Returns

- (str): A tabulated summary of the materials.

5.4.5 PLY

class `AeroComBAT.Structures.`**`Ply`**(*Material*, *th*)

Creates a CLT ply object.

A class inspired by CLT, this class can be used to generate laminates to be used for CLT or cross-sectional analysis. It is likely that ply objects won't be created individually and then assembeled into a lamiante. More likely is that the plies will be generated within the laminate object. It should also be noted that it is assumed that the materials used are effectively at most transversely isotropic.

Attributes

- *E1 (float)*: Stiffness in the fiber direction.
- *E2 (float)*: Stiffness transverse to the fiber direction.
- *nu_12 (float)*: In plane poisson ratio.
- *G_12 (float)*: In plane shear modulus.
- *t (float)*: Thickness of the ply.
- ***Qbar (1x6 np.array[float])*: The terms in the rotated, reduced stiffness** matrix. Ordering is as follows: [Q11,Q12,Q16,Q22,Q26,Q66]
- ***MID (int)*: An integer refrencing the material ID used for the** constitutive relations.
- ***th (float)*: The angle about which the fibers are rotated in the plane** in degrees.

Methods

- ***genQ*: Given the in-plane stiffnesses used by the material of the ply,** the method calculates the terms of ther reduced stiffness matrix.
- ***printSummary*: This prints out a summary of the object, including** thickness, referenced MID and in plane angle orientation theta in degrees.

`__init__`(*Material*, *th*)

Initializes the ply.

This method initializes information about the ply such as in-plane stiffness repsonse.

Args

- ***Material (obj)*: A material object, most likely coming from a** material library.
- ***th (float)*: The angle about which the fibers are rotated in the** plane in degrees.

Returns

- None

genQ(*E1*, *E2*, *nu12*, *G12*)

A method for calculating the reduced compliance of the ply.

Intended primarily as a private method but left public, this method, for those unfarmiliar with CLT, calculates the terms in the reduced stiffness matrix given the in plane ply stiffnesses. It can be thus inferred that this requires the assumption of plane stres. This method is primarily used during the ply instantiation.

Args

- *E1 (float)*: The fiber direction stiffness.
- *E2 (float)*: The stiffness transverse to the fibers.
- *nu12 (float)*: The in-plane poisson ratio.
- *G12 (float)*: The in-plane shear stiffness.

Returns

- ***(1x4 np.array[float])*: The terms used in the reduced stiffness** matrix. The ordering is: [Q11,Q12,Q22,Q66].

printSummary()

Prints a summary of the ply object.

A method for printing a summary of the ply properties, such as the material ID, fiber orientation and ply thickness.

Args

- None

Returns

- *(str)*: Printed tabulated summary of the ply.

5.4.6 LAMINATE

class AeroComBAT.Structures.**Laminate**(*n_i_tmp*, *m_i_tmp*, *matLib*, ***kwargs*)

Creates a CLT laminate object.

This class has two main uses. It can either be used for CLT analysis, or it can be used to build up a 2D mesh for a descretized cross-section.

Attributes

- ***mesh (NxM np.array[int])*: This 2D array holds NIDs and is used** to represent how nodes are organized in the 2D cross-section of the laminate.
- ***xmesh (NxM np.array[int])*: This 2D array holds the rigid x-coordinates** of the nodes within the 2D descretization of the laminate on the local xsect CSYS.
- ***ymesh (NxM np.array[int])*: This 2D array holds the rigid y-coordinates** of the nodes within the 2D descretization of the laminate on the local xsect CSYS.
- ***zmesh (NxM np.array[int])*: This 2D array holds the rigid z-coordinates** of the nodes within the 2D descretization of the laminate on the local xsect CSYS.
- *H (float)*: The total laminate thickness.

- *rho_A (float)*: The laminate area density.
- ***plies (1xN array[obj])*: Contains an array of ply objects used to** construct the laminate.
- *t (1xN array[float])*: An array containing all of the ply thicknesses.
- ***ABD (6x6 np.array[float])*: The CLT 6x6 matrix relating in-plane strains** and curvatures to in-plane force and moment resultants.
- ***abd (6x6 np.array[float])*: The CLT 6x6 matrix relating in-plane forces** and moments resultants to in-plane strains and curvatures.
- ***z (1xN array[float])*: The z locations of laminate starting and ending** points. This system always starts at -H/2 and goes to H/2
- ***equivMat (obj)*: This is orthotropic material object which exhibits** similar in-plane stiffnesses.
- ***forceRes (1x6 np.array[float])*: The applied or resulting force and** moment resultants generated during CLT analysis.
- ***globalStrain (1x6 np.array[float])*: The applied or resulting strain** and curvatures generated during CLT analysis.

Methods

- ***printSummary*: This method prints out defining attributes of the** laminate, such as the ABD matrix and layup schedule.

`__init__`(*n_i_tmp*, *m_i_tmp*, *matLib*, ***kwargs*)

Initializes the Laminate object

The way the laminate initialization works is you pass in two-three arrays and a material library. The first array contains information about how many plies you want to stack, the second array determines what material should be used for those plies, and the third array determines at what angle those plies lie. The class was developed this way as a means to fascilitate laminate optimization by quickly changing the number of plies at a given orientation and using a given material.

Args

- ***n_i_tmp (1xN array[int])*: An array containing the number of plies** using a material at a particular orientation such as: (theta=0,theta=45...)
- ***m_i_tmp (1xN array[int])*: An array containing the material to be** used for the corresponding number of plies in the n_i_tmp array
- ***matLib (obj)*: The material library holding different material** objects.
- *sym (bool)*: Whether the laminate is symetric. (False by default)
- ***th (1xN array[float])*: An array containing the orientation at which** the fibers are positioned within the laminate.

Returns

- None

Note: If you wanted to create a [0_2/45_2/90_2/-45_2]_s laminate of the same material, you could call laminate as:

lam = Laminate([2,2,2,2],[1,1,1,1],matLib,sym=True)

Or:

lam = Laminate([2,2,2,2],[1,1,1,1],matLib,sym=True,th=[0,45,90,-45])

Both of these statements are equivalent. If no theta array is provided and n_i_tmp is not equal to 4, then Laminate will default your fibers to all be running in the 0 degree orientation.

---

printSummary(***kwargs*)

Prints a summary of information about the laminate.

This method can print both the ABD matrix and ply information schedule of the laminate.

Args

- ***ABD (bool)*: This optional argument asks whether the ABD matrix** should be printed.
- ***decimals (int)*: Should the ABD matrix be printed, python should** print up to this many digits after the decimal point.
- ***plies (bool)*: This optional argument asks whether the ply schedule** for the laminate should be printed.

Returns

- None

5.4.7 MESHER

class AeroComBAT.Structures.**Mesher**

Meshes cross-section objects

This class is used to descritize cross-sections provided laminate objects. Currently only two cross-sectional shapes are supported. The first is a box beam using an airfoil outer mold line, and the second is a hollow tube using as many laminates as desired. One of the main results is the population of the nodeDict and elemDict attributes for the cross-section.

Attributes

- None

Methods

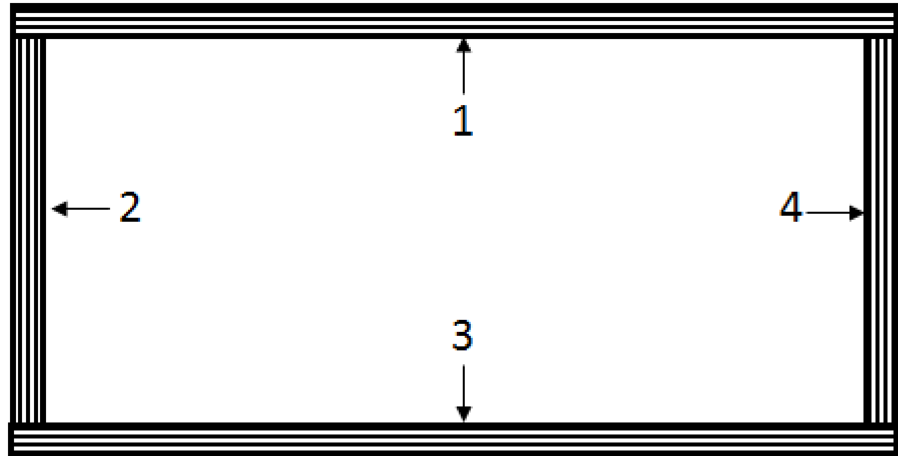
- ***boxBeam*: Taking several inputs including 4 laminate objects and meshes** a 2D box beam cross-section.
- *laminate*: Meshes the cross-section of a single laminate.
- ***cylindricalTube*: Taking several inputs including n laminate objects and** meshes a 2D cylindrical tube cross-section.
- ***rectBoxBeam*: Meshes a rectangular cross-section, but it is more** restrictive than boxBeam method. In this method, each of the four laminates must have the same number of plies, each of which are the same thickness.

boxBeam(*xsect*, *meshSize*, *x0*, *xf*, *matlib*)

Meshes a box beam cross-section.

This meshing routine takes several parameters including a cross-section object *xsect*. This cross-section object should also contain the laminate objects used to construct it. There are no restrictions place on these laminates. Furthermore the outer mold line of this cross- section can take the form of any NACA 4-series

airfoil. Finally, the convention is that for the four laminates that make up the box-beam, the the first ply in the laminate (which in CLT corresponds to the last ply in the stack) is located on the outside of the box beam. This convention can be seen below:



Args

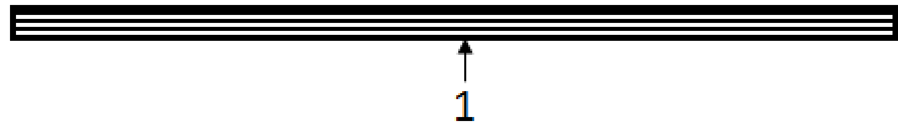
- *xsect (obj)*: The cross-section object to be meshed.
- *meshSize (int)*: The maximum aspect ratio an element can have
- ***x0 (float)*: The non-dimensional starting point of the cross-section** on the airfoil.
- ***xf (float)*: The non-dimesnional ending point of the cross-section** on the airfoil.
- ***matlib (obj)*: The material library object used to create CQUADX** elements.

Returns

- None

`laminate` (*xsect*, *meshSize*, *x0*, *xf*, *matlib*)
Meshes laminate cross-section.

This method meshes a simple laminate cross-section. It is assumed that the unit normal vector of the laminate points in the y-direction. This method only requires one laminate, which can take any shape. The cross- section geometry can be seen below:



Args

- *xsect (obj)*: The cross-section object to be meshed.
- *meshSize (int)*: The maximum aspect ratio an element can have
- ***x0 (float)*: The non-dimensional starting point of the cross-section** on the airfoil.
- ***xf (float)*: The non-dimesnional ending point of the cross-section** on the airfoil.
- ***matlib (obj)*: The material library object used to create CQUADX** elements.

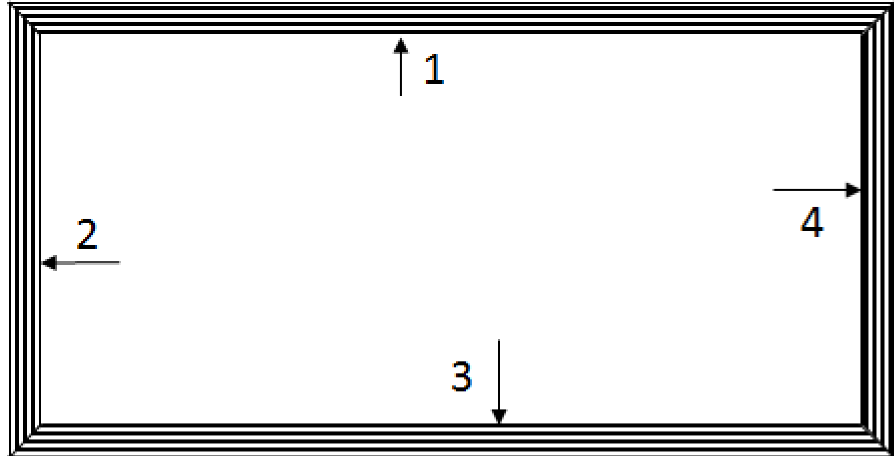
Returns

•None

rectBoxBeam (*xsect*, *meshSize*, *x0*, *xf*, *matlib*)

Meshes a box beam cross-section.

This method meshes a similar cross-section as the boxBeam method. The geometry of this cross-section can be seen below. The interfaces between the laminates is different, and more restrictive. In this case all of the laminates must have the same number of plies, which must also all be the same thickness.



Args

- *xsect (obj)*: The cross-section object to be meshed.
- *meshSize (int)*: The maximum aspect ratio an element can have
- ***x0 (float)*: The non-dimensional starting point of the cross-section** on the airfoil.
- ***xf (float)*: The non-dimesnional ending point of the cross-section** on the airfoil.
- ***matlib (obj)*: The material library object used to create CQUADX** elements.

Returns

- None

5.4.8 CROSS-SECTION

class AeroComBAT.Structures.**XSect** (*XID*, *Airfoil*, *xdim*, *laminates*, *matlib*, ***kwargs*)

Creates a beam cross-section object,

This cross-section can be made of multiple materials which can be in general anisotropic. This is the main workhorse within the structures library.

Attributes

- ***Color (touple)*: A length 3 touple used to define the color of the** cross-section.
- ***Airfoil (obj)*: The airfoil object used to define the OML of the cross-** section.
- ***typeXSect (str)*: Defines what type of cross-section is to be used.** Currently the only supported type is 'box'.
- ***normalVector (1x3 np.array[float])*: Expresses the normal vector of the** cross-section.

- ***nodeDict (dict)*: A dictionary of all nodes used to descretize the** cross-section surface. The keys are the NIDs and the values stored are the Node objects.
- ***elemDict (dict)*: A dictionary of all elements used to descretize the** cross-section surface. the keys are the EIDs and the values stored are the element objects.
- ***X (ndx6 np.array[float])*: A very large 2D array. This is one of the** results of the cross-sectional analysis. This array relays the force and moment resultants applied to the cross-section to the nodal warping displacements exhibited by the cross-section.
- ***Y (6x6 np.array[float])*: This array relays the force and moment** resultants applied to the cross-section to the rigid section strains and curvatures exhibited by the cross-section.
- ***dXdz (ndx6 np.array[float])*: A very large 2D array. This is one of the** results of the cross-sectional analysis. This array relays the force and moment resultants applied to the cross-section to the gradient of the nodal warping displacements exhibited by the cross-section with respect to the beam axis.
- ***xt (float)*: The x-coordinate of the tension center (point at which** tension and bending are decoupled)
- ***yt (float)*: The y-coordinate of the tension center (point at which** tension and bending are decoupled)
- ***xs (float)*: The x-coordinate of the shear center (point at which shear** and torsion are decoupled)
- ***ys (float)*: The y-coordinate of the shear center (point at which shear** and torsion are decoupled)
- ***refAxis (3x1 np.array[float])*: A column vector containing the reference** axis for the beam.
- ***bendAxes (2x3 np.array[float])*: Contains two row vectors about which** bending from one axis is decoupled from bending about the other.
- ***F_raw (6x6 np.array[float])*: The 6x6 compliance matrix that results** from cross-sectional analysis. This is the case where the reference axis is at the origin.
- ***K_raw (6x6 np.array[float])*: The 6x6 stiffness matrix that results** from cross-sectional analysis. This is the case where the reference axis is at the origin.
- ***F (6x6 np.array[float])*: The 6x6 compliance matrix for the cross-** section about the reference axis. The reference axis is by default at the shear center.
- ***K (6x6 np.array[float])*: The 6x6 stiffness matrix for the cross-** section about the reference axis. The reference axis is by default at the shear center.
- ***T1 (3x6 np.array[float])*: The transformation matrix that converts** strains and curvatures from the local xsect origin to the reference axis.
- ***T2 (3x6 np.array[float])*: The transformation matrix that converts** forces and moments from the local xsect origin to the reference axis.
- ***x_m (1x3 np.array[float])*: Center of mass of the cross-section about in** the local xsect CSYS
- ***M (6x6 np.array[float])*: This mass matrix relays linear and angular** velocities to linear and angular momentum of the cross-section.

Methods

- ***resetResults*: This method resets all results (displacements, strains** and stresse) within the elements used by the cross-section object.
- ***calcWarpEffects*: Given applied force and moment resultants, this method** calculates the warping displacement, 3D strains and 3D stresses within the elements used by the cross-section.
- ***printSummary*: This method is used to print characteristic attributes of** the object. This includes the elastic, shear and mass centers, as well as the stiffness matrix and mass matrix.

- ***plotRigid*: This method plots the rigid cross-section shape, typically** in conjunction with a full beam model.
- ***plotWarped*: This method plots the warped cross-section including a** contour criteria, typically in conjuction with the results of the displacement of a full beam model.

`__init__`(*XID*, *Airfoil*, *xdim*, *laminates*, *matlib*, ***kwargs*)

Instantiates a cross-section object.

The constructor for the class is effectively responsible for creating the 2D desretized mesh of the cross-section. It is important to note that while meshing technically occurs in the constructor, the work is handeled by another class altogether. While not computationally heavily intensive in itself, it is responsible for creating all of the framework for the cross-sectional analysis.

Args

- *XID (int)*: The cross-section integer identifier.
- ***Airfoil (obj)*: An airfoil object used to determine the OML shape of** the cross-section.
- ***xdim (1x2 array[float])*: The non-dimensional starting and stoping** points of the cross-section. In other words, if you wanted to have your cross-section start at the 1/4 chord and run to the 3/4 chord of your airfoil, xdim would look like xdim=[0.25,0.75]
- ***laminates (1xN array[obj])*: Laminate objects used to create the** descretized mesh surface. Do not repeat a laminate within this array! It will referrence this object multiple times and not mesh the cross-section properly then!
- *matlib (obj)*: A material library
- ***typeXSect (str)*: The general shape the cross-section should take.** Note that currently only a box beam profile is supported. More shapes and the ability to add stiffeners to the cross-section will come in later updates.
- ***meshSize (int)*: The maximum aspect ratio you would like your 2D** CQUADX elements to exhibit within the cross-section.

Returns

- None

`calcWarpEffects`(***kwargs*)

Calculates displacements, stresses, and strains for applied forces

The second most powerful method of the XSect class. After an analysis is run, the FEM class stores force and moment resultants within the beam element objects. From there, warping displacement, strain and stress can be determined within the cross-section at any given location within the beam using this method. This method will take a while though as it has to calculate 4 displacements and 24 stresses and strains for every element within the cross-section. Keep that in mind when you are surveying your beam or wing for displacements, stresses and strains.

Args

- ***force (6x1 np.array[float])*: This is the internal force and moment** resultant experienced by the cross-section.

Returns

- None

plotRigid(***kwargs*)

Plots the rigid cross-section along a beam.

This method is very useful for visually debugging a structural model. It will plot out the rigid cross-section in 3D space with regards to the reference axis.

Args

- ***x (1x3 np.array[float])*: The rigid location on your beam you are** trying to plot:
- ***beam_axis (1x3 np.array[float])*: The vector pointing in the** direction of your beam axis.
- *figName (str)*: The name of the figure.
- ***wireMesh (bool)*: A boolean to determine of the wiremesh outline** should be plotted.*

Returns

- *(fig)*: Plots the cross-section in a mayavi figure.

Note: Because of how the mayavi wireframe keyword works, it will

---

apear as though the cross-section is made of triangles as opposed to quadrilateras. Fear not! They are made of quads, the wireframe is just plotted as triangles.

plotWarped(***kwargs*)

Plots the warped cross-section along a beam.

Once an analysis has been completed, this method can be utilized in order to plot the results anywhere along the beam.

Args

- ***displScale (float)*: The scale by which all rotations and** displacements will be mutliplied in order make it visually easier to detect displacements.
- ***x (1x3 np.array[float])*: The rigid location on your beam you are** trying to plot:
- ***U (1x6 np.array[float])*: The rigid body displacements and rotations** experienced by the cross-section.
- ***beam_axis (1x3 np.array[float])*: The vector pointing in the** direction of your beam axis.
- ***contour (str)*: Determines what value is to be plotted during as a** contour in the cross-section.
- *figName (str)*: The name of the figure.
- ***wireMesh (bool)*: A boolean to determine of the wiremesh outline** should be plotted.*
- ***contLim (1x2 array[float])*: Describes the upper and lower bounds of** contour color scale.
- ***warpScale (float)*: The scaling factor by which all warping** displacements in the cross-section will be multiplied.

Returns

- *(fig)*: Plots the cross-section in a mayavi figure.

Note: Because of how the mayavi wireframe keyword works, it will

---

apear as though the cross-section is made of triangles as opposed to quadrilateras. Fear not! They are made of quads, the wireframe is just plotted as triangles.

printSummary(*refAxis=True*, *decimals=8*, ***kwargs*)

Print characterisic information about the cross-section.

This method prints out characteristic information about the cross- section objects. By default, the method will print out the location of the reference axis, the shear, tension, and mass center. This method if requested will also print the stiffness and mass matricies.

Args

- ***refAxis (bool)*: Boolean to determine if the stiffness matrix** printed should be about the reference axis (True) or about the local xsect origin (False).
- ***stiffMat (bool)*: Boolean to determine if the stiffness matrix** should be printed.
- ***tensCntr (bool)*: Boolean to determine if the location of the tension** center should be printed.
- ***shearCntr (bool)*: Boolean to determine if the location of the shear** center should be printed.
- ***massCntr (bool)*: Boolean to determine if the location of the mass** center should be printed.
- ***refAxisLoc (bool)*: Boolean to determine if the location of the** reference axis should be printed.

Returns

- *(str)*: Prints out a string of information about the cross-section.

resetResults()

Resets displacements, stress and strains within an xsect

This method clears all results (both warping, stress, and strain) within the elements in the xsect object.

Args

- None

Returns

- None

xSectionAnalysis(***kwargs*)

Analyzes an initialized corss-section.

This is the main workhorse of the class. This method assembles the finite element model generated using the meshing class, and solve the HIGH dimensional equilibrium equations associated with the cross- section. In doing so, it generates the warping displacement, the section strain, and the gradient of the warping displacement along the beam axis as a function of force-moment resultants. With these three things, the 3D strains->stresses can be recovered.

This method has been EXTENSIVELY tested and validated against various sources (see theory guide for more info). Since this method is so robust, the biggest limitation of the XSect class is what the mesher is capable of meshing. Finally, keep in mind that due to the high dimensionality of this problem, this method uses up a lot of resources (primarily memory). If this method is taking too many resources, choose a larger aspect ratio for your XSect initialization.

Args

- ***ref_ax (str or 1x2 array[float])*: Currently there are two supported** input types for this class. The first is the are string key-words. These are ‘shearCntr’, ‘massCntr’, and ‘origin’. Currently ‘shearCntr’ is the default value. Also suported is the ability to pass a length 2 array containing the x and y coordinates of the reference axis relative to the origin. This would take the form of: ref_ax=[1.,3.] to put the reference axis at x,y = 1.,3.

Returns

- None

5.4.9 TIMOSHENKO BEAM

class `AeroComBAT.Structures.`**`TBeam`**(*EID*, *x1*, *x2*, *xsect*, *SBID=0*, *nid1=0*, *nid2=1*, *chordVec=array([1., 0., 0.])*)

Creates a Timoshenko beam finite element object.

The primary beam finite element used by AeroComBAT, this beam element is similar to the Euler-Bernoulli beam finite element most are farmiliar with, with the exception that it has the ability to experience shear deformation in addition to just bending.

Attributes

- *type (str)*:String describing the type of beam element being used.
- ***U1 (dict)*: This dictionary contains the results of an analysis set. The** keys are the string names of the analysis and the values stored are 6x1 np.array[float] vectors containing the 3 displacements and 3 rotations at the first node.
- ***U2 (dict)*: This dictionary contains the results of an analysis set. The** keys are the string names of the analysis and the values stored are 6x1 np.array[float] vectors containing the 3 displacements and 3 rotations at the second node.
- ***Umode1 (dict)*: This dictionary contains the results of a modal analysis** set. The keys are the string names of the analysis and the values stored are 6xN np.array[float]. The columns of the array are the displacements and rotations at the first node associated with the particular mode.
- ***Umode2 (dict)*: This dictionary contains the results of a modal analysis** set. The keys are the string names of the analysis and the values stored are 6xN np.array[float]. The columns of the array are the displacements and rotations at the second node associated with the particular mode.
- ***F1 (dict)*: This dictionary contains the results of an analysis set. The** keys are the string names of the analysis and the values stored are 6x1 np.array[float] vectors containing the 3 internal forces and 3 moments at the first node.
- ***F2 (dict)*: This dictionary contains the results of an analysis set. The** keys are the string names of the analysis and the values stored are 6x1 np.array[float] vectors containing the 3 internal forces and 3 moments at the second node.
- ***Fmode1 (dict)*: This dictionary contains the results of a modal analysis** set. The keys are the string names of the analysis and the values stored are 6xN np.array[float]. The columns of the array are the forces and moments at the first node associated with the particular mode.*
- ***Fmode2 (dict)*: This dictionary contains the results of a modal analysis** set. The keys are the string names of the analysis and the values stored are 6xN np.array[float]. The columns of the array are the forces and moments at the second node associated with the particular mode.*
- ***xsect (obj)*: The cross-section object used to determine the beams** stiffnesses.
- *EID (int)*: The element ID of the beam.

- *SBID (int)*: The associated Superbeam ID the beam object belongs to.
- *n1 (obj)*: The first nodal object used by the beam.
- *n2 (obj)*: The second nodal object used by the beam.
- *Fe (12x1 np.array[float])*: The distributed force vector of the element
- *Ke (12x12 np.array[float])*: The stiffness matrix of the beam.
- ***Keg (12x12 np.array[float])*: The geometric stiffness matrix of the** beam. Used for beam buckling calculations.
- *Me (12x12 np.array[float])*: The mass matrix of the beam.
- *h (float)*: The magnitude length of the beam element.
- ***xbar (float)*: The unit vector pointing in the direction of the rigid** beam.
- *T (12x12 np.array[float])*:

Methods

- ***printSummary*: This method prints out characteristic attributes of the** beam finite element.
- *plotRigidBeam*: Plots the the shape of the rigid beam element.
- *plotDisplBeam*: Plots the deformed shape of the beam element.
- ***printInternalForce*: Prints the internal forces of the beam element for** a given analysis set

---

Note: The force and moments in the Fmode1 and Fmode2 could be completely

---

fictitious and be left as an artifact to fascilitate plotting of warped cross-sections. DO NOT rely on this information being meaningful.

`__init__`(*EID*, *x1*, *x2*, *xsect*, *SBID=0*, *nid1=0*, *nid2=1*, *chordVec=array([1.*, *0.*, *0.])*)

Instantiates a timoshenko beam element.

This method instatiates a finite element timoshenko beam element. Currently the beam must be oriented along the global y-axis, however full 3D orientation support for frames is in progress.

Args

- ***x1 (1x3 np.array[float])*: The 3D coordinates of the first beam** element node.
- ***x2 (1x3 np.array[float])*: The 3D coordinates of the second beam** element node.
- ***xsect (obj)*: The cross-section object used to determine stiffnes** and mass properties for the beam.
- *EID (int)*: The integer identifier for the beam.
- *SBID (int)*: The associated superbeam ID.
- *nid1 (int)*: The first node ID
- *nid2 (int)*: The second node ID

Returns

- None

`plotDisplBeam`(***kwargs*)

Plots the displaced beam in 3D space.

This method plots the deformed beam finite element in 3D space. It is not typically called by the beam object but by a SuperBeam object or even a WingSection object.

Args

- ***environment (str)*: Determines what environment is to be used to** plot the beam in 3D space. Currently only mayavi is supported.
- *figName (str)*: The name of the figure in which the beam will apear.
- ***clr (1x3 touple(float))*: This touple contains three floats running** from 0 to 1 in order to generate a color mayavi can plot.
- ***displScale (float)*: The scaling factor for the deformation** experienced by the beam.
- ***mode (int)*: Determines what mode to plot. By default the mode is 0** implying a non-eigenvalue solution should be plotted.

Returns

- *(fig)*: The mayavi figure of the beam.

`plotRigidBeam`(***kwargs*)

Plots the rigid beam in 3D space.

This method plots the beam finite element in 3D space. It is not typically called by the beam object but by a SuperBeam object or even a WingSection object.

Args

- ***environment (str)*: Determines what environment is to be used to** plot the beam in 3D space. Currently only mayavi is supported.
- *figName (str)*: The name of the figure in which the beam will apear.
- ***clr (1x3 touple(float))*: This touple contains three floats running** from 0 to 1 in order to generate a color mayavi can plot.

Returns

- *(fig)*: The mayavi figure of the beam.

`printInternalForce`(***kwargs*)

Prints the internal forces and moments in the beam.

For a particular analysis set, this method prints out the force and moment resultants at both nodes of the beam.

Args

- ***analysis_name (str)*: The analysis name for which the forces are** being surveyed.

Returns

- ***(str)*: This is a print out of the internal forces and moments** within the beam element.

printSummary(*decimals=8*, ***kwargs*)

Prints out characteristic information about the beam element.

This method by default prints out the EID, XID, SBID and the NIDs along with the nodes associated coordinates. Upon request, it can also print out the beam element stiffness, geometric stiffness, mass matricies and distributed force vector.

Args

- ***nodeCoord (bool)*: A boolean to determine if the node coordinate** information should also be printed.
- ***Ke (bool)*: A boolean to determine if the element stiffness matrix** should be printed.
- ***Keg (bool)*: A boolean to determine if the element gemoetric** stiffness matrix should be printed.
- ***Me (bool)*: A boolean to determine if the element mass matrix** should be printed.
- ***Fe (bool)*: A boolean to determine if the element distributed force** and moment vector should be printed.

Returns

- *(str)*: Printed summary of the requested attributes.

5.4.10 SUPER-BEAM

class AeroComBAT.Structures.**SuperBeam**(*SBID*, *x1*, *x2*, *xsect*, *noe*, *btype='Tbeam'*, *sNID=1*, *sEID=1*, ***kwargs*)

Create a superbeam object.

The superbeam object is mainly to fascilitate creating a whole series of beam objects along the same line.

Attributes

- *type (str)*: The object type, a 'SuperBeam'.
- *btype (str)*: The beam element type of the elements in the superbeam.
- *SBID (int)*: The integer identifier for the superbeam.
- *sNID (int)*: The starting NID of the superbeam.
- *enid (int)*: The ending NID of the superbeam.
- ***xsect (obj)*: The cross-section object referenced by the beam elements** in the superbeam.
- *noe (int)*: Number of elements in the beam.
- *NIDs2EIDs (dict)*: Mapping of NIDs to beam EIDs within the superbeam
- ***x1 (1x3 np.array[float])*: The 3D coordinate of the first point on the** superbeam.
- ***x2 (1x3 np.array[float])*: The 3D coordinate of the last point on the** superbeam.
- ***sEID (int)*: The integer identifier for the first beam element in the** superbeam.
- ***elems (dict)*: A dictionary of all beam elements within the superbeam.** The keys are the EIDs and the values are the corresponding beam elements.
- ***xbar (1x3 np.array[float])*: The vector pointing along the axis of the** superbeam.

Methods

- *getBeamCoord*: Returns the 3D coordinate of a point along the superbeam.
- ***printInternalForce*: Prints all internal forces and moments at every** node in the superbeam.
- ***writeDisplacements*: Writes all displacements and rotations in the** superbeam to a .csv
- ***getEIDatx*: Provided a non-dimensional point along the superbeam, this** method returns the local element EID and the non-dimensional coordinate within that element.
- ***printSummary*: Prints all of the elements and node IDs within the beam** as well as the coordinates of those nodes.

`__init__`(*SBID*, *x1*, *x2*, *xsect*, *noe*, *btype='Tbeam'*, *sNID=1*, *sEID=1*, ***kwargs*)

Creates a superelement object.

This method instantiates a superelement. What it effectively does is mesh a line provided the starting and ending points along that line. Keep in mind that for now, only beams running parallel to the z-axis are supported.

Args

- *x1 (1x3 np.array[float])*: The starting coordinate of the beam.
- *x2 (1x3 np.array[float])*: The ending coordinate of the beam.
- *xsect (obj)*: The cross-section used throught the superbeam.
- *noe (int)*: The number of elements along the beam.
- *SBID (int)*: The integer identifier for the superbeam.
- ***btype (str)*: The beam type to be meshed. Currently only Tbeam types** are supported.
- *sNID (int)*: The starting NID for the superbeam.
- *sEID (int)*: The starting EID for the superbeam.

Returns

- None

`getBeamCoord`(*x_nd*)

Determine the global coordinate along superbeam.

Provided the non-dimensional coordinate along the beam, this method returns the global coordinate at that point.

Args

- ***x_nd (float)*: The non-dimensional coordinate along the beam. Note** that x_nd must be between zero and one.

Returns

- *(1x3 np.array[float])*: The global coordinate corresponding to x_nd

`getEIDatx`(*x*)

Returns the beam EID at a non-dimensional x-location in the superbeam.

Provided the non-dimensional coordinate along the beam, this method returns the global beam element EID, as well as the local non- dimensional coordinate within the specific beam element.

Args

- *x (float)*: The non-dimensional coordinate within the super-beam

Returns

- ***EID (int)*: The EID of the element containing the non-dimensional** coordinate provided.
- ***local_x_nd (float)*: The non-dimensional coordinate within the beam** element associated with the provided non-dimensional coordinate within the beam.

`printInternalForce`(***kwargs*)

Prints the internal forces and moments in the superbeam.

For every node within the superbeam, this method will print out the internal forces and moments at those nodes.

Args

- ***analysis_name (str)*: The name of the analysis for which the forces** and moments are being surveyed.

Returns

- *(str)*: Printed output expressing all forces and moments.

`printSummary`(*decimals=8*, ***kwargs*)

Prints out characteristic information about the super beam.

This method by default prints out the EID, XID, SBID and the NIDs along with the nodes associated coordinates. Upon request, it can also print out the beam element stiffness, geometric stiffness, mass matricies and distributed force vector.

Args

- ***nodeCoord (bool)*: A boolean to determine if the node coordinate** information should also be printed.
- ***Ke (bool)*: A boolean to determine if the element stiffness matrix** should be printed.
- ***Keg (bool)*: A boolean to determine if the element gemoetric** stiffness matrix should be printed.
- ***Me (bool)*: A boolean to determine if the element mass matrix** should be printed.
- ***Fe (bool)*: A boolean to determine if the element distributed force** and moment vector should be printed.

Returns

- *(str)*: Printed summary of the requested attributes.

`writeDisplacements`(***kwargs*)

Write internal displacements and rotations to file.

For every node within the superbeam, this method will tabulate all of the displacements and rotations and then write them to a file.

Args

- *fileName (str)*: The name of the file where the data will be written.
- ***analysis_name (str)*: The name of the analysis for which the** displacements and rotations are being surveyed.

Returns

- ***fileName (file)*: This method doesn't actually return a file, rather** it writes the data to a file named "fileName" and saves it to the working directory.

writeForcesMoments(***kwargs*)

Write internal force and moments to file.

For every node within the superbeam, this method will tabulate all of the forces and moments and then write them to a file.

Args

- *fileName (str)*: The name of the file where the data will be written.
- ***analysis_name (str)*: The name of the analysis for which the** forces and moments are being surveyed.

Returns

- ***fileName (file)*: This method doesn't actually return a file, rather** it writes the data to a file named "fileName" and saves it to the working directory.

5.4.11 WING SECTION

class AeroComBAT.Structures.**WingSection**(*x1*, *x2*, *chord*, *name*, *x0_spar*, *xf_spar*, *laminates*, *matLib*, *noe*, *SSBID=0*, *SNID=0*, *SEID=0*, ***kwargs*)

Creates a wing section object.

This class instantiates a wing section object which is intended to represent the section of a wing enclosed by two ribs. This allows primarily for two different things: it allows the user to vary the cross-section design of the wing by enabling different designs in each wing section, as well as enabling the user to estimate the static stability of the laminates that make up the wing-section design.

Attributes

- ***Airfoils (Array[obj])*: This array contains all of the airfoils used** over the wing section. This attribute exists primarily to fascilitate the meshing process and is subject to change.
- ***XSects (Array[obj])*: This array contains all of the cross-section** objects used in the wing section. If the cross-section is constant along the length of the wing section, this array length is 1.
- ***SuperBeams (Array[obj])*: This array contains all of the superbeam** objects used in the wing section. If the cross-section is constant along the length of the wing section, this array length is 1.
- ***xdim (1x2 Array[float])*: This array contains the non-dimensional** starting and ending points of the wing section spar. They are non-dimensionalized by the chord length.
- ***Laminates (Array[obj])*: This array contains the laminate objects used** by the cross-sections in the wing section.
- *x1 (1x3 np.array[float])*: The starting coordinate of the wing section.
- *x2 (1x3 np.array[float])*: The ending coordinate of the wing section.

•*XIDs (Array[int])*: This array containts the integer cross-section IDs

Methods

•*plotRigid*: This method plots the rigid wing section in 3D space.

•***plotDispl*: Provided an analysis name, this method will deformed state** of the wing section. It is also capable of plotting cross-section criteria, such as displacement, stress, strain, or failure criteria.

> **Warning:** While it is possible to use multiple cross-section within the wing section, this capability is only to be utilized for tapering cross sections, not changing the cross-section type or design (such as by changing the laminates used to make the cross-sections). Doing so would invalidate the ritz method buckling solutions applied to the laminate objects.

`__init__` (*x1*, *x2*, *chord*, *name*, *x0_spar*, *xf_spar*, *laminates*, *matLib*, *noe*, *SSBID=0*, *SNID=0*, *SEID=0*, ***kwargs*)

Creates a wing section object

This wing section object is in some way an organizational object. It holds a collection of superbeam objects which in general could all use different cross-sections. One could for example use several super-beams in order to simlate a taper within a wing section descretely. These objects will also be used in order to determine the buckling span of the laminate objects held within the cross-section.

Args

•***x1 (1x3 np.array[float])*: The starting coordinate of the wing** section.

•***x2 (1x3 np.array[float])*: The ending coordinate of the wing** section.

•***chord (func)*: A function that returns the chord length along a wing** provided the scalar length from the wing origin to the desired point.

•***name (str)*: The name of the airfoil to be used to mesh the** cross-section. This is subject to change since the meshing process is only a placeholder.

•***x0_spar (float)*: The non-dimensional starting location of the cross** section. This value is non-dimensionalized by the local chord length.

•***xf_spar (float)*: The non-dimensional ending location of the cross** section. This value is non-dimensionalized by the local chord length.

•***laminates (Array[obj])*: This array contains the laminate objects to** be used in order to mesh the cross-section.

•***matLib (obj)*: This material library object contains all of the** materials to be used in meshing the cross-sections used by the wing section.

•***noe (float)*: The number of beam elements to be used in the wing per** unit length.

•*SSBID (int)*: The starting superbeam ID in the wing section.

•*SNID (int)*: The starting node ID in the wing section.

•*SEID (int)*: The starting element ID in the wing section.

•*SXID (int)*: The starting cross-section ID in the wing section.

•***numSupBeams (int)*: The number of different superbeams to be used** in the wing section.

•***typeXSect (str)*: The type of cross-section used by the wing** section.

- ***meshSize (int)*: The maximum aspect ratio an element can have within** the cross-sections used by the wing sections.
- ***ref_ax (str)*: The reference axis used by the cross-section. This is** axis about which the loads will be applied on the wing section.

Note: The chord function could take the shape of: chord = lambda y: (ctip-croot)*y/b_s+croot

---

`plotDispl`(***kwargs*)

Plots the deformed wing section object in 3D space.

Provided an analysis name, this method will plot the results from the corresponding analysis including beam/cross-section deformation, and stress, strain, or failure criteria within the sampled cross-sections.

Args

- ***figName (str)*: The name of the plot to be generated. If one is not** provided a semi-random name will be generated.
- ***environment (str)*: The name of the environment to be used when** plotting. Currently only the 'mayavi' environment is supported.
- ***clr (1x3 tuple(int))*: This tuple represents the RGB values that the** beam reference axis will be colored with.
- ***numXSects (int)*: This is the number of cross-sections that will be** plotted and evenly distributed throughout the beam.
- ***contour (str)*: The contour to be plotted on the sampled cross** sections.
- ***contLim (1x2 Array[float])*: The lower and upper limits for the** contour color plot.
- ***warpScale (float)*: The visual multiplication factor to be applied** to the cross-sectional warping displacement.
- ***displScale (float)*: The visual multiplication factor to be applied** to the beam displacements and rotations.
- ***analysis_name (str)*: The analysis name corresponding to the results** to pe visualized.
- ***mode (int)*: For modal analysis, this corresponds to the mode-shape** which is desired to be plotted.

Returns

- *(figure)*: This method returns a 3D plot of the rigid wing section.

> **Warning:** In order to limit the size of data stored in memory, the local cross-sectional data is not stored. As a result, for every additional cross-section that is plotted, the time required to plot will increase substantially.

`plotRigid`(***kwargs*)

Plots the rigid wing section object in 3D space.

This method is exceptionally helpful when building up a model and debugging it.

Args

- ***figName (str)*: The name of the plot to be generated. If one is not** provided a semi-random name will be generated.
- ***environment (str)*: The name of the environment to be used when** plotting. Currently only the 'mayavi' environment is supported.
- ***clr (1x3 tuple(int))*: This tuple represents the RGB values that the** beam reference axis will be colored with.
- ***numXSects (int)*: This is the number of cross-sections that will be** plotted and evenly distributed throughout the beam.

Returns

- *(figure)*: This method returns a 3D plot of the rigid wing section.

> **Warning:** In order to limit the size of data stored in memory, the local cross-sectional data is not stored. As a result, for every additional cross-section that is plotted, the time required to plot will increase substantially.

5.5 Aerodynamics Module

This module contains a library of classes devoted to modeling aircraft parts.

The main purpose of this library is to model various types of aircraft parts. Currently only wing objects are suported, however in the future it is possible that fuselages as well as other parts will be added.

SUMARRY OF THE METHODS

- ***K*: The kernel function used in the doublet-lattice method to relate** downwashes to panel pressures.
- ***calcAIC*: Provided several vectors of numbers as well as a reduced frequency** and mach number, this method calculates a matrix of AIC's using doublet- lattice method elementary solutions. This method is used by the FEM class flutterAnalysis method.

SUMARRY OF THE CLASSES

- ***Airfoil*: Primarily used for the generation of structural cross-sectional** meshes, this class represent an airfoil. This class could be expanded in future to use simple 2D panel methods for an airfoil of arbitrary shape.
- ***CQUADA*: This class creates quadrilateral panels intended to be used for** potential flow panel methods. Currently it is used for the unsteady doublet-lattice panels.
- *CAERO1*: This class is used to generate a lattice of CQUADA panels.

5.5.1 DOUBLET-LATTICE KERNEL FUNCTION

`AeroComBAT.Aerodynamics.K()`

Evaluates the doublet-lattice kernel function.

Provided several geometric parameters about the sending and recieving panels, this method evaluates the kernel function which relates the pressure on one panel to the downwash induced at another panel.

Args

- *Xr (1x3 np.array[float])*: The location of the recieving point.

- *Xs (1x3 np.array[float])*: The location of the sending point.
- ***gamma_r (1x3 np.array[float])*: The dihedral of the panel corresponding** to the recieving point.
- ***gamma_s (1x3 np.array[float])*: The dihedral of the panel corresponding** to the sending point.
- *M (float)*: The mach number
- *br (float)*: The reference semi-chord
- *kr (float)*: The reduced frequency
- ***r1 (float)*: The scalar distance between the sending point and the** recieving point.

Returns

- ***Kbar (complex128)*: The evaluation of the unsteady kernel function which** is complex in nature.

5.5.2 DOUBLET-LATTICE AIC METHOD

`AeroComBAT.Aerodynamics.`**`calcAIC`**`()`

Calculate the doublet-lattice AIC's.

Provided the geometry of all of the doublet-lattice panels, this method calculates the AIC matrix.

Args

- *M (float)*: The mach number.
- *kr (float)*: The reduced frequency.
- *br (float)*: The reference semi-chord.
- *delta_x_vec (1xN array[float]*: An array of chord length of the panels.
- *sweep_vec (1xN array[float])*: An array of sweep angles of the panels.
- ***l_vec (1xN array[float])*: An array of average doublet line lengths of** the panels.
- ***dihedral_vec (1xN array[float])*: An array of dihedral angles of the** panels.
- ***Xr_vec (Nx3 np.array[float])*: A matrix of recieving points, where a row** are the 3D coordinates of the point.
- ***Xi_vec (Nx3 np.array[float])*: A matrix of inboard sending points, where** a row are the 3D coordinates of the point.
- ***Xc_vec (Nx3 np.array[float])*: A matrix of center sending points, where** a row are the 3D coordinates of the point.
- ***Xo_vec (Nx3 np.array[float])*: A matrix of outboard sending points,** where a row are the 3D coordinates of the point.
- ***symxz (bool)*: A boolean operater intended to determine whether or not** a reflection of the panels should be considered over the xz-plane.

Returns

- ***D (NPANxNPAN np.array[complex128])*: The matrix which relates pressures** over panels to induced velocities over those panels. In more simple terms, this is the inverse of the desired AIC matrix.

5.5.3 AIRFOIL

class `AeroComBAT.Aerodynamics.`**`Airfoil`**(*c*, ***kwargs*)

Creates an airfoil object.

This class creates an airfoil object. Currently this class is primarily used in the generation of cross-sectional meshes. Currently only NACA 4 series arfoil and rectangular boxes are supported.

Attributes

- *c (float)*: The chord length of the airfoil.
- *t (float)*: The max percent thickness of the airfoil.
- ***p (float)*: The location of the max camber of the airfoil, in 10%** increments.
- *m (float)*: The max camber of the airfoil as a percent of the chord.

Methods

- ***points*: Generates the x and y upper and lower coordinates of the** airfoil.

`__init__`(*c*, ***kwargs*)

Airfoil object constructor.

Initializes the airfoil object.

Args

- *c (float)*: The chord length of the airfoil.
- ***name (str)*: The name of the airfoil section. This can either be** a 'NACAXXXX' airfoil or 'box' which signifies the OML is a rectangle.

Returns

- None

`points`(*x*)

Generates upper and lower airfoil curves.

This method will generate the x and y coordinates for the upper and lower airfoil surfaces provided the non-dimensional array of points x.

Args

- ***x (1xN np.array[float])*: An array of floats for which the upper and** lower airfoil curves should be generated.

Returns

- *xu (1xN np.array[float])*: The upper x-coordinates of the curve.
- *yu (1xN np.array[float])*: The upper y-coordinates of the curve.
- *xl (1xN np.array[float])*: The lower x-coordinates of the curve.
- *yl (1xN np.array[float])*: The lower y-coordinates of the curve.

printSummary(*x*)

A method for printing a summary of the airfoil object.

Prints the airfoil chord length as well as airfoil name.

Args

•None

Returns

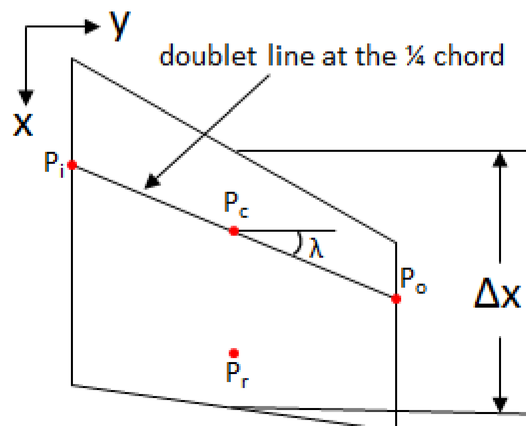
•(str): Prints the tabulated chord length and name of the airfoil

5.5.4 CQUADA

class AeroComBAT.Aerodynamics.**CQUADA**(*PANID*, *xs*)

Represents a CQUADA aerodynamic panel.

This CQUADA panel object is used for the unsteady aerodynamic doublet- lattice method currently, although it could likely easily be extended to support the vortex lattice method as well. The geometry of a generic panel can be seen in the figure below.



Attributes

•*type (str)*: The type of object.

•*PANID (int)*: The integer ID linked with the panel.

•*xs (1x4 np.array[float])*: The x coordinates of the panel.

•*ys (1x4 np.array[float])*: The y coordinates of the panel.

•*zs (1x4 np.array[float])*: The z coordinates of the panel.

•***DOF (dict[NID,factor])*: This dictionary is for connecting the movement** of the panel to the movement of an associated structure. Since a panel's control point could be between two nodes (in the middle of an element), the position of the panel can interpolated using a finite element formulation. The NID's link the movement of the panel to the movement of a corresponding node. The factor allows for a finite element interpolation.

•*Area (float)*: The area of the panel.

•*sweep (float)*: The average sweep of the panel's doublet line.

•*delta_x (float)*: The average chord line of the panel.

- *l (float)*: The length of the panel's doublet line.
- *dihedral (float)*: The dihedral of the panel.
- *Xr (1x3 np.array[float])*: The coordiantes of the panel's sending point.
- ***Xi (1x3 np.array[float])*: The coordinates of the panel's inboard** sending point.
- ***Xc (1x3 np.array[float])*: The coordinates of the panel's center** sending point.
- ***Xo (1x3 np.array[float])*: The coordinates of the panel's outboard** sending point.

Methods

- ***x*: Provided the non-dimensional coordinates eta and xi which go from -1** to 1, this method returns corresponding the x coordinates.
- ***y*: Provided the non-dimensional coordinates eta and xi which go from -1** to 1, this method returns corresponding the y coordinates.
- ***z*: Provided the non-dimensional coordinates eta and xi which go from -1** to 1, this method returns corresponding the z coordinates.
- ***J*:Provided the non-dimensional coordinates eta and xi which go from -1** to 1, this method returns the jacobian matrix at that point. This method is primarily used to fascilitate the calculation of the panels area.
- *printSummary*: Prints a summary of the panel.

Note: The ordering of the xs, ys, and zs arrays should be ordered in a

---

finite element convention. The first point refers to the root trailing edge point, followed by the tip trailling edge, then the tip leading edge, then root leading edge.

J (*eta*, *xi*)

Calculates the jacobian at a point in the element.

This method calculates the jacobian at a local point within the panel provided the master coordinates eta and xi.

Args

- *eta (float)*: The eta coordinate in the master coordinate domain.*
- *xi (float)*: The xi coordinate in the master coordinate domain.*

Returns

- ***Jmat (3x3 np.array[float])*: The stress-resutlant transformation** array.

Note: Xi and eta can both vary between -1 and 1 respectively.

__init__ (*PANID*, *xs*)

Initializes the panel.

This method initializes the panel, including generating many of the geometric properties required for the doublet lattice method such as Xr, Xi, etc.

Args

- *PANID (int)*: The integer ID associated with the panel.
- ***xs (1x4 array[1x3 np.array[float]])*: The coordinates of the four** corner points of the elements.

Returns

- None

printSummary()

A method for printing a summary of the CQUADA panel.

Prints out a tabulated information about the panel such as it's panel ID, and the coordinates of it's four corner points.

Args

- None

Returns

- *summary (str)*: The summary of the CQUADA attributes.

x(*eta*, *xi*)

Calculate the x-coordinate within the panel.

Calculates the x-coordinate on the panel provided the desired master coordinates eta and xi.

Args

- *eta (float)*: The eta coordinate in the master coordinate domain.*
- *xi (float)*: The xi coordinate in the master coordinate domain.*

Returns

- *x (float)*: The x-coordinate within the element.

Note: Xi and eta can both vary between -1 and 1 respectively.

---

y(*eta*, *xi*)

Calculate the y-coordinate within the panel.

Calculates the y-coordinate on the panel provided the desired master coordinates eta and xi.

Args

- *eta (float)*: The eta coordinate in the master coordinate domain.*
- *xi (float)*: The xi coordinate in the master coordinate domain.*

Returns

- *y (float)*: The y-coordinate within the element.

Note: Xi and eta can both vary between -1 and 1 respectively.

---

z (*eta*, *xi*)

Calculate the z-coordinate within the panel.

Calculates the z-coordinate on the panel provided the desired master coordinates eta and xi.

Args

- *eta (float)*: The eta coordinate in the master coordinate domain.*
- *xi (float)*: The xi coordinate in the master coordinate domain.*

Returns

- *z (float)*: The z-coordinate within the element.

Note: Xi and eta can both vary between -1 and 1 respectively.

5.5.5 CAERO1

class `AeroComBAT.Aerodynamics.`**`CAERO1`**(*SID*, *x1*, *x2*, *x3*, *x4*, *nspan*, *nchord*, ***kwargs*)

Represents an aerodynamic surface.

This CAERO1 object represents an aerodynamic lifting surface to be modeled using the doublet-lattice method.

Attributes

- *type (str)*: The type of object.
- *SID (int)*: The integer ID linked with the surface.
- *xs (1x4 np.array[float])*: The x coordinates of the panel.
- *ys (1x4 np.array[float])*: The y coordinates of the panel.
- *zs (1x4 np.array[float])*: The z coordinates of the panel.
- ***mesh ((NPAN+1)x(NPAN+1) np.array[int])*: The panel ID's in the relative** positions of their corresponding panels.
- ***xmesh ((NPAN+1)x(NPAN+1) np.array[float])*: The x-coordinates of the** lifting surface nodes.
- ***ymesh ((NPAN+1)x(NPAN+1) np.array[float])*: The y-coordinates of the** lifting surface nodes.
- ***zmesh ((NPAN+1)x(NPAN+1) np.array[float])*: The z-coordinates of the** lifting surface nodes.
- ***CQUADAs (dict[PANID, CQUADA])*: A dictionary mapping panel ID's to** CQUADA panel objects.

Methods

- ***x*: Provided the non-dimensional coordinates eta and xi which go from -1** to 1, this method returns corresponding the x coordinates.
- ***y*: Provided the non-dimensional coordinates eta and xi which go from -1** to 1, this method returns corresponding the y coordinates.
- ***z*: Provided the non-dimensional coordinates eta and xi which go from -1** to 1, this method returns corresponding the z coordinates.
- ***plotLiftingSurface*: Plots the lifting surface in 3D space. Useful for** debugging purposes.
- *printSummary*: Prints a summary of the panel.

Note: The ordering of the xs, ys, and zs arrays should be ordered in a

finite element convention. The first point refers to the root leading edge point, followed by the root trailling edge, then the tip trailing edge, then tip leading edge.

`__init__`(*SID*, *x1*, *x2*, *x3*, *x4*, *nspan*, *nchord*, ***kwargs*)
Constructor for the CAERO1 lifting surface object.

Provided several geometric parameters, this method initializes and discretizes a lifting surface using CQUADA panel objects.

Args

- *SID (int)*: The integer ID for the surface.
- *x1 (1x3 np.array[float])*: The coordinate of the root leading edge.
- *x2 (1x3 np.array[float])*: The coordinate of the root trailing edge.
- *x3 (1x3 np.array[float])*: The coordinate of the tip trailing edge.
- *x4 (1x3 np.array[float])*: The coordinate of the tip leading edge.
- *nspan (int)*: The number of panels to run in the spanwise direction.
- ***nchord (int)*: The number of panels to run in the chordwise** direction.

Returns

- None

`plotLiftingSurface`(***kwargs*)
Plots the lifting surface using the MayaVi environment.

This method plots the lifting surface using the MayaVi engine. It is most useful for debugging models, allowing the user to verify that the wing they thought they generated is actually what was generated.

Args

- *figName (str)*: The name of the figure

Returns

- *(figure)*: MayaVi Figure of the laminate.

`printSummary`()
A method for printing a summary of the CAERO1 element.

Prints out the surface ID, as well as the number of chordwise and spanwise panels.

Args

- None

Returns

- *summary (str)*: A summary of the CAERO1 surface attributes.

x (*eta*, *xi*)

Calculate the x-coordinate within the lifting surface.

Calculates the x-coordinate within the lifting surface provided the desired master coordinates eta and xi.

Args

- *eta (float)*: The eta coordinate in the master coordinate domain.*
- *xi (float)*: The xi coordinate in the master coordinate domain.*

Returns

- *x (float)*: The x-coordinate within the element.

Note: Xi and eta can both vary between -1 and 1 respectively.

---

y (*eta*, *xi*)

Calculate the y-coordinate within the lifting surface.

Calculates the y-coordinate within the lifting surface provided the desired master coordinates eta and xi.

Args

- *eta (float)*: The eta coordinate in the master coordinate domain.*
- *xi (float)*: The xi coordinate in the master coordinate domain.*

Returns

- *y (float)*: The y-coordinate within the element.

Note: Xi and eta can both vary between -1 and 1 respectively.

---

z (*eta*, *xi*)

Calculate the z-coordinate within the lifting surface.

Calculates the z-coordinate within the lifting surface provided the desired master coordinates eta and xi.

Args

- *eta (float)*: The eta coordinate in the master coordinate domain.*
- *xi (float)*: The xi coordinate in the master coordinate domain.*

Returns

- *z (float)*: The y-coordinate within the element.

Note: Xi and eta can both vary between -1 and 1 respectively.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

AeroComBAT.Aerodynamics,
AeroComBAT.AircraftParts,
AeroComBAT.FEM,
AeroComBAT.Structures,