Development of an Aerostructural Analysis Tool for a Low-Sweep Parametric Wing

Julian Bardin

500758483

Presented to Dr. Joon Chung

Ryerson University

April 16, 2021

Acknowledgements

The author would like to thank Dr. Joon Chung for providing the opportunity and support to develop this project. The author would also like to thank Pratik Pradhan and Mohsen Rostami for their regular guidance throughout the development of this analysis. The author would lastly like to thank Ryerson University for the chance to create an undergraduate thesis.

Abstract

An aerostructural analysis program was developed to predict the aerodynamic performance of a non-rigid, low-sweep wing. The wing planform was geometrically defined to have a rectangular section, and a trapezoidal section. The cross-section was further set to an airfoil shape which was consistent across the entire wingspan. Furthermore, to enable the inclusion of this multidisciplinary analysis module into an optimization scheme, the wing geometry was defined by a series of parameters: root chord, taper ratio, leading-edge sweep, semi-span length, and the kink location.

Aerodynamic analysis was implemented through the quasi-three-dimensional approach, including a three-dimensional inviscid solution and a sectional two-dimensional viscous solution. The inviscid analysis was provided through the implementation of the vortex ring lifting surface method, which modelled the wing about its mean camber surface.

The viscous aerodynamic solution was implemented through a sectional slicing of the wing. For each section, the effective angle of attack was determined and provided as an input to a two-dimensional airfoil solver. This airfoil solution was comprised of two subcomponents: a linear-strength vortex method inviscid solution, and a direct-method viscous boundary layer computation. The converged airfoil solution was developed by adjusting the effective airfoil geometry to account for the boundary layer displacement thickness, which in itself required the inviscid tangential speeds to compute.

The structural solution was implemented through classical beam theory, with a torsion and bending calculator included. The torque and bending moment distribution along the wing were computed from the lift distribution, neglecting the effects of drag, and used to compute the twist and deflection of the wing.

Interdisciplinary coupling was achieved through an iterative scheme. With the developed implementation, the inviscid lift loads were used to compute the deformation of the wing. This deformation was used to update the wing mesh, and the inviscid analysis was run again. This iteration was continued until the lift variation between computations was below 0.1%. Once the solution was converged upon by the inviscid and structural solutions, the viscous calculator was run to develop the parasitic drag forces. Once computation had completed, the aerodynamic lift and drag forces were output to mark the completion of execution.

Table of Contents

Nomenclature

Acronyms

2D	Two-Dimensional
3D	Three-Dimensional
CPU	Central Processing Unit
MATLAB	Matrix Laboratory
MDA	Multidisciplinary Analysis
MDF	Multidisciplinary Feasible
MDO	Multidisciplinary Optimization
Q3D	Quasi-Three-Dimensional
RAM	Random Access Memory
RHS	Right-Hand-Side, Inviscid Free-Stream Contribution
UTH	Speed-Theta-Shape Factor, Boundary Layer Analysis
VLM	Vortex Lattice Method

Symbols

A	Area
α	Angle of Attack
α_i	Panel Incidence Angle
a_{KL}	3D Aerodynamic Influence Coefficient
b_{KL}	3D Parallel Aerodynamic Influence Coefficient
C_{di}	Induced Drag Coefficient
C_f	Friction Coefficient
C_l	Lift Coefficient
C_{l0}	Zero Angle-of-Attack Lift Coefficient
C_P	Pressure Coefficient
c	Chord Length
ΔL	Panel Lift Force
ΔD	Panel Drag Force
D_i	Induced Drag
D_p	Parasitic Drag
d_i	Section Induced Drag
d_p	Section Parasitic Drag
δ^*	Boundary Layer Displacement Thickness
ε	Strain
Γ	3D Vorticity

γ	2D Vorticity
H	Shape Factor
I_{xx},I_{zz}	Moment of Inertia
I_{zx}	Product of Inertia
L	Lift Force
l	Section Lift
λ	Pressure-Gradient Parameter
M	Bending Moment
M_∞	Free-Stream Mach Number
ϕ	Twist Angle
p	Shear Flow Lever Arm
$\vec{Q}\infty$	Free Stream Velocity Vector
q	Induced Velocity
q_s	Shear Flow
ρ	Air Density
$\vec{r}$	Position Vector
Re	Reynolds Number
S	Shear Force
s	Curve Path Distance
θ	Deflection Angle or Boundary Layer Momentum Thickness
θ_p	Principal Axis Angle
T	Torque
t	Thickness
u	Longitudinal Velocity Component
u_e	Inviscid Edge Velocity
V	Shear Force
V_∞	Free-Stream Speed
w	Vertical Velocity Component
w_{ind}	Induced Downwash
x	Longitudinal Direction, Parallel to Aircraft Axes
y	Lateral Direction, Parallel to Aircraft Axes
z	Vertical Direction, Parallel to Aircraft Axes

List of Figures

List of Tables

1 Introduction

Optimization of aircraft components can be achieved through either a series of single-disciplinary optimization stages, or through a coupled multidisciplinary approach [1]. In pursuing the multidisciplinary approach, designs exhibiting better performance become possible due to an inclusion of the coupled interactions in the analysis [1]. One implementation of this multidisciplinary strategy is the multidisciplinary feasible (MDF) architecture, wherein a multidisciplinary analyser (MDA) is run each time a constraint or objective function is computed [1]. As a precursor to this form of multidisciplinary optimization, an MDA implementation was required.

With a specific focus given to the aerostructural performance of an aircraft wing, a multidisciplinary analysis implementation was developed to predict the resulting aerodynamic performance of a wing when structural deformation was considered. Building off the work of FEMWET by Elham and van Tooren [2], this implementation sought to develop a simplified, thereby faster to compute, aerostructural modelling approach for low-sweep wings. This was comprised of a quasi-three-dimensional (Q3D) aerodynamic solver coupled with a classical bending-torsional structure solver.

The Q3D aerodynamic solver was built utilizing a Vortex Lattice Method (VLM) solver, based on the analysis developed by Katz and Plotkin [3], used to compute the inviscid aerodynamic effects. Viscous effects were subsequently computed using aerodynamic strip theory to divide the wing into a series of sections which were individually analysed [2]. Along each strip, the effective angle of attack was computed from the induced drag, and the acting parasitic drag force was computed iteratively between an inviscid two-dimensional airfoil panel solver and a direct boundary layer approximation. In this implementation, a linear-strength vortex panel method was used to predict the tangential speed distribution along the airfoil surface. This was developed according to the analysis developed by Katz and Plotkin [3]. Using the speed distribution, the resulting boundary layer was computed along the length of the upper and lower airfoil surfaces according to the method presented by Fujiwara et al [4]. The airfoil geometry was adjusted by the boundary layer results, and the converged solution was taken as the parasitic drag acting on the section. The spanwise summation of this was used as the acting parasitic drag for the wing.

The structural solver itself was developed using classical beam theory as presented by Hibbeler [5]. Using this, an implementation was developed where the wing twist about the shear centre was computed, and then the bending deflection of the wing was determined. The combined rotations and deflections along the wing were used to deform the aerodynamic model and adjust the acting loads. For rapid computation, the wing box structure was analysed as a series of area concentrations laid about the wing box frame.

The global aerostructural solution was determined through the coupling of the aerodynamic and structural solvers. To achieve this, the structural effects of the drag was considered negligible and omitted from deforming the wing. An iterative scheme was then employed to apply the loads from the VLM to the structure, which then deformed to adjust the VLM mesh. This process was repeated until the solution had converged, after which the viscous effects were computed, and the multidisciplinary analysis was considered to be complete.

2 Analysis Architecture

2.1 Flow of Information

The chosen architecture for the multidisciplinary analysis began first by creating a top-level main function from which the individual analysers were called to produce their respective results. To this end, an encapsulated analysis function was created for each solver, which contained calls for the various sub-functions required to generate the aerodynamic or structural outputs. Additionally, to enable separate iteration schemes, the inviscid aerodynamic solver and the viscous aerodynamic solver were encapsulated separately.

The general flow of information from the main function to and from the encapsulated analysers was presented in Figure 2.1. Inputs to a function have been placed in the same column, while outputs have been placed within the same row [1]. Arrows have also been included to highlight the direction of information.

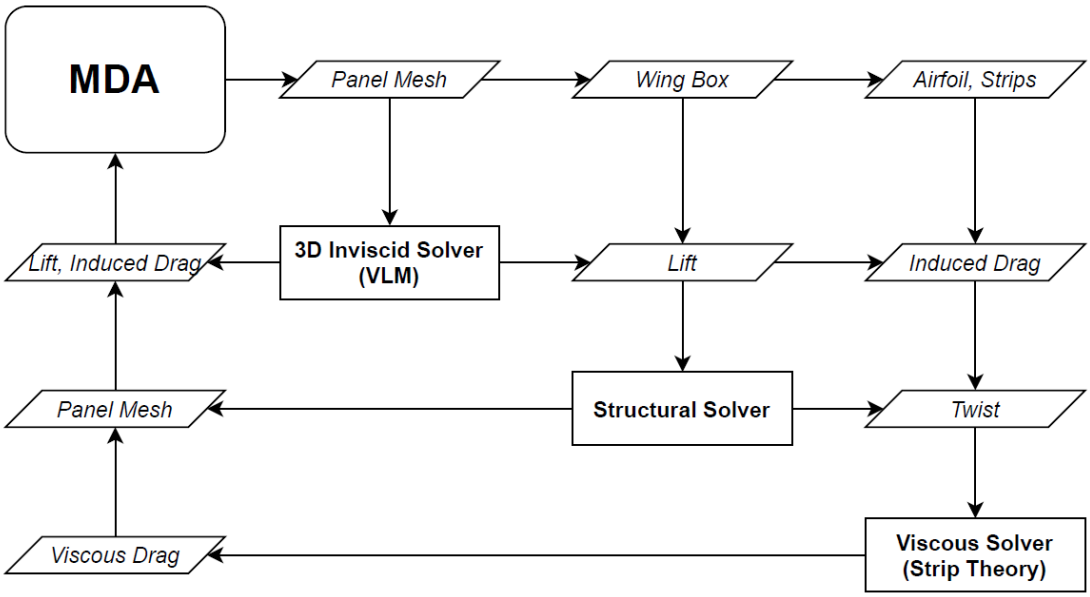


Figure 2.1: Global MDA Architecture

As was shown in the figure, geometric and material properties were sent by the main function into the three solvers. Receiving this information, the solvers would compute their respective outputs. Starting with the inviscid aerodynamic solver, the lift and induced drag distributions along the wing were output. In terms of encapsulation, this was the simplest of the three functions containing only the function calls to populate and solve the aerodynamic influence matrices.

The next solver that was called by the main function was the structural deformation calculator. This did not require any additional internal control logic, and the subfunctions were merely computed in sequence. Internally, this calculator was comprised of two modules: the wing twist solver and the wing deflection solver. First, the torque acting on each section of the wing was used to compute the twist angle at each spanwise location. From there, the bending analysis would determine the deflection through analysing the bending moment at each section and using the twist angle to adjust the location probed for the radius of curvature.

A third module was categorized within the structural solver but was present within the main analysis function rather than the encapsulated structural solver. This module was the function used to deform the VLM mesh according to the determined bending and twist distribution. This was placed outside of the structural solver to allow for the analysis program to limit the relative change in deformation per iteration.

A visual representation of the structural analysis was presented in Figure 2.2, illustrating the flow of loading, structural, and deformation information through the three sub-modules which comprised this domain.

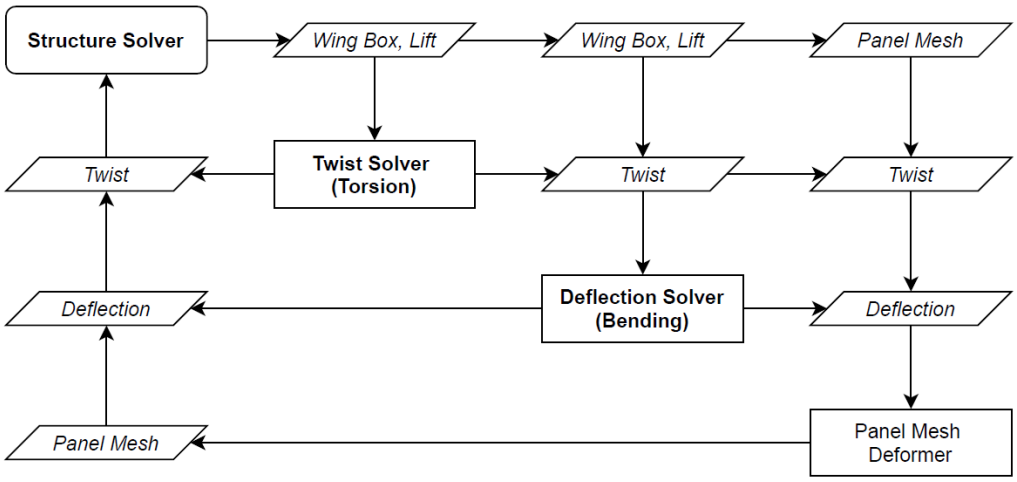


Figure 2.2: Structural Solver Architecture

The last of the three solvers to execute was the viscous drag aerodynamic analysis. Unlike the inviscid aerodynamic and structural analysers, this function contained additional control logic with its own iteration scheme. Four sub-modules comprised this analyser: an effective angle of attack calculator, a 2D inviscid airfoil analyser, a boundary layer solver, and an airfoil geometry manipulator. These modules have been presented in Figure 2.3.

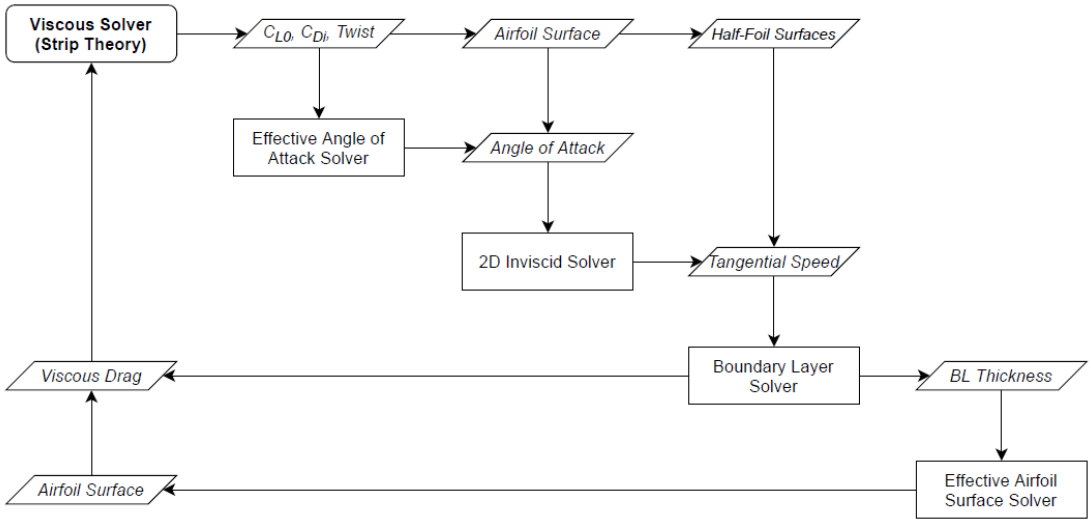


Figure 2.3: Viscous Aerodynamic Solver Architecture

In using this solver, the wing was divided into a series of strips based on the panel discretization developed for the VLM mesh. For each strip, the effective angle of attack was computed from the induced drag coefficient and a linear-lift approximation. With this parameter determined, an iterative viscous-inviscid coupled 2D aerodynamic analysis was run. With this, the inviscid airfoil panel solver was run to produce a tangential speed distribution along the airfoil surface. This was then used to compute the boundary layer characteristics, including its displacement thickness. The airfoil surface geometry sent to the inviscid calculator was then adjusted with this displacement thickness, and the analysis was run repeatedly until a solution was converged upon [3]. With the solution converged, the sectional viscous drag was returned. This process was repeated for each spanwise section, and the summation of these drag forces was taken as the total parasitic drag acting on the wing.

2.2 Lift-Structural Coupling

To simplify the analysis, coupling of the aerodynamic loads with the structural analysis was limited to only the lift loads [2], with the drag loads assumed to have a negligible impact on the wing deformation. With this assumption, the VLM was coupled to the structural solver and the viscous drag calculator was omitted from this stage of the computation. Additionally, while the induced drag was computed by the VLM, its effects too were omitted from structural consideration.

With the coupling mode defined, the iterative scheme was defined with the following steps:

1. Compute the lift force distribution along the wing with the VLM.
2. Compute the deflection and twist along the spanwise direction with the structural solver.
3. Scale the change in twist and deflection for the given iteration (prevented divergent solution)
4. Calculate the percentage change in lift for the given iteration
5. Check if change in lift is below 0.1%. If so, exit the iteration.
6. Otherwise, deform the wing VLM mesh and repeat from Step 1.

The iterative approach was necessitated by the interdependencies present between the aerodynamic and structural solutions. Essentially, the structural solver required the VLM loads as an input while the VLM required the deformed structure as an input. Through the implemented approach, the two solvers were run repeatedly until a final convergent solution was output.

2.3 Aerodynamic Coupling

With the two aerodynamic solvers, the flow of information was established to be one-directional. With this implementation, the viscous analysis required the induced drag output by the VLM, but the VLM did not require any input from the viscous solver. Following from this, the viscous analyser was only run a single time for a wing analysis, which was done after the lift-structural coupled solution had been computed.

3 Geometric and Simulation Definitions

To enable the usage of the tool, the wing was defined through a set of parameters. These parameters specified the wing planform, the airfoil geometry, and the wing box characteristics. Additional inputs were given to the analysis which defined the airflow properties.

Furthermore, to retain consistent nomenclature with aircraft, the chordwise direction was given the 'x' label, the spanwise direction was given the 'y' label, and the vertical direction was given the 'z' label. The geometric origin for analysis was placed at the leading edge of the wing root chord.

3.1 Planform Geometry

The planform of the wing was defined as having a rectangular section, and a trapezoidal section. The spanwise location where the rectangular section ended was subsequently defined as the "kink" [2]. For the overall wing geometry, the root chord length and semi-span length were used as defining parameters. The location of the kink was defined as a proportion of the semi-span, through the use of a kink distance ratio. For the trapezoidal section, the leading-edge sweep angle was also defined, and the tip chord was computed as the product of the taper ratio and the root chord length. A sample planform with these 5 parameters overlaid was presented in Figure 3.1.

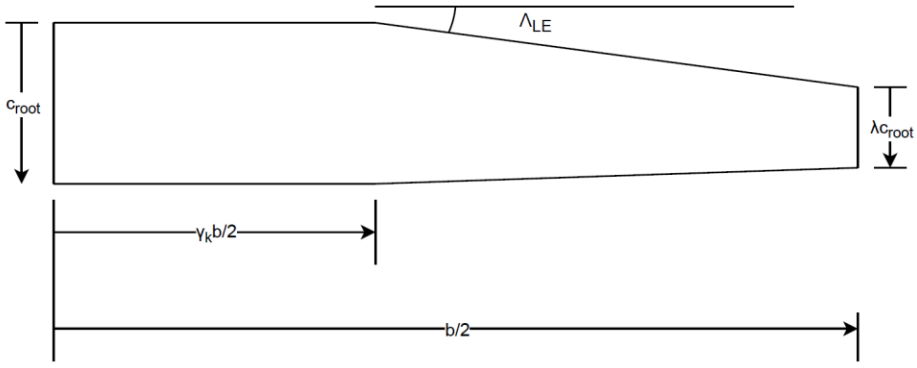


Figure 3.1: Wing Planform Parameters

3.2 Airfoil Geometry

The airfoil geometry was defined as a list of points with x and z coordinates [6]. The first point in the array was located at the trailing edge. The subsequent data points defined the curve along the upper surface, then the leading edge, and finally the lower surface [7]. The last point specified in this array was located at the lower surface trailing edge. The geometry of a unit-length airfoil was then defined as the linear interpolation, from the upper trailing edge to the lower trailing edge, of these points. An example of this representation was presented in Figure 3.2, using a NACA 6409 airfoil [6].

Figure 3.2: Sample Airfoil Geometry

3.3 Wing Box Geometry

The wing box was defined both geometrically and materially. Geometrically, the wing box shape was defined with the airfoil geometry and the placement of the forward and rear spars. The upper and lower surfaces of the wing box were interpolated from the airfoil geometry. The number of longitudinal stiffeners was also included as a wing box defining parameter. The thicknesses of the skin and spars was additionally specified, as well as the cross-sectional area of the stiffeners. For the structural modelling, the elastic modulus and shear modulus were also specified. An example of the geometry resulting from a wing box with spars placed at 20% and 70% chord length was presented in Figure 3.3. In this figure, longitudinal stiffeners were represented with dots placed along the skin.

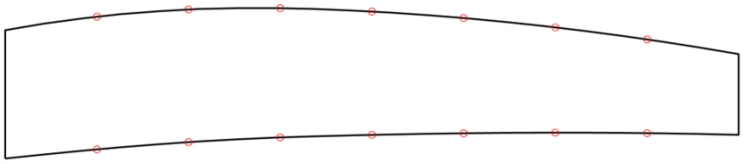


Figure 3.3: Sample Wing Box Geometry

3.4 Airflow Properties

A series of additional parameters were defined for the free-stream air properties, which were required to compute the forces acting on the wing. The free-stream true airspeed was defined, as well as the corresponding Mach number. The density and viscosity of air were also specified for the simulation. Finally, and angle of attack was defined.

4 Inviscid Aerodynamic Analysis

The inviscid aerodynamic analysis, used to compute the lift and induced drag distribution of the wing, was implemented through the use of a lifting surface solution comprised of vortex ring elements as presented by Katz and Plotkin [2]. This approximation involved modelling the wing as its mean camber surface, imposing discrete panels and vortex rings along it. The solution was then developed and solved using matrix operations [2].

4.1 Surface Discretization

In implementing the lifting surface method outlined by Katz and Plotkin [3], the mean camber surface, defined as the surface connecting the mean camber lines of the wing, was approximated by a series of thin panels. Each of these panels was trapezoidal in geometry, with two sides parallel to the free-stream air. A planform view of these trapezoidal panels was provided in Figure 4.1.

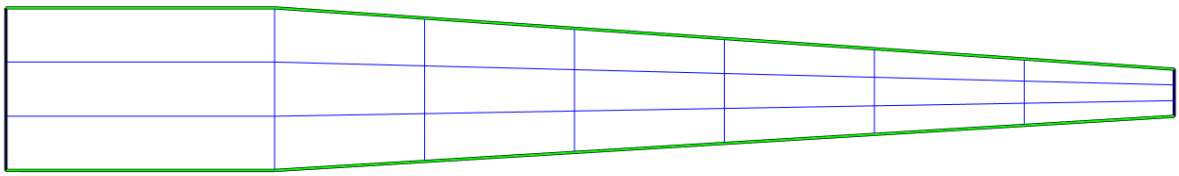


Figure 4.1: Wing Panels Planform View

As was apparent in Figure 4.1, the panels were arranged to produce a deformed grid. This grid separated the camber surface into a series of sections in the y-direction, with each section being further divided by into a set number of individual panels in the x-direction.

The z-direction coordinates for the panel vertexes were then computed from the mean camber line at each section. To produce a smooth solution, the airfoil geometry was first divided into its upper and lower surfaces. Then, each surface was interpolated from using MATLAB's modified Akima cubic interpolation, chosen as it prevented waving at the leading and trailing edge while still enabling a smooth solution. The mean z-coordinate between the upper and lower surface curves was taken to lie on the mean camber line and was used for the panel coordinates.

4.2 Analysis Components

Following with the requirements of the vortex ring analysis [3], a corresponding mesh of vortex rings was computed. These were calculated from the panels rather than the mean camber surface and were offset rearward by one-quarter panel length [3] to comply with the two-dimensional Kutta condition. The z-direction coordinate for the vortex ring sections behind the wing, caused by the offset, was set to 0.

An additional row of vortex rings was added to the rear of the mesh in order to model the trailing edge wing wake [3]. The rearmost line of these panels was given an x-direction coordinate of 1 000 000, considered to be an adequate substitute for infinity.

Additional colocation points were placed on the midpoint of each panel three-quarter chord line [3]. In terms of planform coordinates, while these were placed on the panels, the x- and y-coordinates aligned with the centers of the vortex rings. The z-coordinate, however, was computed from the panels. These colocation points served as the locations where the vorticities and loads were computed. A unit-length vector normal to each panel surface was placed at these colocation points.

A visual example of the discretized wing mesh was provided with Figure 4.2.

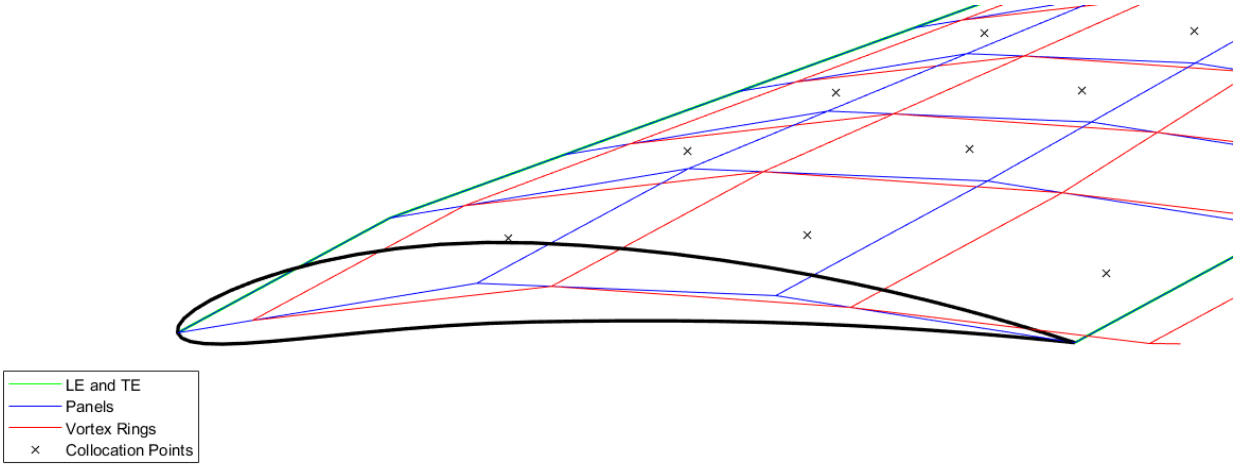


Figure 4.2: Wing Discretization

4.3 Mathematical Solution

With the geometric locations of all simulation components determined, the mathematical definitions were implemented. Globally, the vorticity solution was defined in matrix form with the following equation [3]:

$$\begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} \end{bmatrix} \begin{bmatrix} \Gamma_1 \\ \vdots \\ \Gamma_m \end{bmatrix} = \begin{bmatrix} RHS_1 \\ \vdots \\ RHS_m \end{bmatrix} \tag{4.1}$$

Where a was the aerodynamic influence coefficient, Γ was the vorticity of the vortex ring, and RHS was the free-stream contribution. Regarding the influence matrix, each row represented the influences acting on a specific colocation point, while the columns referred to the contribution of each colocation to the others. Each coefficient within the aerodynamic influence matrix was computed with the following equation:

$$a_{KL} = (q_L + q'_L + q_W + q'_W) \cdot \hat{n}_K \tag{4.2}$$

Where a_{KL} was the influence coefficient for the Kth colocation point and Lth vortex ring, q_L was the induced velocity at this colocation point caused by the Lth vortex ring, q'_L was the induced velocity by the mirrored vortex ring (as this analysis modeled the right wing, the mirror accounted for the left wing), $q_W$was the induced velocity of the wake, and q'_W was the induced velocity of the mirrored wake. For non-trailing edge panels, the induced velocity of the wake panels was zero. For all induced velocity computations, a unit vorticity was used. The actual vorticity values were solved with the matrix representation provided in (4.1).

The induced velocity of a vortex ring was computed as the sum of the influences of the individual vortex lines. As each vortex ring was trapezoidal, four vortex lines were summed. The effects of a single vortex line were computed with the following formula [3]:

$$q = \frac{\Gamma}{4\pi|\vec{r_1}\times\vec{r_2}|^2}\left(\frac{\vec{r_0}\cdot\vec{r_1}}{|\vec{r_1}|} - \frac{\vec{r_0}\cdot\vec{r_2}}{|\vec{r_2}|}\right) \cdot (\vec{r_1} \times \vec{r_2}) \tag{4.3}$$

Where $\vec{r_1}$ was the position vector from the first vortex line point and the colocation point, $\vec{r_2}$ was the position vector from the second vortex line point and the colocation point, and $\vec{r_0}$ was position vector for the vortex line.

Entries within the free-stream contribution matrix were subsequently computed with the following formula [3]:

$$RHS_K = -\vec{Q}_\infty \cdot \hat{n}_K \tag{4.4}$$

Where $\vec{Q}_\infty$ was the velocity vector of the free-stream air, accounting for the angle of attack, and $\hat{n}_K$ was the unit-length normal vector of each panel. With the aerodynamic influence coefficient matrix and the free-stream contribution vector determined, the vorticity of each vortex ring was computed numerically. This was represented in matrix form with the following formula:

$$[\Gamma] = [A]^{-1}[RHS] \tag{4.5}$$

Where $[A]$ was the aerodynamic influence coefficient matrix.

With the ring vorticities computed, sufficient information had been determined to compute the lift force acting on each panel. Determination of the induced drag, however, required the induced downwash at each colocation points. This was computed according to the following relation [3]:

$$\begin{bmatrix} w_{ind,1} \\ \vdots \\ w_{ind,m} \end{bmatrix} = \begin{bmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mm} \end{bmatrix} \begin{bmatrix} \Gamma_1 \\ \vdots \\ \Gamma_m \end{bmatrix} \tag{4.6}$$

Where w_{ind} was the induced downwash at a colocation point, and b was the component of the aerodynamic influence coefficient determined using only the vortex lines parallel to the free-stream flow.

4.4 Force Distribution

The lift and drag force acting on each panel was computed using the vorticity and induced downwash at each colocation point, as well as the free-stream air density and velocity. The formula used for lift computation, which was an implementation of the Kutta-Joukowski theorem, was as follows [3]:

$$\Delta L_{ij} = \begin{cases} \rho Q_\infty \Gamma_{i,j} \Delta y_{ij} & , \ i = 1 \\ \rho Q_\infty \left(\Gamma_{i,j} - \Gamma_{i-1,j} \right) & , \ i > 1 \end{cases} \tag{4.7}$$

Where ρ was the density in air, Q_∞ was the free stream airspeed, Δy was the panel width, and the variables i and j were indices. The induced drag was subsequently computed with the following relation [3]:

$$\Delta D_{ij} = \begin{cases} -\rho w_{ind,ij} \Gamma_{i,j} \Delta y_{ij} & , \ i = 1 \\ -\rho w_{ind,ij} \left(\Gamma_{i,j} - \Gamma_{i-1,j} \right) & , \ i > 1 \end{cases} \tag{4.8}$$

As the vortex ring analysis was an incompressible analysis, the lift and induced drag were corrected to account for compressible effects. This was done with the Prandtl-Glauert correction, represented with following formulae [3]:

$$\Delta L'_{ij} = \frac{\Delta L_{ij}}{\sqrt{1 - M_\infty^2}} \tag{4.9a}$$

$$\Delta D'_{ij} = \frac{\Delta D_{ij}}{\sqrt{1 - M_\infty^2}} \tag{4.9b}$$

Where $\Delta L'_{ij}$ was the lift force acting on a panel in compressible flow, $\Delta D'_{ij}$ was the induced drag force acting on a panel in compressible flow, and M_∞ was the free-stream Mach number. As a final computation for this stage of the analysis, the total lift and induced drag acting on the wing was computed as the sum of the forces acting on the panels, as shown with the following equations [3]:

$$L = \sum \Delta \mathrm{L'_{ij}} \tag{4.10a}$$

$$D_i = \sum \Delta D'_{ij} \tag{4.10b}$$

Where L was the total lift force and D_i was the total induced drag.

5 Viscous Aerodynamic Analysis

The viscous aerodynamic analysis of the wing was done through the use of strip theory, where the wing was approximated as a series of spanwise sections, for which 2D effects could be readily modelled [2]. The analysis for each cross section was an iterative viscous-inviscid coupled approach, where an inviscid 2D potential flow analysis was used in conjunction with an empirical viscous drag analysis to converge on a value for parasitic drag.

5.1 Strip Discretization

To align with VLM outputs, the wing was cut into a series of strips such that the ends of each strip aligned with the panel mesh boundaries parallel to the free-stream velocity. This was represented in Figure 4.1, where vertical lines of panel boundaries were present, and served as the basis for the sectional analysis. As each mesh section was trapezoidal rather than rectangular, the average chord length for the strip was used.

The inviscid forces, determined through the VLM analysis, acting on this section were used to compute the force coefficients used with the 2D analysis. Specifically, the induced drag coefficient was required for each section to compute the effective angle of attack. This was computed with the following formulae:

$$d_i = \frac{\sqrt{1-M_\infty^2}}{w}\sum \Delta D_i' \tag{5.1a}$$

$$C_{di} = \frac{2d_i}{\rho V_\infty^2 c} \tag{5.1b}$$

Where d was the induced drag force per unit length, w was the width of the section, ΔD_i was the induced drag acting on each panel within this section, C_{di} was the 2D induced drag coefficient, ρ was the density of air, and c was the average chord length of the section. The compressibility correction was also removed from the induced drag force computed by the VLM, as the analysis for determining the effective angle of attack required incompressible flow.

5.2 Effective Angle of Attack Computation

As a precursor to the inviscid analysis, the effective angle of attack acting on the airfoil was required. This was done to account for the induced drag effects acting on the section, where the downwash produced an effective tilting of the free-stream velocity vector. This, in turn, rotated and altered the acting lift force, allowing a component of the vector to act in the drag direction. The various relevant vectors pertaining to this stage of the analysis were visualized in Figure 5.1.

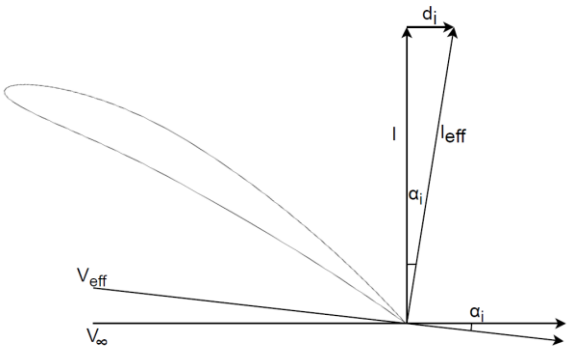


Figure 5.1: Induced Drag 2D Representation

Utilizing the definitions provided in the previous figure, the effective angle of attack was defined with the following formula:

$$\alpha_{eff} = \alpha_g + \phi - \alpha_i \tag{5.1}$$

Where α_{eff} was the effective angle of attack, ϕ was the twist of the section, α_g was the global angle of attack of the wing, and α_i was the induced angle of attack caused by the downwash. Utilizing the equation determined in Appendix A, including a linear lift assumption, the effective angle of attack was solved numerically using the following relation:

$$0 = \left(2\pi\alpha_{eff} + C_{l0}\right)\sin\left(\alpha_g + \phi - \alpha_{eff}\right) - C_{di} \tag{5.2}$$

Where C_{l0} was the lift coefficient of the airfoil at zero angle of attack. This formula, using the linearly lift assumption already imposed by the VLM, accounted for both the tilting of the lift vector and its change in magnitude caused by the change in angle of attack. Equation (5.2) was solved numerically using MATLAB's *fzero()* function, producing a value for the effective angle of attack experienced by the wing strip. This was used as an input for the inviscid airfoil analysis.

5.3 Inviscid Airfoil Analysis

The inviscid analysis of the wing section, modelled as a 2D airfoil analysis, was required to compute the tangential speeds along the surface. This was implemented through the use of the linear-strength vortex method defined by Katz and Plotkin [3], similar in execution to XFOIL [7], and generating the solution at the effective angle of attack determined with the model in Section 5.2.

5.3.1 Airfoil Discretization

The 2D airfoil geometry was modelled as a series of vortex panels connecting the geometric nodes of the airfoil [3]. This produced a continuous curve of vortex panels from the upper trailing edge to the lower trailing edge, aligning with a linear interpolation of the airfoil geometry. A colocation point, where the vorticities were computed to be acting, was also placed at the midpoint of each vortex panel [3]. A unit-length normal and tangent vector was computed and placed on each colocation point. A visual representation of this discretized form was presented in Figure 5.2.

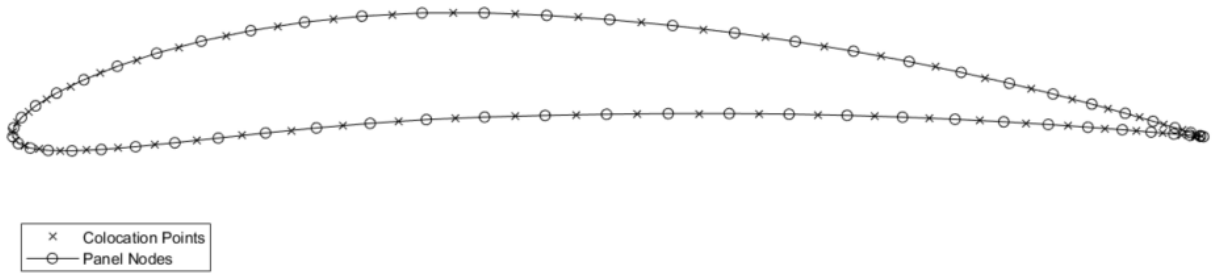


Figure 5.2: Airfoil Discretization

5.3.2 Mathematical Solution

The goal of the mathematical solution was to determine the vorticity of each vortex panel, which was then used to compute the pressure coefficients and tangential velocity acting along airfoil surface. This was represented in matrix form with the following matrix relation [3]:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,N+1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{N,N+1} \\ 1 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} \gamma_1 \\ \vdots \\ \gamma_N \\ \gamma_{N+1} \end{bmatrix} = \begin{bmatrix} RHS_1 \\ \vdots \\ RHS_N \\ 0 \end{bmatrix} \tag{5.3}$$

Where a was the aerodynamic influence coefficient, γ was the vorticity of a panel node, and RHS was the free-stream contribution. The size of the aerodynamic influence matrix was a square matrix of $N+1$, where N was the number of colocation points. The size of $N+1$ resulted from the use of linear strength vortex panels over constant strength panels, as vorticities were defined at the two ends of the panels. Additionally, the final row of the matrix representation was the inclusion of the Kutta condition, which specified that the two trailing edge vorticities must equal:

$$\gamma_1 + \gamma_{N+1} = 0 \tag{5.4}$$

The aerodynamic influence coefficients within this matrix were computed as the dot product of the self-induced velocity of the panel, from a unit strength vortex, and the normal vector [3]. This was presented with the following equation:

$$a_{ij} = \left\{ \left(u^b, w^b\right)_{i,j-1} + \left(u^a, w^a\right)_{i,j} \right\} \cdot \hat{n}_i \tag{5.5}$$

Where u and w were the velocity components induced by the jth vorticity, and $\hat{n}_i$ was the unit vector normal to the ith panel. The induced velocity components were calculated, in the panel coordinate system, with the following equations derived from the approach specified by Katz and Plotkin [3]:

$$u_j^a = -\frac{z}{2\pi}\left(\frac{\gamma_j}{x_2 - x_1}\right)\ln\left(\frac{r_2}{r_1}\right) + \frac{\gamma_j(x_2 - x)}{2\pi(x_2 - x_1)}(\theta_2 - \theta_1) \tag{5.6a}$$

$$u_j^b = \frac{z}{2\pi}\left(\frac{\gamma_{j+1}}{x_2 - x_1}\right)\ln\left(\frac{r_2}{r_1}\right) + \frac{\gamma_{j+1}(x - x_1)}{2\pi(x_2 - x_1)}(\theta_2 - \theta_1) \tag{5.6b}$$

$$w_j^a = -\left(\frac{\gamma_j(x_2 - x)}{2\pi(x_2 - x_1)}\right)\ln\left(\frac{r_2}{r_1}\right) - \frac{z}{2\pi}\left(\frac{\gamma_j}{x_2 - x_1}\right)\left[\frac{x_2 - x_1}{z} + (\theta_2 - \theta_1)\right] \tag{5.6c}$$

$$w_j^b = -\left(\frac{\gamma_{j+1}(x - x_1)}{2\pi(x_2 - x_1)}\right)\ln\left(\frac{r_1}{r_2}\right) + \frac{z}{2\pi}\left(\frac{\gamma_{j+1}}{x_2 - x_1}\right)\left[\frac{x_2 - x_1}{z} + (\theta_2 - \theta_1)\right] \tag{5.6d}$$

Where the panel was defined as the line connecting points 1 and 2, defined at locations (x_1, z_1) and (x_2, z_2). The vorticity acting at point 1 was set to γ_j, while the vorticity acting at point 2 was set to γ_{j+1}. The colocation point, for which the induced velocity was computed to be acting on, was defined as (x, z) within the panel coordinate system. The distance between the colocation point and point 1 was defined as r_1, while the distance to point 2 was defined as r_2. The angles were defined as the angle between $\overrightarrow{r_1}$ or $\overrightarrow{r_2}$ and the panel tangent vector.

To compute these values, the reference frame was required to shift into the local panel reference frame. Points were translated into the panel coordinate system with a rotation matrix [3]. Additionally, the origin of the reference frame was placed at point 1 of the panel. The computation to determine the positions within the panel reference frame was presented in the following formula [3]:

$$\begin{bmatrix} x_p \\ z_p \end{bmatrix} = \begin{bmatrix} \cos(\alpha_i) & \sin(\alpha_i) \\ -\sin(\alpha_i) & \cos(\alpha_i) \end{bmatrix} \begin{bmatrix} x - x_0 \\ z - z_0 \end{bmatrix} \tag{5.7}$$

Where the values (x_p, z_p) were within the panel reference frame, the values (x, z) were within the global reference frame, the values (x_0, z_0) specified the position of the panel origin in the global coordinate system, and α_i was the incidence angle of the panel. Once the velocities were computed, another relation was required to rotate them from the panel reference into the global reference [3]. This was presented below:

$$\begin{bmatrix} u \\ w \end{bmatrix} = \begin{bmatrix} \cos(-\alpha_i) & \sin(-\alpha_i) \\ -\sin(-\alpha_i) & \cos(-\alpha_i) \end{bmatrix} \begin{bmatrix} u_p \\ w_p \end{bmatrix} \tag{5.8}$$

Where, the values (u_p, w_p) were the velocity components in the panel reference frame, and the values (u, w) were the velocity components in the global reference frame. The remaining component for the solution was the free-stream contribution, which was computed with the following relation [3]:

$$RHS_i = \vec{Q}_\infty \cdot \hat{n}_i \tag{5.9}$$

Where $\vec{Q}_\infty$ was the free-stream air velocity vector. The vorticity of each panel node was then computed, in matrix form, with the following relationship:

$$[\gamma] = [A]^{-1}[RHS] \tag{5.10}$$

Where $[\gamma]$ was the vorticity vector, $[A]$ was the aerodynamic influence coefficient matrix, and $[RHS]$ was the free-stream contribution vector.

To compute the induced velocity at the various colocation points, and by extension the tangential velocity and pressure coefficients, an additional matrix computation was required. This was presented in the following formula [3]:

$$\begin{bmatrix} Q_{ind,1} \\ \vdots \\ Q_{ind,N} \\ 0 \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1,N+1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{N1} & b_{N2} & \cdots & b_{N,N+1} \\ 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} \gamma_1 \\ \vdots \\ \gamma_N \\ \gamma_{N+1} \end{bmatrix} \tag{5.11}$$

Where Q_{ind} was the induced velocity vector, and b was the tangential aerodynamic influence coefficient. The values for the tangential coefficients were computed with the following equation [3]:

$$b_{ij} = \left\{ \left(u^b, w^b\right)_{i,j-1} + \left(u^a, w^a\right)_{i,j} \right\} \cdot \hat{t}_i \tag{5.12}$$

Where $\hat{t}_i$ was the unit vector tangent to the panel.

5.3.3 Analysis Outputs

With the vorticities and induced velocity distribution along the airfoil surface computed, the tangential velocity and pressure coefficients could be determined. The tangential velocity was taken as the sum of the induced and free-stream velocity contributions, both acting tangential to the surface. This was represented with the following equation [3]:

$$Q_{t,j} = Q_{ind,j} + \vec{Q}_{\infty} \cdot \hat{t}_j \tag{5.13}$$

Where $Q_{t,j}$ was the tangential velocity and $\vec{Q}_{\infty}$ was the free-stream velocity vector. Utilizing the tangential velocity, the pressure coefficient acting on each panel was computed with the following relation [3]:

$$C_{P,j} = 1 - \left(\frac{Q_{t,j}}{|\vec{Q}_{\infty}|}\right)^2 \tag{5.14}$$

Where $C_{P,j}$ was the pressure coefficient.

An additional capability, required for the effective angle of attack computation, was the calculation of the zero angle-of-attack lift coefficient for the tested airfoil. This was calculated through a summation of the pressure effects, presented in the following equation [3]:

$$C_l = \sum C_{P,j} \Delta c_j \cos\left(\alpha_j\right) \tag{5.15}$$

Where C_l was the lift coefficient, Δc_j was the distance between the jth and $(j+1)$th colocation point, and α_i was the incident angle of the panel.

5.4 Viscous Airfoil Analysis

The viscous analysis of the airfoil was determined using the analysis scheme presented by Fujiwara et al [4]. This required the tangential velocity distribution along the upper and lower airfoils surfaces, and determined the corresponding sectional drag and boundary layer displacement thickness. The chosen approach was to implement a direct boundary layer scheme, due to its simplicity over the simultaneous solution method. While the simultaneous method would be capable of predicting low-Reynolds number performance accurately, this capability was considered to be unnecessary as the VLM mandated large Reynolds number cases [4]. As such, the direct method was selected, wherein the viscous solution was computed after the inviscid solution and was used to modify the geometry input to the inviscid calculator.

This analysis was computed twice for a given airfoil, once for each airfoil surface. As a precursor to execution, the airfoil was divided at its stagnation point, or whichever node presented the lowest tangential speed.

5.4.1 Laminar Boundary Layer

The laminar boundary layer solution was a compressible implementation of Curle's method [8], which was an adjustment on Thwaites' method [4], with empirical relations used to compute the boundary layer characteristics at each node using the previous node properties. For each node, the momentum thickness was calculated first using the tangential edge velocity computed by the inviscid airfoil solver. This was presented in the following relation:

$$\theta^2 u_e^6 = 0.441 \frac{\mu}{\rho} \int_0^s u_e^5 ds \tag{5.16}$$

Where θ was the momentum thickness of the boundary layer, u_e was the inviscid edge velocity set equal to the tangential speed computed by the inviscid solution, μ was the viscosity of air, ρ was the density of air, and s was the distance along the airfoil surface curve from the stagnation point. Converting this equation into a numerical iteration problem, it was computed in discrete form using the following equation:

$$\theta = \sqrt{\frac{0.441\mu}{\rho u_{e,i}^6} \sum_{j=1}^{i} u_{e,j}^5 \Delta s_j} \tag{5.17}$$

Where i was the index of the panel being solved for. For this panel, a local Reynolds number per unit length was also defined, as presented with the following definition [4]:

$$Re_l = \frac{\rho u_e}{\mu} \tag{5.18}$$

This value was subsequently used to compute the momentum thickness Reynolds number with the following relation:

$$Re_\theta = Re_l \theta \tag{5.19}$$

Where Re_θ was the momentum thickness Reynolds number. This value was used to determine if the flow has transitioned to turbulence using the condition presented in Section 5.4.2.

Following with Thwaites' approach, the dimensionless pressure-gradient parameter was computed through the following approximation [4]:

$$\lambda = \frac{\theta^2 \rho}{\mu} \frac{\Delta u_e}{\Delta s} \tag{5.20}$$

Where λ was the pressure-gradient parameter, and $\Delta u_e / \Delta s$ was the velocity gradient of the panel. This parameter was input to Twaites' empirical relations for the shape factor [4]:

$$H = \begin{cases} 3.9155 & , \ \lambda \leq -0.1 \\ 2.088 + \frac{0.0731}{\lambda + 0.14} & , \ -0.1 < \lambda \leq 0.0 \\ 2.61 - 3.75\lambda + 5.24\lambda^2 & , \ 0.0 < \lambda \leq 0.1 \\ 2.2874 & , \ \lambda > 0.1 \end{cases} \tag{5.21}$$

Where H was the shape factor.

An additional parameter, the l parameter was also computed empirically using relations developed by Thwaites, as presented below [4]:

$$l = \begin{cases} -0.1773 & , \ \lambda \leq -0.1 \\ 0.22 + 1.402\lambda + \frac{0.018\lambda}{\lambda + 0.107} & , \ -0.1 < \lambda \leq 0.0 \\ 0.22 + 1.57\lambda - 1.8\lambda^2 & , \ 0.0 < \lambda \leq 0.1 \\ 0.3590 & , \ \lambda > 0.1 \end{cases} \tag{5.22}$$

With the two parameters determined, the boundary layer displacement thickness and friction coefficient were computed for each panel. This was done with the following two relations [4]:

$$\delta^* = H\theta \tag{5.23}$$

$$C_f = \frac{2l}{Re_\theta}(1 + 0.2M^2) \tag{5.24}$$

Where δ^* was the displacement thickness of the boundary layer, C_f was the compressible friction coefficient for the panel, and M was the Mach number of the inviscid flow for the panel. The boundary layer properties were defined by equations (5.23) and (5.24) until the transition condition was met, and the turbulent analysis was used in its stead.

5.4.2 Transition Condition

The transition condition, following the approach of Fujiwara et al, was achieved through implementing Michel's transition criterion [4]. This first required the computation of the arc-length Reynolds number, as presented in the following definition:

$$Re_s = Re_l s \tag{5.25}$$

Where Re_s was the arc-length Reynolds number, and s was the distance along the airfoil surface curve from the stagnation point. The transition condition was subsequently computed with the following definition [4]:

$$Re_{\theta,transition} = 1.174\left(1 + \frac{22400}{Re_s}\right)Re_s^{0.46} \tag{5.26}$$

Where $Re_{\theta,transition}$ was the transition momentum thickness Reynolds number. This condition was computed for each panel within the laminar region, and once the momentum thickness Reynolds number was found to exceed this parameter, the flow was considered to have transitioned to turbulent flow. For this panel and all subsequent panels to the trailing edge, the turbulent boundary layer analysis was used.

5.4.3 Turbulent Boundary Layer

Once the flow exceeded the transition condition, the turbulent boundary layer model was used. The implementation involved using the boundary layer properties of a given panel to compute the characteristics at the succeeding panel. The first step of this process was computing the boundary layer shape factor with the following relation, developed by Stanford University through modifying the approach developed by Cebeci and Schlichting [9]:

$$H_1 = 3.0445 + \frac{0.8702}{(H - 1.1)^{1.2721}} \tag{5.27}$$

Where H_1 was the boundary layer shape factor, and H was the shape factor.

Another mathematical quantity required was termed the UTH factor and was computed with the following relation. This was required to use an empirical differential relationship presented by Fujiwara et al [4].

$$UTH = u_e \theta H_1 \tag{5.28}$$

Where UTH was the needed factor. This was required to compute the properties of the succeeding panel according to the following relation [4] which defined the change in this quantity with respect to the airfoil arc.

$$\frac{d}{ds}(UTH) = \frac{0.0306 u_e}{(H_1 - 3.0)^{0.6169}} \tag{5.29}$$

Using equation (5.29), an iterative scheme was produced to predict the value of the UTH factor for the succeeding panel, as presented in the following formula where Euler's method was employed:

$$UTH_{i+1} = UTH_i + \frac{d}{ds}(UTH) \cdot \Delta s \tag{5.30}$$

Where Δs was the length of the panel, equal to the arc length step. Computing the succeeding panel momentum thickness value required the use of the von Karmen's compressible integral momentum equation, reordered in the following relation [4]:

$$\frac{d\theta}{ds} = \frac{C_f}{2} - \frac{\theta}{u_e}(2 + H - M^2)\frac{\Delta u_e}{\Delta s} \tag{5.31}$$

This equation required known properties of the panel, but also required the friction coefficient. This was computed with the empirical relationship presented below [4]:

$$C_f = \frac{0.246}{10^{0.678H} Re_\theta^{0.268}}(1 + 0.2M^2) \tag{5.32}$$

Where Re_θ was the momentum thickness Reynolds number, and M was the Mach number of the flow above the panel. With the displacement thickness gradient defined, Euler's method was again used to compute the momentum thickness of the succeeding panel with the following iterative definition:

$$\theta_{i+1} + \theta_i + \frac{d\theta}{ds}\Delta s \tag{5.33}$$

Using equations (5.30) and (5.33), the properties of the succeeding panels were able to be computed. First, the boundary layer shape factor was computed from the momentum thickness, the inviscid tangential velocity, and the UTH factor as presented below:

$$H_1 = \frac{UTH}{\theta u_e} \tag{5.34}$$

The shape factor corresponding to this value was subsequently computed with the following empirical definition [4]:

$$H = \begin{cases} 3.0 & , \ H_1 < 3.33 \\ 0.6778 + 1.1536(H_1 - 3.3)^{-0.326} & , \ 3.3 \leq H_1 \leq 5.3 \\ 1.1 + 0.86(H_1 - 3.3)^{-0.777} & H_1 > 5.3 \end{cases} \tag{5.35}$$

Finally, the boundary layer displacement thickness was computed with equation (5.23), restated below:

$$\delta^* = H\theta \tag{5.36}$$

With the displacement thickness and friction coefficients determined, the relevant quantities of the boundary layer were able to be modelled. Additionally, following the recommendations of Fujiwara et al, once the solution was completed, the boundary layer thickness for the last 5% of the chord was overridden to have a constant thickness, correcting for the strong adverse pressure gradients produced by the trailing edge Kutta condition [4].

5.5 Inviscid-Viscous Coupling

Coupling of the inviscid and viscous airfoil calculators was achieved through an iterative scheme. First, the inviscid solution was computed using the defined airfoil geometry. The viscous analysis was subsequently computed using the tangential velocity distribution predicted by the inviscid analysis.

The viscous analysis generated a boundary layer displacement thickness as one of its outputs. This was used to adjust the airfoil geometry provided to the inviscid calculator, so that the inviscid analysis was conducted about the boundary layer rather than the airfoil geometry itself. This defined the iterative scheme, which was exited once the boundary layer thickness at the middle of the airfoil was no longer changing significantly.

A sample result was provided in Figure 5.3, showing a magnified view of an airfoil and boundary layer. The boundary layer displacement thickness was visibly thin, and only adjusted the geometry of the airfoil slightly. Nonetheless, the difference in predicted skin friction was significant between the first and last iteration, justifying the need for the iterative procedure.

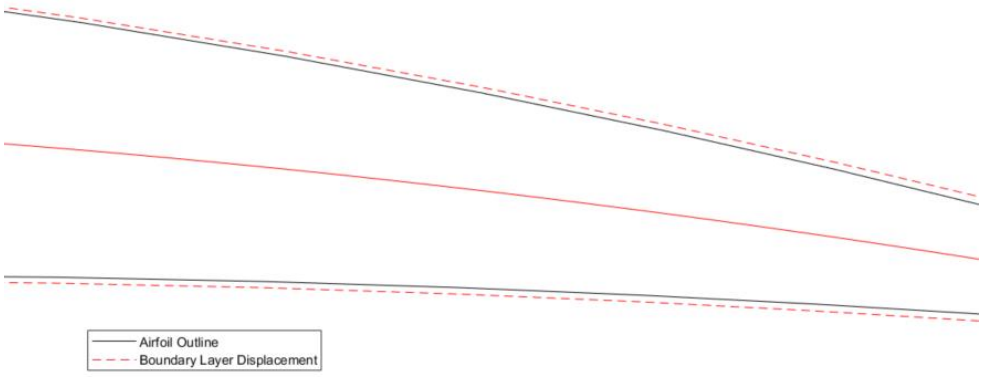


Figure 5.3: Viscous Airfoil Analysis Output

5.6 Parasitic Drag Computation

The parasitic drag acting on each section was computed from the friction coefficients output by the boundary layer analysis. This was defined with the following relation:

$$d_p = \frac{1}{2}\rho \sum Q_{t,i}^2 C_{f,i} \Delta s_i \tag{5.37}$$

Where d_p was the parasitic drag per unit length acting on the section, and Δs was the length of each panel comprising the airfoil.

The overall parasitic drag force acting on the section was computed by multiplying the sectional drag with the section width, as presented in the following equation:

$$\Delta D_{p,i} = d_{p,i}\Delta y_i \tag{5.38}$$

Where ΔD_p was the parasitic drag force acting on the section, and Δy was the spanwise length of the section. The total parasitic drag acting on the wing was taken as the summation of the drag force acting on each constituent section:

$$D_p = \sum \Delta D_{p,i} \tag{5.39}$$

Where D_p was the total parasitic drag acting on the wing.

6 Structural Analysis

Structural modelling was utilized to predict how the wing would deform under the loading conditions applied to it. This was done through an implementation of classical beam theory, which predicted the bending deflection and twist angle of each section along the wing.

6.1 Wing Box Bending Model

To facilitate computation of the wing box bending properties, the structure was modelled as a series of area concentrations. Through analysing the wing box in this form, an accurate approximation was possible for the moments of inertia and centroid, provided a sufficient resolution was utilized. By extension, this enabled the computation of the principal axes which were required to accurately predict the response of asymmetric cross sections. A visual representation of this discretization was presented in Figure 6.1, depicting the skin, spars, and stringers as a series of area concentration points.

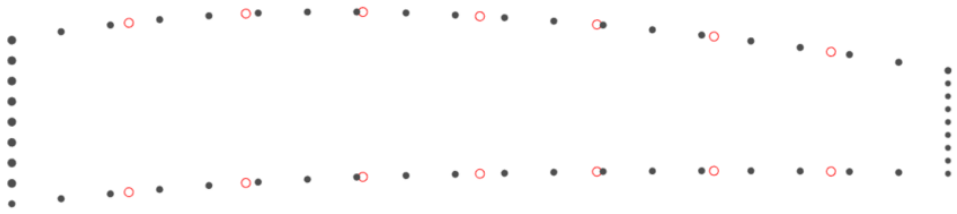


Figure 6.1: Wing Box as Area Concentrations

This numerical approximation scheme allowed for the following approximations to be used for the centroid computations [5]:

$$x_c = \frac{\sum x_i A_i}{\sum A_i} \tag{6.1a}$$

$$z_c = \frac{\sum z_i A_i}{\sum A_i} \tag{6.1b}$$

Where (x_c, z_c) was the location of the centroid, (x_i, z_i) was the location of each area concentration, and A_i was the area concentrated at each point. The moments and product of inertia were similarly computed according to the following numerical definitions [5]:

$$I_{xx} = \sum z_i^2 A_i \tag{6.2a}$$

$$I_{zz} = \sum x_i^2 A_i \tag{6.2b}$$

$$I_{zx} = \sum x_i z_i A_i \tag{6.3c}$$

Where I_{xx} and I_{zz} were the area moments of inertia, and I_{zx} was the product of inertia. With these computations, the moments and product of inertia were computed for the centroidal reference frame which was defined as having the origin placed at the centroid with the axes parallel to the global x- and z-axes.

For the computation of the bending response, the principal axes were required. This was computed according to the definition provided by Hibbeler [5]:

$$\theta_p = \frac{1}{2}\arctan\left(-\frac{I_{zx}}{0.5(I_{xx} - I_{zz})}\right) \tag{6.4}$$

Where θ_p was the angle between the principal axis and the global axis. Note that MATLAB's *atand()* function was used here to ensure the angle was bounded by $[-90, 90]$. A visual representation of the principal axes relative to the centroidal axes was provided in Figure 6.2.

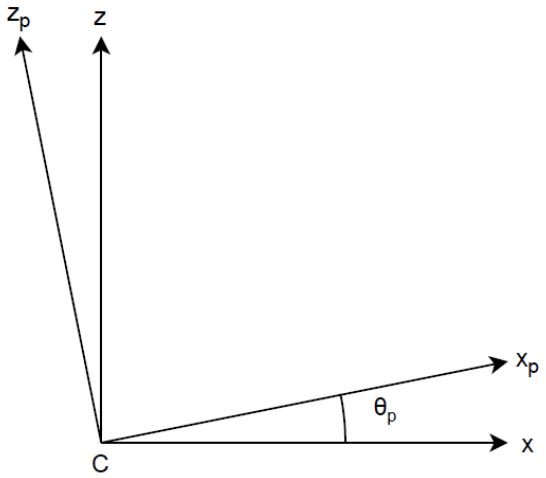


Figure 6.2: Principal Axes

Within the principal reference frame, defined by the principal axes, the global-reference moments and product of inertia were computed according to the following equations [5]:

$$I_{x'x'} = \frac{I_{xx}+I_{zz}}{2} + \frac{I_{xx}-I_{zz}}{2}\cos(2\theta_p) - I_{zx}\sin(2\theta_p) \tag{6.5a}$$

$$I_{z'z'} = \frac{I_{xx}+I_{zz}}{2} + \frac{I_{xx}-I_{zz}}{2}\cos(2\theta_p) + I_{zx}\sin(2\theta_p) \tag{6.5b}$$

$$I_{z'x'} = 0 \tag{6.5c}$$

Where $I_{x'x'}$ and $I_{z'z'}$ were the moments of inertia within the principal reference frame, and $I_{z'x'}$ was the product of inertia within the principal reference frame, set to zero by definition.

Converting the coordinates of points into and out of the principal reference frame was done according to the following matrix definitions:

$$\begin{bmatrix} x' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta_p & \sin\theta_p \\ -\sin\theta_p & \cos\theta_p \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix} \tag{6.6a}$$

$$\begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} \cos\theta_p & -\sin\theta_p \\ \sin\theta_p & \cos\theta_p \end{bmatrix} \begin{bmatrix} x' \\ z' \end{bmatrix} \tag{6.6b}$$

Where (x', z') was the coordinates of a given point in the principal reference frame, and (x, z) were the coordinates of the same point within the centroidal reference frame.

6.2 Wing Box Torsion Model

To compute the torsion response of the wing, the shear centre location was required. This was computed by applying an arbitrary unit-magnitude shear force in the x'-direction to locate the z'-coordinate of the shear center. Following this, a unit magnitude force was applied in the z' −direction and was used to locate the x'-coordinate of the shear center.

Locating the shear center first required the shear flow within the structure to be computed. This was defined with the following equation [10]:

$$q_s = -\left(\frac{S_x I_{xx} - S_z I_{zx}}{I_{xx} I_{zz} - I_{zx}^2}\right)\int_0^s tx\, ds - \left(\frac{S_z I_{zz} - S_x I_{zx}}{I_{xx} I_{zz} - I_{zx}^2}\right)\int_0^s ty\, ds + q_{s,0} \tag{6.7}$$

Where q_s was the shear flow, $q_{s,0}$ was the constant shear flow, S_x and S_z were the applied shear forces, and s was the distance along the curve of the wing box surface. The summation of the two integral terms and their coefficients was termed the "basic shear flow" [10], and was given the identifier q_b.

Numerically approximating the basic shear flow, and computing its value within the principal reference frame, and thereby setting the product of inertia to zero, the following formula was used to compute this parameter for the analyser:

$$q_{b,i} = -\frac{S_x}{I_{z'z'}}\sum_{j=i}^{i}\left(t_i x_j' \Delta s_j\right) - \frac{S_z}{I_{x'x'}}\sum_{j=0}^{i}\left(t_j z_j' \Delta s_j\right) \tag{6.8}$$

Where $q_{b,i}$ was the basic shear flow value at a wing box node, t_i was the thickness of the skin or spar at that node, (x_j', z_j') was the coordinates of that node within the principal reference frame, and Δs_j was the distance between the node and the previous node. The selection of the initial node, and thereby the first index, was arbitrary and did not affect the solution to the overall shear flow.

The constant shear flow was also computed using a numerical approximation, defined by the following formula [10]:

$$q_{s,0} = -\frac{\sum q_{b,i}\Delta s_i/t_i}{\sum \Delta s_i/t_i} \tag{6.9}$$

With the shear flow numerically approximated along the aircraft surface, the following definition was used as a basis for computing the location of the shear centre [10]:

$$S_x\eta - S_z\xi = \oint pq_b\, ds + 2A_{mean}q_{s,0} \tag{6.10}$$

Where η was the z-coordinate of the shear center, ξ was the x-coordinate of the shear center, p was the lever arm of the shear flow acting on the origin of the principal reference frame, and A_{mean} was the mean 2D area encased by the wing box. This was numerically approximated into the following form:

$$S_x\eta - S_z\xi = \sum p_i q_{b,i}\Delta s_i + 2A_{mean}q_{s,0} \tag{6.11}$$

Additionally, with a derivation provided in Appendix B, the shear arm was computed with the following formula:

$$p_i = \left|\frac{z_i - \frac{\Delta z}{\Delta x}x_i}{1-\left(\frac{\Delta z}{\Delta x}\right)^2}\right|\sqrt{\left(\frac{\Delta z}{\Delta x}\right)^2 + 1} \tag{6.12}$$

Where p_i was the lever arm corresponding to the wing box node, $\Delta z/\Delta x$ was the slope of the wing box at the node, and (x_i, z_i) was the coordinates of the node. Equation (6.11) was used twice, once with the x'-direction force and a second time with z'-direction force. For each, the force in the other direction was set to zero and the coordinates were solved through algebraic manipulation. The coordinates were then rotated back into the centroidal reference frame with equation (6.6b).

6.3 Wing Box Scaling Factors

For computational expediency, the properties of the wing box were computed to correspond to a unit-length airfoil cross section. Its various characteristics were then multiplied by scaling factors to produce the properties of the wing box at a given section. These were presented in Table 6.1, where properties were scaled as a function of chord length c.

Table 6.1: Wing Box Scaling Factors

Parameter	**Scaling Factor**
Length (Δs)	c
Skin/Spar Thickness	(unchanged)
Coordinates (x, z)	c
Area Concentration (A_i)	c
Mean Area (A_{mean})	c^2
Moment of Inertia (I_{xx}, I_{zz})	c^3
Product of Inertia (I_{zx})	c^3
Principal Angle (θ_p)	(unchanged)

6.4 Wing Loading

The loading applied to the wing was computed through analysing the lift forces generated by the VLM on each section, producing a load distribution along the wingspan. As with the viscous aerodynamic analyzer, the sections were divided by the panel boundaries which were parallel to the free stream.

6.4.1 Torsional Loads

Computing first the torsional loading, the torque acting on each section was computed with the following formula:

$$T_{s,j} = \sum \Delta L_i (x_{sc} - x_i) \tag{6.13}$$

Where $T_{s,j}$ was the torque acting on the section, ΔL_i was the lift force acting on teach panel within the section, x_i was the colocation point x-coordinate where the force was acting, and x_{sc} was the shear centre x-coordinate.

To compute the torsion distribution along the wingspan, the torque acting on the root section was first set as the negative total torque acting on the wing, as defined in the following equation [5]:

$$T(y_1) = -\sum T_{s,j} \tag{6.14}$$

Where $T(y_1)$ was the value of the torque distribution function at the root. The values for the torque distribution function were then computed iteratively according to the following formula:

$$T(y_j) = T(y_{j-1}) + T_{s,j} \tag{6.15}$$

Where y was the spanwise coordinate.

6.4.2 Bending Loads

To compute the bending loads, the shear force and bending moment functions were defined along the span of the wing. First, the shear load distribution was computed, with the shear load acting on the root section set to equal the negative total lift force acting on the wing [5]:

$$V(y_1) = -L \tag{6.16}$$

Where $V(y_1)$ was the value of the shear load function at the root. The subsequent values for the shear load function were defined iteratively as presented below:

$$V(y_j) = V(y_{j-1}) + \sum \Delta L_{ij} \tag{6.17}$$

Where $\sum \Delta L_{ij}$ was the total lift force acting on the section.

The moment distribution was computed in a similar iterative fashion. The bending moment at the wing tip was first set to zero [5]:

$$M(y_n) = 0 \tag{6.18}$$

Where $M(y_n)$ was the value of the moment function at the wing tip. The remaining values for the function were computed iteratively from the tip to the root with the following definition [5]:

$$M(y_j) = M(y_{j+1}) + \frac{1}{2}\Delta y_j \left(\sum \Delta L_{ij}\right) \tag{6.19}$$

Where $\Delta y_j$was spanwise panel width, and $\sum \Delta L_{ij}$ was the lift force acting on the section.

6.5 Wing Deformation

Deformation of the wing was computed from the twist and deflection distribution along the span. For this analysis, first the twist angle distribution along the wing was computed. Following this, the deflection angle and vertical deflections were found, using the twist angle to adjust the point tested in determining the radius of curvature.

6.5.1 Twist Angle Prediction

The twist angle at each point along the wingspan was calculated through the following approximation provided by Hibbeler [5]. The twist angle at the root was set to zero degrees, while each successive angle was computed with the torsion distribution function.

$$\phi_{j+1} = \phi_j - \frac{T(y_j)\Delta y_j}{4A_{mean}^2 G}\sum\frac{\Delta s_i}{t_i} \tag{6.20}$$

Where ϕ was the twist angle and Δy_j was the spanwise panel width. The quantity $\sum\frac{\Delta s_i}{t_i}$ was computed along the wing box surface.

6.5.2 Deflection Prediction

Prediction of the deflection required the radius of curvature for the wing box to be determined at each spanwise section. To compensate for the twist of the wing and ensure that the computed deflection was in the vertical direction, the point probed to compute this value was rotated by the twist angle.

Prior to this computation, however, the moment vector components acting on the structure were required. Resulting from the lift force distribution, this vector was parallel to the global x-axis while the components acting along the principal axes was needed, and further accounting for the rotation caused by twist was necessary.

The deconstruction of the moment vector was presented in Figure 6.3. In this figure the axes shown were centered on the wing box centroid. The x-axis was set parallel to the global x-axis, while the x_c-axis was set parallel to the untwisted x-axis of the wing box. Finally, the x_p- and z_p-axes were the principal axes of the wing box structure.

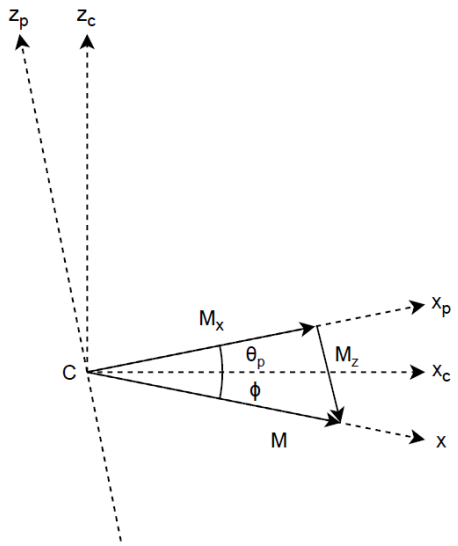


Figure 6.3: Moment Vector Components

Geometrically derived from Figure 6.3, the bending moment vector acting on the wing box was separated into its constituent principal-axis components with the following to equations:

$$M_x = M\cos(-\theta_p - \phi) \tag{6.21a}$$

$$M_z = M\sin(-\theta_p - \phi) \tag{6.21b}$$

Where M_x was the moment vector component about the principal x-axis, M_z was the moment vector component about the z-axis, and M was the total bending moment acting on the cross section.

Having determined the bending moment components acting on the structure, the lateral strain acting on any arbitrary point within the wing box structure was able to be computed with the following formula [5]:

$$\varepsilon_y = \frac{M_z x'}{EI_{z'z'}} - \frac{M_x z'}{EI_{x'x'}} \tag{6.22}$$

Where ε_y was the lateral strain, (x', z') was the coordinates of an arbitrary point defined in the principal reference frame, and E was the elastic modulus of the material. To compute the lateral strain value used for the deflection angle computation, an arbitrary point near the top of the wing box was probed. To align the radius of curvature and ensure that the computed deflection was vertical, the probed point was rotated from the centroidal axis by the twist angle as presented in Figure 6.4.

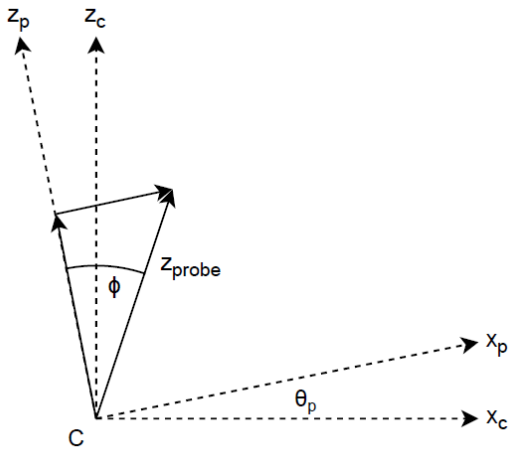


Figure 6.4: Probe Point Selection

Deriving from the geometry presented in Figure 6.4, the probed point defined within the principal reference frame was computed with the following equations:

$$x' = z_{probe}\sin(\theta_p + \phi) \tag{6.23a}$$

$$z' = z_{probe}\cos(\theta_p + \phi) \tag{6.23b}$$

Where (x', z') was the probed point coordinates defined in the principal reference frame, and z_{probe} was the z-axis component of the point probed for the strain.

With the lateral strain value computed, the deflection angle slope was able to be approximated with the following equation [5]:

$$\frac{d\theta}{dy} = -\frac{\varepsilon_y}{z_{probe}} \tag{6.24}$$

Where θ was the deflection angle, and $d\theta/dy$ was the deflection angle slope. Employing Euler's method alongside an initial deflection angle of zero at the root, the successive deflection angles along the wingspan were computed with the following formula:

$$\theta_{j+1} = \theta_j + \frac{d\theta}{dy}\,\Delta y_j \tag{6.25}$$

Where Δy_j was the spanwise section length. Conducting this computation along the entire wingspan resulted in the deflection angle being determined for each section of the wing.

Finally, with the deflection angle at each location along the wing computed, the vertical deflection was determined through a similar iterative scheme. This was presented in the following formula, where the deflection at the wing root was set to zero.

$$\Delta z_{j+1} = \Delta z_j + \tan\theta_j\,\Delta y_j \tag{6.26}$$

Where Δz was the vertical deflection of the wing section.

6.5.3 Mesh Deformation

To deform the VLM mesh using the predetermined twist and deflection distribution, the wing was again divided into spanwise sections, aligning with the free-stream-parallel panel boundaries. Each point defining the panel geometry was rotated and deflected according to the following equation:

$$x_d = \sqrt{x^2 - z^2}\cos\left(\arctan\left(\frac{z}{x}\right) + \phi_j\right) \tag{6.27a}$$

$$z_d = \sqrt{x^2 - z^2}\sin\left(\arctan\left(\frac{z}{x}\right) + \phi_j\right) + \Delta z_j \tag{6.27b}$$

Where (x_d, z_d) was the location of the panel point after it was deformed, and (x, z) was the location of the panel point before the deformation. Additionally, MATLAB's *arctan2()* function was used to compute the angle to ensure that the point was rotated from the correct quadrant.

7 Sample Results

7.1 Tested Wing Configuration

To demonstrate the output capabilities of the developed multidisciplinary analysis program, a generic low-sweep wing was utilized, intended to represent a typical turboprop aircraft. To further illustrate the wing twisting, a longer wing was utilized. The parameters which defined this computation were presented in Table 7.1.

Table 7.1: Sample Parameters

Air Properties	
Density	1.225 kg/m^3
Viscosity	1.789•10^{-5} kg/(m•s)
Velocity	58 m/s
Mach Number	0.2
Angle of Attack	3°
Wing Parameters	
Airfoil	NACA 6409
Root Chord	2.5 m
Taper Ratio	0.29
Semi-Span Length	18 m
Kink Distance Ratio	0.23
Leading Edge Sweep	3°
Wing Box Parameters	
Forward Spar	20% Chord
Rear Spar	70% Chord
Number of Stringers	2×7
Stringer Area	0.0002 m^2
Skin Thickness	0.001 m
Spar Thickness	0.003 m
Elastic Modulus	68.3•10^9 Pa
Shear Modulus	35.8•10^9 Pa

7.2 Aerodynamic Results

7.2.1 Lift Distribution

One of the core outputs of the VLM was the lift distribution along the aircraft wing. This was presented in Figure 7.1, where greater lift forces were demarked in yellow and lower forces were indicated with blue.

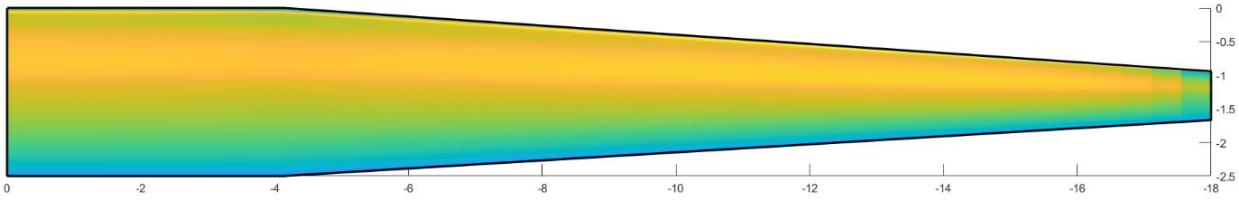


Figure 7.1: Sample Lift Distribution

7.2.2 Aerodynamic Forces

Combining the outputs of both aerodynamic analyzers, the aerodynamic forces acting on the aircraft were computed and presented in Table 7.2. This included the lift and induced drag, which were computed by the VLM, and the parasitic drag output by the viscous analyzer. The overall lift to drag ratio of the wing was also presented.

Table 7.2 Sample Aerodynamic Forces

Output	Value
Lift	61370 N
Induced Drag	870 N
Parasitic Drag	430 N
Lift-to-Drag Ratio	47

7.3 Wing Loading Results

7.3.1 Bending Loads

The loading outputs predicted by the bending solver include the shear force and bending moment distribution. The shear force corresponding to this sample case, modeled as a function of spanwise distance, was presented in Figure 7.2. The bending moment distribution which corresponded to this was presented in Figure 7.3.

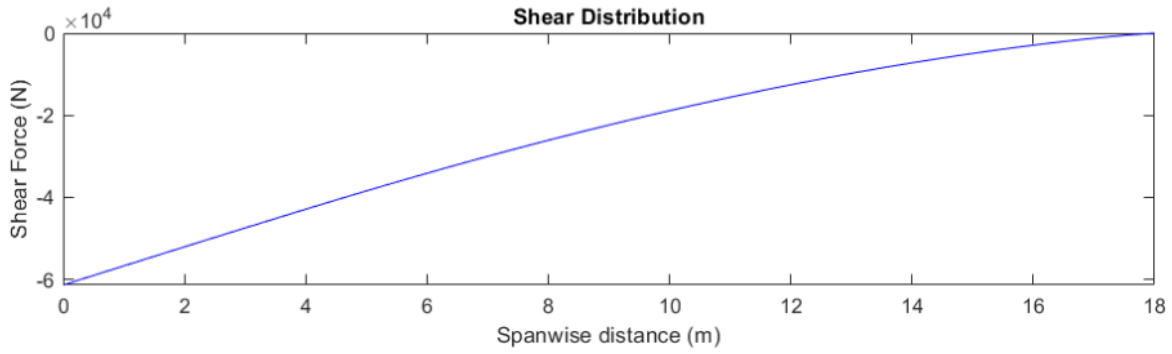


Figure 7.2: Sample Shear Distribution

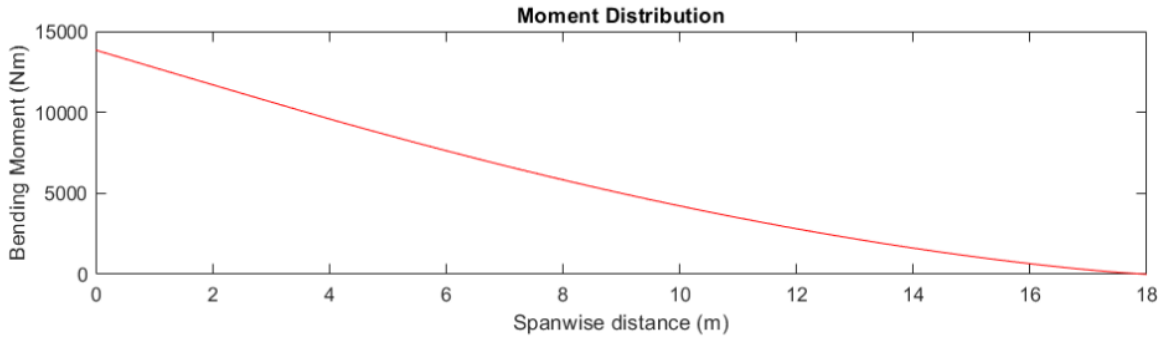


Figure 7.3: Sample Bending Moment Distribution

7.3.2 Torsion Loads

The torsion load calculator was responsible for predicting the torque distribution as a function of the spanwise distance. The plot presenting this for the sample case was given in Figure 7.4.

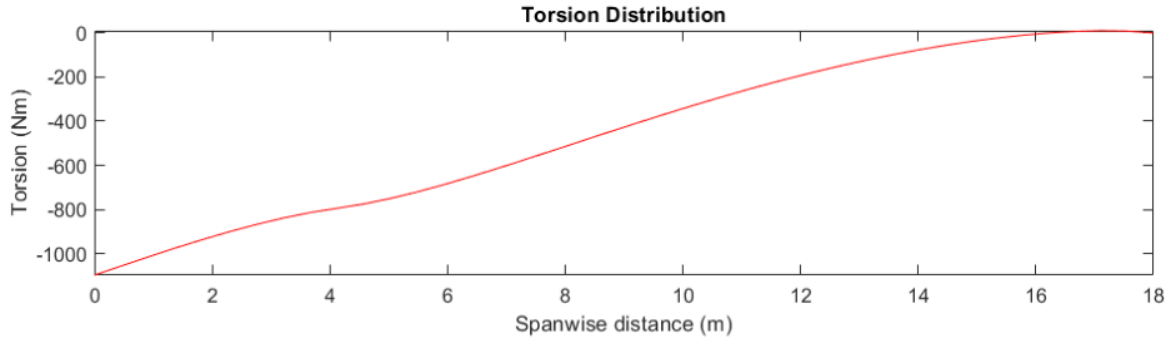


Figure 7.4: Sample Torsion Distribution

7.3.3 Resulting Deformation

The core functionality of the wing structural analyzer was the bending deflection and twist angle as a function of spanwise distance. The bending deflection predicted for this sample cases was presented in Figure 7.5, while the predicted twist was shown in Figure 7.6.

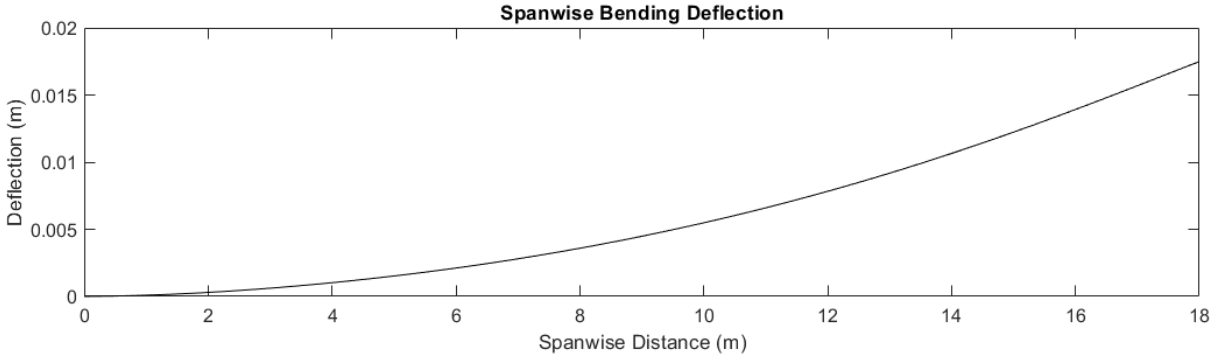


Figure 7.5: Sample Wing Deflection

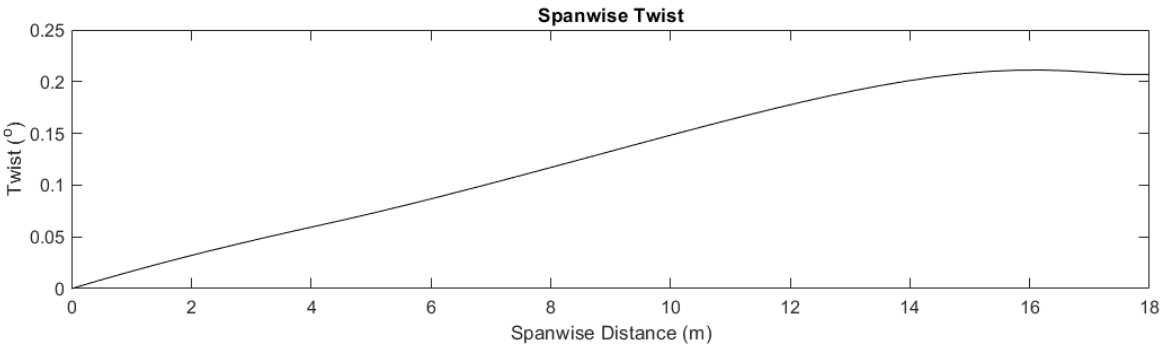


Figure 7.6: Sample Wing Twist

7.4 Computer Performance

As the developed program was a numerical approximation, a trade-off existed between simulation resolution and compute time. A series of resolutions were tested, with the required iterations and time to compute presented in Table 7.3. These results were produced using a MATLAB environment given 24 parallel workers, on an AMD Ryzen-9 3950X computer system with 16 CPU cores and 32 gigabytes of RAM. During the program operation, the system was able to maintain approximately 90% CPU utilization with the developed architecture.

Table 7.3: Computational Requirements

Number of x-panels	Number of y-panels	Iterations to Converge	Time to Compute (s)
3	7	10	1.4
10	40	7	4.2
40	40	4	20.8
60	40	5	55.8
60	80	5	213.9
60	100	5	330.2

8 Conclusions

An aerostructural analyzer was developed to model the deformation and aerodynamic loading of a low-sweep wing. This implementation was developed within the MATLAB environment, and included coupled aerodynamic and structural solvers. Each of these encapsulated solving functions were further subdivided into additional functions which sequentially or iteratively generated the required outputs.

The aerodynamic analysis approach used was the quasi-three-dimensional modelling approach, with a separate inviscid and viscous analyzer [2]. The inviscid effects were modelled with a vortex ring lifting surface model, implementing the method presented by Katz and Plotkin [3]. Viscous effects, specifically that of parasitic drag, were modelled with strip theory analysis using a 2D airfoil analyzer [2]. This 2D analysis was in itself subdivided, including an effective angle of attack computation, an inviscid linear-strength vortex airfoil analysis [3], and an empirical direct boundary layer computation [4].

Structural modelling was supplied through an implementation of classical beam theory [5]. This was comprised of a torsional twist prediction and a bending deflection computation. For solution purposes, the wing box was numerically approximated as a swarm of area concentrations that allowed for the rapid determination of the wing box centroid, moments of inertia, and shear centre.

The global solution was produced through a coupling of these solvers. For a given analysis, the implementation would iteratively converge the inviscid loading and structural response loads. Once this was completed, the viscous analysis was run to develop the parasitic drag load and allow for the computation of the wing aerodynamic efficiency. As developed, the analyzer was capable of predicting the dominant aerodynamic loads for the modeled non-rigid wing within acceptable time frames.

9 References

[1] J. Martins and A. Lambe, "Multidisciplinary Design Optimization: A Survey of Architectures," *AIAA Journal,* vol. 51, no. 9, pp. 1-48, 2013.

[2] A. Elham and M. van Tooren, "Tool for preliminary structural sizing, weight estimation, and aeroelastic optimization of lifting surfaces," *Proceedings of the Institution of Mechanical Engineers,* vol. 230, no. 2, pp. 280-295, 2015.

[3] J. Katz and A. Plotkin, Low-Speed Aerodynamics, New York: Cambridge University Press, 2001.

[4] G. Fujiwara, D. Chaparro and N. Nguyen, "An Integral Boundary Layer Direct Method Applied to 2D Transonic Small-Disturbance Equations," in *AIAA Applied Aerodynamics Conference*, Washington DC, 2016.

[5] R. C. Hibbeler, Mechanics of Materials, Pearson Prentice Hall, 2011.

[6] M. Selig, "UIUC Airfoil Coordinates Database," University of Illinois, [Online]. Available: https://m-selig.ae.illinois.edu/ads/coord_database.html#N. [Accessed 2021].

[7] M. Drela, "XFOIL: An Analysis and Design System for Low Reynolds Number Airfoils," *Low Reynolds Number Aerodynamics,* pp. 1-12, 1989.

[8] N. Curle, The Laminar Boundary Layer Equations, Oxford: Oxfor University Press, 1962.

[9] B. J. Cantwell, "Viscous Flow Along a Wall," Stanford University Department of Aeronautics and Astronautics, [Online]. Available: https://web.stanford.edu/~cantwell/AA200_Course_Material/Ch09_Viscous_Flow_Along_a_Wall.pdf. [Accessed March 2021].

[10] T. H. G. Megson, Aircraft Structures for Engineering Students, Kidlington: Elsevier Ltd., 2017.

Appendix A: Effective Angle of Attack Derivation

Given that the induced drag coefficient C_{di}, the global angle of attack α_g, and twist angle ϕ were known, and that α_i was unknown.

The effective angle of attack was defined with the following equation:

$$\alpha_{eff} = \alpha_g + \phi - \alpha_i \tag{A.1}$$

Using the linear-lift assumption from Thin Airfoil Theory, the lift coefficient function was defined as:

$$C_l(\alpha) = 2\pi\alpha + C_{l0} \tag{A.2}$$

Where α was some angle of attack in radians, and C_{l0} was the zero angle-of-attack lift coefficient of the aircraft. The C_{l0} was assumed to be a known value.

The induced drag coefficient was defined with the following equation:

$$C_{di} = C_l(\alpha_{eff}) \cdot \sin\alpha_i \tag{A.3}$$

Where C_{di} was the induced drag coefficient.

Equation (A.3) was rearranged into the following form, and equation (A.2) was substituted in:

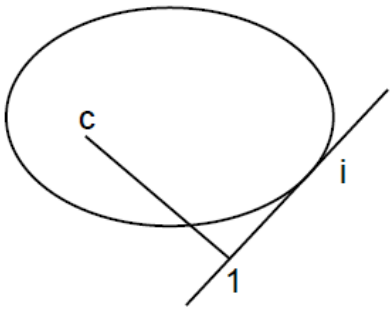
$$0 = (2\pi\alpha_{eff} + C_{l0})\sin\alpha_i - C_{di} \tag{A.4}$$

Equation (A.1) was rearranged and substituted in for α_i. This produced the final equation used for determination of the effective angle of attack:

$$0 = (2\pi\alpha_{eff} + C_{l0})\sin(\alpha_g + \phi - \alpha_{eff}) - C_{di}$$

Appendix B: Shear Center Arm Derivation

Given the arbitrary cross section presented in the following figure of an arbitrary cross section for a thin member:



Three points where defined. Point c was located at the centroid, point i was located on the edge of the thin member, and point 1 was set where the lever arm met the tangent line. The distance between point c and point 1 was defined as the lever arm distance with the following equation:

$$p = \sqrt{(x_1 - x_c)^2 + (z_1 - z_c)^2} \tag{B.1}$$

The tangent line was defined as having the slope $m = dz/dx$, which permitted the following two definitions for the tangent and lever lines:

$$\frac{z_1 - z_i}{x_1 - x_i} = m \tag{B.2}$$

$$\frac{z_1 - z_c}{x_1 - x_c} = \frac{1}{m} \tag{B.3}$$

Equation (B.3) was rearranged and substituted into equation (B.1) to produce the following relation.

$$p = \sqrt{m^2(z_1 - z_c)^2 + (z_1 - z_c)^2} \tag{B.4}$$

Equation (B.4) was simplified into the following form:

$$p = \sqrt{m^2 + 1}|z_1 - z_c| \tag{B.5}$$

Equations (B.2) and (B.3) were rearranged into the following forms:

$$z_1 = m(x_1 - x_i) + z_i \tag{B.6}$$

$$x_1 = m(z_1 - z_c) + x_c \tag{B.7}$$

Equation (B.7) was substituted into equation (B.6) to produce the following formula.

$$z_1 = \frac{mx_c - m^2 z_c - mx_i + z_i}{1 + m^2} \tag{B.8}$$

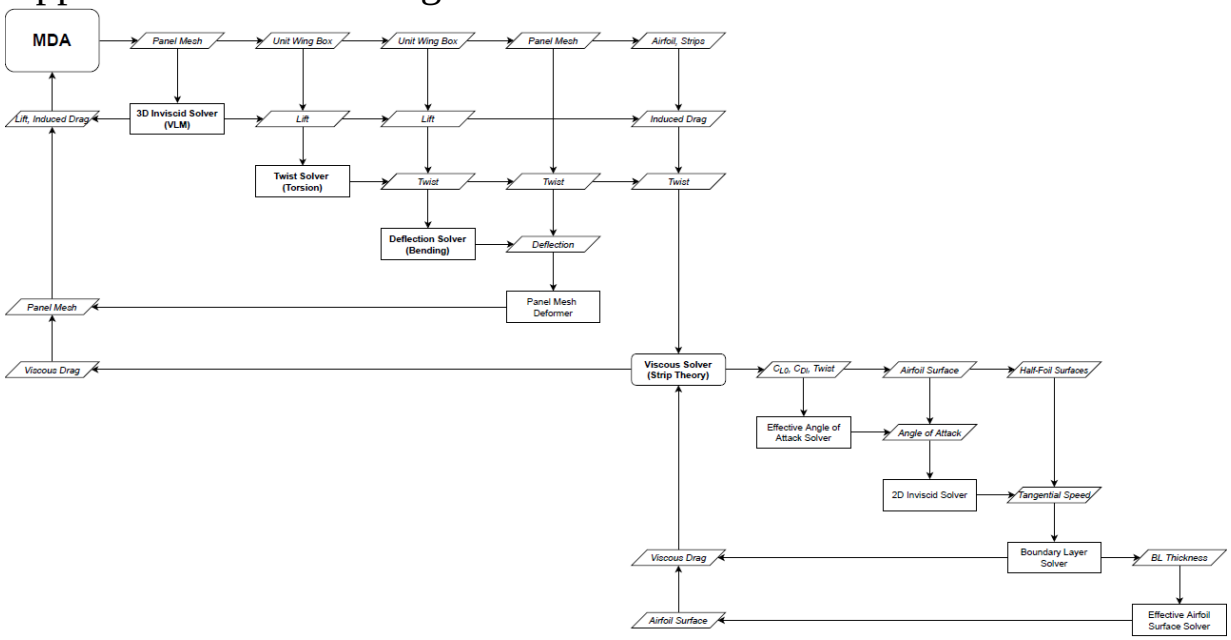
Setting $x_c = 0$ and $z_c = 0$ as the centroid was used the origin for the reference frame. Equations (B.5) and (B.8) combined to become:

$$p = \left|\frac{z_i - m x_i}{1 - m^2}\right| \sqrt{m^2 + 1} \tag{B.9}$$

Substituting the definition of m yielded the final equation below:

$$p = \left|\frac{z_i - \frac{dz}{dx} x_i}{1 - \left(\frac{dz}{dx}\right)^2}\right| \sqrt{\left(\frac{dz}{dx}\right)^2 + 1}$$

Appendix C: Global Program Architecture



Appendix D: Main MATLAB Function

```
% wing_solve_main.m
%
% Description
%   This function is the main function used to operate the aero-structual
%   analysis suite
%
% Written by:   Julian Bardin
% Date:         2021-02-09
%
clear;clc;

% Initialize parallel compute workers
workers = gcp;

% Add subfolders to the path
addpath('./Aerodynamics/');
addpath('./Utilities/');
addpath('./Structures/');

% Define the simulation parameters
V_inf = 58;   % Free stream velocity in m/s
M_inf = 0.2;  % Free stream mach number
rho   = 1.225; % Density of air
mu  = 1.789E-5 ;    % viscosity of air

% Read in the airfoil
[airfoil] = read_selig_foil('../01_Data/airfoil_test_NACA6.txt');
% [airfoil] = read_selig_foil('../01_Data/airfoil_test.txt');
% render_airfoil(airfoil);

% Define a placeholder wing
wing            = Wing_Param();
wing.airfoil    = airfoil;
wing.aoa        = 3;
wing.chord_root = 2.5;
wing.tr_tip     = 0.29;
wing.semi_span  = 18;
wing.sweep_LE   = 3;
wing.dr_kink    = 0.23;

% Define the placeholder wing box
stringers       = 7; % Number of stringers on top or bottom surface
A_stringers     = 0.0002;
t_skin          = 0.0010;
t_spar          = 0.0030;
box_chords      = [0.2 0.7];
E               = 68.3E9;    % Young's modulus in Pa
G               = 35.8E9;    % Shear modulus in Pa

% Compute the zero-alpha lift coefficient
foil = Airfoil(airfoil);
[CP,Qt,foil] = airfoil_solver_inviscid(foil,0,V_inf);
Cl0 = calc_airfoil_Cl(foil,CP);

% Define the resolution of the 3D VLM Analysis
x_panels = 60;
y_panels = 40;
% x_panels = 3;
% y_panels = 7;

% Define the resolution of the wingbox resolution
res_skin = 20;
res_spar = 9;

% Split the wing planform up into panels
wing_calc = convert_wing_obj(wing,x_panels,y_panels);
wing_calc_ref = wing_calc;

% Determine the wingbox for a unit-length airfoil
```

```
box_ref =
struct_solver_wingbox(foil,box_chords,stringers,t_skin,t_spar,A_stringers,res_skin,res_spar);

% Compute the coupled inviscid-lift and structural analysis
fprintf('-----------------------------------------------------\n');
L_last     = 0;
twist_last = zeros(1,y_panels+1);
dz_last    = zeros(1,y_panels+1);
scale_factor = 0.4;
exit_percent = 0.1;
for i = 1:50
    % Wing Inviscid Lift and Induced Drag Calculator
    [L,Di,dL,dDi,w] = aero_solver_inviscid(wing_calc,wing.aoa,V_inf,M_inf,rho);

    % Compute the deflection of the wing due to bending
    [y_lift,dz,twist,T_lift,V_lift,M_lift]...
        = struct_solver_deformation(wing_calc_ref,box_ref,L,dL,E,G);

    % Set the twist and deflection using the scale factor
    twist = scale_factor.*(twist-twist_last) + twist_last;
    dz    = scale_factor.*(dz-dz_last)       + dz_last;

    % Calculate the relative difference in lift
    percent_diff = abs(L-L_last)/abs(L_last)*100;

    % Output the percent difference
    fprintf('Lift change for iteration %2.2d: %6.3f %%\n',i,percent_diff);

    % If the change in lift is less than the exit percent, exit
    if percent_diff < exit_percent
        fprintf('Aerostructural analysis complete with %d iterations\n',i);
        break;
    end

    % Bend and twist the wing;
    wing_calc = wing_bend_and_twist(wing_calc_ref,box_ref,twist,dz);

    % Store the current values for the next iteration to compare against
    L_last      = L;
    twist_last  = twist;
    dz_last     = dz;
end
fprintf('-----------------------------------------------------\n');

% Wing Parasitic Drag Calculator
AoA = wing.aoa*ones(1,y_panels+1) + twist;
Dp = aero_solver_viscous(wing_calc,foil,y_panels,dDi,V_inf,rho,mu,AoA,Cl0,M_inf);

% Output the Aerodynamic Properties
fprintf('Lift:              %10.2f N\n',L);
fprintf('Drag (Induced):    %10.2f N\n',Di);
fprintf('Drag (Parasitic):  %10.2f N\n',Dp);
fprintf('L/D:               %10.2f\n',L/(Di+Dp));
fprintf('-----------------------------------------------------\n');

% Render wingbox cross section
% render_wingbox(box_ref);
% render_loading(y_lift,V_lift,M_lift,T_lift);
% render_structure_response(y_lift,dz,twist);

% Render the wing planform
% render_wing_2D(wing, wing_calc_ref);
% render_wing_3D(wing, wing_calc_ref,1);
% render_wing_3D_lift(wing, wing_calc_ref,dL);
% render_airfoil(foil);
% [CP,~,foil] = airfoil_solver_inviscid(foil,wing.aoa,V_inf);
% render_airfoil_CP(foil,CP);
% [~,zBL] = airfoil_solver_viscous(foil,7,0,V_inf,rho,mu,M_inf);
% render_airfoil2(foil,zBL)
```

Appendix E: Aerodynamic Functions

```
% aero_solver_inviscid.m
%
% Description:
%   This function encapsulates the inviscid aerodynamic solver. This is an
%   implementation of the vortex-lattice-method, where the wing is
%   approximated as a thin sheet along its mean-camber line. Thickness
%   effects are neglected. The resulting computation produces the lift and
%   drag distributions for the wing, and the total summation of both.
%
% Written by:   Julian Bardin
% Date:         2021-02-14
%

function [L,Di,dL,dDi,w] = aero_solver_inviscid(wing_calc,AoA,V_inf,M_inf,rho)
% Calculate the infulence coefficient matrices
[A,B] = calc_influence_coeffs(wing_calc);

% Convert the free-stream velocity into vector form
Q_inf = calc_vel_vector(V_inf,AoA);

% Calculate the right-hand-side vector used in solving for vorticity
RHS = calc_RHS(wing_calc,Q_inf);

% Calculate the vorticity vector
Gamma = A\RHS;

% Calculate the lift distribution and wing lift
[L,dL] = calc_panel_lift(wing_calc,Gamma,V_inf,M_inf,rho);

% Calculate the induced vertical velocities
w = B*Gamma;

% Calculate the induced drag distribution and total wing induced drag
[Di,dDi] = calc_panel_induced_drag(wing_calc,Gamma,w,M_inf,rho);
end
```

```
% aero_solver_viscous.m
%
% Description:
%   This function encapsulatese the viscous drag computations used for the
%   wing. This includes determining the effective angle of attack of each
%   airfoil section based on the induced drag, and ustilizing von Karmen's
%   equations to compute drag.
%
% Written by:   Julian Bardin
% Date:         2021-03-20
%

function Dp = aero_solver_viscous(wing_calc,foil,y_panels,dDi,V_inf,rho,mu,AoA,Cl0,M_inf)
% Initialize the variable to store spanwise drag
D = NaN(1,y_panels);

% Compute the drag of each spanwise section
parfor i = 1:y_panels
    % Determine the mean chord and spanwise length of the section
    chord_l = abs(wing_calc.panel_x(end,i) - wing_calc.panel_x(1,i));
    chord_r = abs(wing_calc.panel_x(end,i+1) - wing_calc.panel_x(1,i+1));
    chord_section = (chord_l + chord_r)/2;
    length_section = abs(wing_calc.panel_y(1,i+1)-wing_calc.panel_y(1,i));

    % Determine the induced drag of the section
    induced_drag_section = sum(dDi(:,i))/length_section;

    % Compute the induced drag coefficient, and determine the
    % incompressible coefficient
    Cdi = induced_drag_section/(0.5*rho*V_inf^2*chord_section)*sqrt(1-M_inf.^2);

    % Compute the effective angle of attack that the airfoil experiences
    a_eff = calc_effective_AoA(AoA(i),0,Cdi,Cl0);

    % Compute the parasite drag of the airfoil
    [d,~] = airfoil_solver_viscous(foil,chord_section,a_eff,V_inf,rho,mu,M_inf);

    % Add it to the total parasite drag
    D(i)= d*length_section;
end

% Compute the total parasitic drag
Dp = sum(D,'all');

end
```

```
% airfoil_solver_inviscid.m
%
% Description:
%   This function encapsulates the inviscid airfoil solver, used in
%   conjuntion with the viscous airfoil solver to determine the parasitic
%   drag of the wing.
%
% Written by:   Julian Bardin
% Date:         2021-03-17
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin FPages 303-305
%

function [CP,Qt,foil] = airfoil_solver_inviscid(foil,AoA,V_inf)
[foil.x_coloc,foil.z_coloc] = calc_airfoil_coloc_coords(foil);
[foil.a_panels] = calc_airfoil_panel_angle(foil);

[foil.x_norm,foil.z_norm] = calc_panel_norm_2D(foil.a_panels);
[foil.x_tang,foil.z_tang] = calc_panel_tang_2D(foil.a_panels);

[A,B] = calc_airfoil_influence_coeffs(foil);
Q_inf = calc_vel_vector(V_inf,AoA);
RHS = calc_airfoil_RHS(foil,Q_inf);
gamma = A\RHS;
[CP,Qt] = calc_airfoil_CP(foil,Q_inf,V_inf,B,gamma);
end
```

```
% airfoil_solver_viscous.m
%
% Description:
%   This function encapsulates the viscous-inviscid coupled solver for the
%   airfoil, used to predict the sectional drag generated by an airfoil.
%   The problem is solved by interatively adjusting the airfoil geometry
%   sent to the inviscid solver in order to account for the boundary layer
%   displacement thickness.
%
% Written by:   Julian Bardin
% Date:         2021-03-20
%

function [d,zBL] = airfoil_solver_viscous(foil,chord,AoA,V_inf,rho,mu,M_inf)
% Initialize the buffer variables
break_cond_0 = 0;
break_cond_1 = 9E20;
zBL_old = foil.z_coords;
foil_BL = foil;

% Iterate until the airfoil simulation has converged
while abs(break_cond_0 - break_cond_1) > 0.001
    % Update the value for the previous iteration's break condition
    break_cond_0 = break_cond_1;

    % Run the inviscid solver
    [~,Qt,~] = airfoil_solver_inviscid(foil_BL,AoA,V_inf);

    % Find the index of the stagnation point
    idx_stag = find_stagnation_point(Qt);

    % Split up the airfoil into two have surfaces based on the stagnation point
    [x_1,z_1,Qt_1,x_2,z_2,Qt_2] = split_airfoil(foil,Qt,idx_stag);
    s_1 = calc_curve_dist(x_1*chord,z_1*chord);
    s_2 = calc_curve_dist(x_2*chord,z_2*chord);

    % Compute the skin friction and boundary layer displacement thickness
    [del_str_1,~,Cf_1] = halffoil_solver_viscous(s_1,Qt_1,rho,mu,M_inf,V_inf);
    [del_str_2,~,Cf_2] = halffoil_solver_viscous(s_2,Qt_2,rho,mu,M_inf,V_inf);

    % Compute the new airfoil z values based on the displacement thickness
    zBL_1 = compute_new_z_from_disp(x_1,z_1,-del_str_1/chord);
    zBL_2 = compute_new_z_from_disp(x_2,z_2,del_str_2/chord);

    % Merge the upper and lower z values to create the new airfoil geometry
    zBL = [fliplr(zBL_1) zBL_2];

    % Update the airfoil
    foil_BL.z_coords = zBL;

    % Determine the break condition through analysing the boundary layer
    % near the middle of the airfoil
    l1 = length(zBL_1);
    break_cond_1 = sum(abs(zBL_1(floor(l1/3):ceil(2*l1/3)) - zBL_old(floor(l1/3):ceil(2*l1/3))));

    % Store this current boundary layer as the "old" value for comparison
    % in the next loop
    zBL_old = zBL;
end

% Compute the sectional drag of the airfoil and apply compressiblity
% correction
d_1 = calc_halffoil_drag(rho,Qt_1,Cf_1,s_1);
d_2 = calc_halffoil_drag(rho,Qt_2,Cf_2,s_2);
d = d_1 + d_2;
end
```

```
% calc_airfoil_Cl.m
%
% Description:
%   This function is used to compute lift coefficient of an airfoil based
%   on its geometry and pressure distribution.
%
% Written by:   Julian Bardin
% Date:         2021-03-20
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Page 283

function Cl = calc_airfoil_Cl(foil,CP)
% Extract the organize the airfoil geometry into two row vectors
x_coloc = foil.x_coloc;
z_coloc = foil.z_coloc;

% Determine the number of panels
num_panels = length(CP);

% Go through the pressure coefficients and sum up the contributions to the
% lift coefficient
Cl = 0;
for i = 1:num_panels-1
    % Calculate the distance between colocation points
    dist_coloc = sqrt((z_coloc(i) - z_coloc(i+1))^2 + (x_coloc(i) - x_coloc(i+1))^2);

    % Add the contribution to the lift coefficient
    Cl = Cl - CP(i)*dist_coloc*cos(foil.a_panels(i));
end
end
```

```
% calc_airfoil_coloc_coords.m
%
% Description:
%   This function computes the colocation points of all the panels the
%   comprise the airfoil. These colocation points are located at the
%   midpoints of all the panels.
%
% Written by:   Julian Bardin
% Date:         2021-02-16
%

function [x,z] = calc_airfoil_coloc_coords(foil)
% Extract the organize the airfoil geometry into two row vectors
x_foil = foil.x_coords;
z_foil = foil.z_coords;

% Initialize the output matrices for the colocation points
x = NaN(1,length(x_foil)-1);
z = NaN(1,length(x_foil)-1);

% Go through the airfoil geometry and place the colocation points at the
% midpoint of each panel. As all panels are linear, the midpoints are
% merely the average of the coordinates of the panel vertices
for i = 1:length(x)
    x(i) = (x_foil(i) + x_foil(i+1))/2;
    z(i) = (z_foil(i) + z_foil(i+1))/2;
end
end
```

```
% calc_airfoil_CP.m
%
% Description:
%   This function is used to compute the pressure coefficient distribution
%   along the airfoil surface. It also computes and returns the tangential
%   velocity for each panel
%
% Written by:   Julian Bardin
% Date:         2021-03-20
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Page 305
%

function [CP,Qt] = calc_airfoil_CP(foil,Q_inf,V_inf,B,gamma)
% Extract the organize the airfoil geometry into two row vectors
x_tang = foil.x_tang;
z_tang = foil.z_tang;

% Convert the 3D velocity vector to 2D in the x-z plane
Q_inf = [Q_inf(1) Q_inf(3)];

% Determine the number of panels
num_panels = length(gamma)-1;

% Initialize the output variables
CP = NaN(1,num_panels);
Qt = NaN(1,num_panels);

% Multiply B by gamma to determine the induced tangential velocities. Note
% that the bottom row of B is NaNs because B is one column and row larger
% than the number of panels
Q_ind = B*gamma;

% Calculate the tangential velocity and pressure coefficient at each
% colocation point
for i = 1:num_panels
    Qt(i)  = Q_ind(i) + dot(Q_inf,[x_tang(i) z_tang(i)]);
    CP(i) = 1-(Qt(i)/V_inf)^2;
end

end
```

```
% calc_airfoil_influence_coeffs.m
%
% Description:
%   This function is responsible for calculating the influence coefficients
%   used in the computation of inviscid airfoil performance. These
%   coefficients are stored in the A matrix. And additional B matrix is
%   also returned, storing the nodal tangential speed at unit vorticity.
%
% Written by:   Julian Bardin
% Date:         2021-02-16
%
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Eq 11.103, 11.103b,
%   11.103c, 11.104, and 11.105
%

function [A,B] = calc_airfoil_influence_coeffs(foil)
% Extract the panel geometry
x_foil = foil.x_coords;
z_foil = foil.z_coords;
a_foil = foil.a_panels;

% Extract the colocation points
x_coloc = foil.x_coloc;
z_coloc = foil.z_coloc;

% Extract the normal vectors
x_norm  = foil.x_norm;
z_norm  = foil.z_norm;

% Extract the tangent vectors
x_tang  = foil.x_tang;
z_tang  = foil.z_tang;


% Determine the number colocation points
num_c = length(z_coloc);

% Initialize the output matrix
A = NaN(num_c+1);
B = NaN(num_c+1);

% Go through the various points
% For each colocation points i
for i = 1:num_c
    % Define the colocation point as point p
    p = [x_coloc(i) z_coloc(i)];

    % Initialize variables to store the induced velocities
    V_ua = NaN(1,num_c);
    V_wa = NaN(1,num_c);
    V_ub = NaN(1,num_c);
    V_wb = NaN(1,num_c);

    % Compute the induced velocity of each vortex on this colocation points
    for j = 1:num_c
        % Define the two verticies of the panel and p1 and p2
        p1 = [x_foil(j)   z_foil(j)];
        p2 = [x_foil(j+1) z_foil(j+1)];

        % Compute the panel-reference-frame values
        [p_p,p1_p,p2_p,th1,th2,r1,r2] = determine_panel_frame(p,p1,p2,a_foil(j));

        % Compute the induced velocities
        if i == j
            [V_ua(j),V_wa(j),V_ub(j),V_wb(j)] = calc_velocity_p_ij(p_p,p1_p,p2_p,a_foil(j));
        else
            [V_ua(j),V_wa(j),V_ub(j),V_wb(j)] =
calc_velocity_p(p_p,p1_p,p2_p,th1,th2,r1,r2,a_foil(j));
        end
    end
```

```
    % Compute the influence coefficients for j = 1
    j = 1;
    A(i,j) = dot([V_ua(j) V_wa(j)],[x_norm(i) z_norm(i)]);
    B(i,j) = dot([V_ua(j) V_wa(j)],[x_tang(i) z_tang(i)]);

    % Compute the influence coefficents for j = 2:N
    for j = 2:num_c
        % Extract the velocity vector
        V = [V_ua(j)+V_ub(j-1) V_wa(j)+V_wb(j-1)];

        % Set the influence coefficient
        A(i,j) = dot(V,[x_norm(i) z_norm(i)]);
        B(i,j) = dot(V,[x_tang(i) z_tang(i)]);
    end

    % Compute the influence coefficient for N+1
    j = num_c+1;
    A(i,j) = dot([V_ub(j-1) V_wb(j-1)],[x_norm(i) z_norm(i)]);
    B(i,j) = dot([V_ub(j-1) V_wb(j-1)],[x_tang(i) z_tang(i)]);
end

% Set the bottom row of the matrix to be the Kutta condition
row_bottom = zeros(1,num_c+1);
row_bottom(1)   = 1;
row_bottom(end) = 1;
A(end,:) = row_bottom;
end

function [u_a,w_a,u_b,w_b] = calc_velocity_p(p_p,p1_p,p2_p,th1,th2,r1,r2,alpha_i)
% Extract the point information
x  = p_p(1);
z  = p_p(2);
x1 = p1_p(1);
x2 = p2_p(1);

% Calculate the three speeds in the panel reference frame
up_a = -(z*log(r2/r1)+x*(th2-th1)-x2*(th2-th1))/(2*pi*x2);
up_b =  (z*log(r2/r1) + x*(th2-th1))/(2*pi*x2);
wp_a = -(x2-z*(th2-th1) - x*log(r1/r2) + x2*log(r1/r2))/(2*pi*x2);
wp_b =  (x2-z*(th2-th1) - x*log(r1/r2))/(2*pi*x2);

% Calculate the rotation matrix
rot_p2g = [cos(-alpha_i) sin(-alpha_i) ; -sin(-alpha_i) cos(-alpha_i)];

% Rotate the calculated velocities
Vp_a = [up_a ; wp_a];
Vp_b = [up_b ; wp_b];
V_a  = rot_p2g*Vp_a;
V_b  = rot_p2g*Vp_b;

% Extract the rotated velocities for outputting
u_a = V_a(1);
w_a = V_a(2);
u_b = V_b(1);
w_b = V_b(2);
end

function [u_a,w_a,u_b,w_b] = calc_velocity_p_ij(p_p,p1_p,p2_p,alpha_i)
% Extract the point information
x  = p_p(1);
z  = p_p(2);
x1 = p1_p(1);
x2 = p2_p(1);

% Calculate the three speeds in the panel reference frame
up_a = -0.5*(x-x2)/x2;
up_b =  0.5*x/x2;
wp_a = -1/(2*pi);
wp_b =  1/(2*pi);
```

```
% Calculate the rotation matrix
rot_p2g = [cos(-alpha_i) +sin(-alpha_i) ; -sin(-alpha_i) cos(-alpha_i)];

% Rotate the calculated velocities
Vp_a = [up_a ; wp_a];
Vp_b = [up_b ; wp_b];
V_a  = rot_p2g*Vp_a;
V_b  = rot_p2g*Vp_b;

% Extract the rotated velocities for outputting
u_a = V_a(1);
w_a = V_a(2);
u_b = V_b(1);
w_b = V_b(2);

end
```

```
% calc_airfoil_RHS.m
%
% Description:
%   This function is used to compute the RHS vector, required to solve the
%   vorticities of the airfoil approximation.
%
% Written by:   Julian Bardin
% Date:         2021-02-16
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Eq 11.103, 11.103b,
%   11.103c, 11.104, and 11.105
%

function RHS = calc_airfoil_RHS(foil,Q_inf)
% Extract the organize the airfoil geometry into two row vectors
x_foil = foil.x_coords;
a_foil = foil.a_panels;

% Extract the normal vectors
x_norm  = foil.x_norm;
z_norm  = foil.z_norm;

% Convert the 3D velocity vector to 2D in the x-z plane
Q_inf = [Q_inf(1) Q_inf(3)];

% Initialize the RHS vector
RHS = NaN(length(x_foil),1);

% Go through all the panels and calculate the RHS vector
for i = 1:length(x_foil)-1
    RHS(i) = dot(Q_inf,[sin(a_foil(i)) -cos(a_foil(i))]);
end

% Add the zero at the end of the RHS vector for the Kutta condition
RHS(end) = 0;

end
```

```
% calc_effective_AoA
%
% Description:
%   This function is used to approximately the effective angle of attack of
%   the cross-section based on the induced drag.
%
% ** ANGLES ARE IN DEGREES
%

function a_eff = calc_effective_AoA(a_global,e_twist,Cdi,Cl0)
% Convert the input angles into radians
a_global = a_global*pi/180;
e_twist = e_twist*pi/180;

% Define the function that will be solved
f = @(a_eff) (2*pi*a_eff + Cl0)*sin(a_global + e_twist-a_eff) - Cdi;

% Solve for the effective angle of attack
a_eff = fzero(f,0);

% Convert the effective angle of attack into degrees
a_eff = a_eff*180/pi;

% Convert the angle to be between +90 and -90
while 1
    if a_eff < -90
        a_eff = a_eff + 180;
    elseif a_eff > 90
        a_eff = a_eff - 180;
    else
        break
    end
end
end
```

```
% calc_airfoil_drag
%
% Description:
%   This function is used to compute the drag generated by one half of the
%   airfoil.
%
% Written by:   Julian Bardin
% Date:         2021-03-20
%

function d = calc_halffoil_drag(rho,Qt,Cf,s)
% Initailize d to zero
d = 0;

% Go through all the points and compute the drag of the section, and add it
% to the overall drag
for i = 2:length(s)
    d = 0.5*rho*(Qt(i-1))^2*Cf(i)*(s(i)-s(i-1)) + d;
end
end
```

```
% calc_influence_coeffs.m
%
% Description:
%   This function calculates the lift and induced drag influence
%   coefficients. The lift influence coefficients are calculated using
%   vortex rights of unit vorticity, and the inverse of the matrix (A)
%   can be used later to solve for the vorticities.
%
% Written by:   Julian Bardin
% Date:         2021-02-13
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Fig. 12.13
%


function [A,B] = calc_influence_coeffs(wing_calc)
% Determine how many panels are present in the wing_calc object
dims = size(wing_calc.panel_x);
x_panels = dims(1)-1;
y_panels = dims(2)-1;

% Determine how many colocation points there are
x_colocs = x_panels;
y_colocs = y_panels;

% Predefine A and B as square matrices with x_panels*y_panels by
% x_panels*y_panels size
A = NaN(x_panels*y_panels);
B = NaN(x_panels*y_panels);

% Extract the ring locations
xr = wing_calc.ring_x;
yr = wing_calc.ring_y;
zr = wing_calc.ring_z;

% Extract the colocation points
xc = wing_calc.coloc_x;
yc = wing_calc.coloc_y;
zc = wing_calc.coloc_z;

% Extract the norms
xn = wing_calc.norm_x;
yn = wing_calc.norm_y;
zn = wing_calc.norm_z;

% Go through the various colocation points
parfor K = 1:x_colocs*y_colocs
    % Convert hte K value into its corresponin i and j
    [i_c,j_c] = K2ij(K,y_panels);

    % Define buffer variables for a row of A and B
    A_row = NaN(1,x_panels*y_panels);
    B_row = NaN(1,x_panels*y_panels);

    % Define the colocation point
    c           = [xc(i_c,j_c)  yc(i_c,j_c) zc(i_c,j_c)];
    c_mirror    = [xc(i_c,j_c) -yc(i_c,j_c) zc(i_c,j_c)];

    % Define the normal vector
    norm = [xn(i_c,j_c)  yn(i_c,j_c) zn(i_c,j_c)];

    % Go through the various vortex rings
    L = 1;
    for i = 1:x_panels
        for j = 1:y_panels

            % Define the points for the vortex ring
            [p1,p2,p3,p4] = calc_ring_corners(i,j,xr,yr,zr);

            % Calculate the induced velocities
            [q1, qt1] = vortex_ring(c       ,p1,p2,p3,p4,1.0);
```

```
            [q2, qt2] = vortex_ring(c_mirror,p1,p2,p3,p4,1.0);

            % Calculate the net induced velocity, and the component only from
            % the vortex lines parallel to the flow
            q  = add_mirrored_q(q1,q2);
            qt = add_mirrored_q(qt1,qt2);

            % Check if its the trailing edge
            if i == x_panels
                % Define the vortex ring points for the wake
                [p1,p2,p3,p4] = calc_ring_corners(i+1,j,xr,yr,zr);

                % Calculate the induced velocities of the wake
                [q3, qt3] = vortex_ring(c       ,p1,p2,p3,p4,1.0);
                [q4, qt4] = vortex_ring(c_mirror,p1,p2,p3,p4,1.0);

                % Add the influence of the wake to the induced velocity
                q  = q  + add_mirrored_q(q3,q4);
                qt = qt + add_mirrored_q(qt3,qt4);
            end

            % Calculate the influence coefficients
            A_row(L) = dot(q, norm);
            B_row(L) = dot(qt,norm);

            % Increment the L counter for the next iteration
            L = L + 1;
        end
    end

    % Insert the influence coefficients into the main matrix
    A(K,:) = A_row;
    B(K,:) = B_row;
end
end

% This function determines the location vectors for the four corners of the
% vortex ring.
function [p1,p2,p3,p4] = calc_ring_corners(i,j,xr,yr,zr)
p1 = [xr(i,j)      yr(i,j)     zr(i,j)];
p2 = [xr(i,j+1)    yr(i,j+1)   zr(i,j+1)];
p3 = [xr(i+1,j+1)  yr(i+1,j+1) zr(i+1,j+1)];
p4 = [xr(i+1,j)    yr(i+1,j)   zr(i+1,j)];
end

% This function adds the mirrored induced velocities
function q = add_mirrored_q(q1,q2)
q(1)  = q1(1)  + q2(1);
q(2)  = q1(2)  - q2(2);
q(3)  = q1(3)  + q2(3);
end
```

```
% calc_panel_induced_drag
%
% Description:
%   This function is used to calculate the induced drag produced by each
%   individual panel, as well as the overall total induced drag generated
%   by the wing.
%
% Written by:   Julian Bardin
% Date:         2021-02-13
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Eq 12.27 and 12.27a
%

function [Di,dDi] = calc_panel_induced_drag(wing_calc,Gamma,w,M_inf,rho)
% Determine how many panels are present in the wing_calc object
dims = size(wing_calc.panel_x);
x_panels = dims(1)-1;
y_panels = dims(2)-1;

% Initialize the matrix dL which will store the lift acting on each panel
dDi = NaN(x_panels,y_panels);

% Extract the panel dimensions
yp = wing_calc.panel_y;

% Go through all the panels and calculate the lift force acting on each
% panel
for j = 1:y_panels
    % Calculate the panel width
    dy = abs(yp(1,j) - yp(1,j+1));

    % Determine the K index for the vorticity vector for the leading edge
    % panel
    K   = ij2K(1,j,y_panels);

    % Calculate the component for the panel induced drag which is only a
    % function of induced velocity (w) and vorticity. Leading edge panel
    % only.
    dDi(1,j) = Gamma(K).*w(K).*dy;

    % For the non-leading edge panels
    for i = 2:x_panels
        % Determine the K index for the vorticity vector
        K   = ij2K(i,j,y_panels);
        K_1 = ij2K(i-1,j,y_panels);

        % Calculate the component for the panel induced drag which is only a
        % function of induced velocity (w) and vorticity.
        dDi(i,j) = (Gamma(K) - Gamma(K_1)).*w(K).*dy;
    end
end

% Multiply by density to determine the drag, then correct for
% compressibility.
dDi = -rho./sqrt(1-M_inf.^2).*dDi;

% Calculate the total induced drag
Di = sum(dDi,'all');
end
```

```
% calc_panel_lift.m
%
% Description:
%   This function is responsible for calculating the lift distribution and
%   total lift of the wing, given the atmospheric conditions and the
%   vorticity vector.
%
% Written by:   Julian Bardin
% Date:         2021-02-13
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Eq 12.25 and 12.25a
%


function [L,dL] = calc_panel_lift(wing_calc,Gamma,V_inf,M_inf,rho)
% Determine how many panels are present in the wing_calc object
dims = size(wing_calc.panel_x);
x_panels = dims(1)-1;
y_panels = dims(2)-1;

% Initialize the matrix dL which will store the lift acting on each panel
dL = NaN(x_panels,y_panels);

% Extract the panel dimensions
yp = wing_calc.panel_y;

% Go through all the panels and calculate the lift force acting on each
% panel
for j = 1:y_panels
    % Calculate the panel width
    dy = abs(yp(1,j) - yp(1,j+1));

    % Determine the K index for the vorticity vector
    K   = ij2K(1,j,y_panels);

    % Calculate the component for the panel left which is only a function
    % of vorticity and panel width
    dL(1,j) = Gamma(K).*dy;

    % For the non-leading edge panels
    for i = 2:x_panels
        % Determine the K index for the vorticity vector
        K   = ij2K(i,j,y_panels);
        K_1 = ij2K(i-1,j,y_panels);

        % Calculate the component for the panel left which is only a function
        % of vorticity and panel width
        dL(i,j) = (Gamma(K) - Gamma(K_1)).*dy;

    end
end

% Multiply by density, V_inf, and correct for compressility to calculate
% the lift for each panel
dL = rho.*V_inf./sqrt(1-M_inf.^2).*dL;

% Calculate the total lift
L = sum(dL,'all');
end
```

```
% calc_panel_norm.m
%
% Description:
%   This function calculates the normal vector of a panel defined by four
%   points p1 through p4. The returned vector is of unit length.
%   Additionally, p1 refers to the forward-leftmost point, and 2 through 4
%   are clockwise from there.
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Eq. 12.20
%


function n = calc_panel_norm(p1,p2,p3,p4)
% Define the two diagonal vectors
r31 = p3 - p1;
r24 = p2 - p4;

% Calculate the cross product in the same direction as the normal
cross_n = cross(r31,r24);

% Calculate the normal vector
n = cross_n./norm(cross_n);
end
```

```
% calc_panel_norm_2D.m
%
% Description
%   This function calculates the panel norms placed at each colocation
%   point for an airfoil.
%
% Written by:   Julian Bardin
% Date:         2021-02-16
%

function [x,z] = calc_panel_norm_2D(alpha)
% Initialize the output matrices for the normal vector
x = NaN(1,length(alpha));
z = NaN(1,length(alpha));

% Go through all the panels and compute the normals
for i = 1:length(alpha)
    x(i) = -sin(alpha(i));
    z(i) = cos(alpha(i));
end
end
```

```
% calc_panel_tang_2D.m
%
% Description
%   This function calculates the panel norms placed at each colocation
%   point for an airfoil.
%
% Written by:   Julian Bardin
% Date:         2021-02-16
%

function [x,z] = calc_panel_tang_2D(alpha)
% Initialize the output matrices for the normal vector
x = NaN(1,length(alpha));
z = NaN(1,length(alpha));

% Go through all the panels and compute the normals
for i = 1:length(alpha)
    x(i) = cos(alpha(i));
    z(i) = sin(alpha(i));
end
end
```

```
% calc_RHS.m
%
% Description:
%   This function calculates the RHS vector, essentially in calculating the
%   vorticity distribution of the wing.
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Eq. 12.24
%

function RHS = calc_RHS(wing_calc,Q_inf)
% Determine how many panels are present in the wing_calc object
dims = size(wing_calc.panel_x);
x_panels = dims(1)-1;
y_panels = dims(2)-1;

% Extract the norms
xn = wing_calc.norm_x;
yn = wing_calc.norm_y;
zn = wing_calc.norm_z;

% Initialize the RHS vector
RHS = NaN(x_panels*y_panels,1);

% Go through all the colocation points and calculate the value for the RHS
% VLM vector
K = 1;
for i = 1:x_panels
    for j = 1:y_panels
         % Calculate the RHS value
         RHS(K) = -dot(Q_inf,[xn(i,j)  yn(i,j) zn(i,j)]);

         % Increment the K counter
         K = K + 1;
     end
end
end
```

```
% determine_coloc_coordinates
%
% Description:
%   This function is responsible for calculating the locations of the
%   colocation points for each panel
%
% Written by:   Julian Bardin
% Date:         2021-02-09
%

function [x,y,z] = determine_coloc_coordinates(wing_calc)
% Determine how many panels are present in the wing_calc object
dims = size(wing_calc.panel_x);
x_panels = dims(1)-1;
y_panels = dims(2)-1;

% Extract the x and y coordinates
x_coords = wing_calc.panel_x;
y_coords = wing_calc.panel_y;
z_coords = wing_calc.panel_z;

% Initialize the coordinate matrices
x = NaN(x_panels,y_panels);
y = NaN(x_panels,y_panels);
z = NaN(x_panels,y_panels);

% Go through each panel and determine colocation point
for j = 1:y_panels
     for i = 1:x_panels
        % Get the points for the vertices of this panel
        x_v = x_coords(i:i+1,j:j+1);
        y_v = y_coords(i:i+1,j:j+1);
        z_v = z_coords(i:i+1,j:j+1);

        % Since y is straight, the y-coordinate can be calculated from the
        % panel's leading edge
        y_point = (y_v(1,1) + y_v(1,2))/2;

        % Get the 3/4 lines for the left and right side
        x_34l = x_v(1,1) + 0.75*abs(x_v(1,1) - x_v(2,1));
        x_34r = x_v(1,2) + 0.75*abs(x_v(1,2) - x_v(2,2));

        % Get the z values at the above left and right x points
        z_34l = interp1(x_v(:,1),z_v(:,1),x_34l);
        z_34r = interp1(x_v(:,2),z_v(:,2),x_34r);

        % Since the y-coordinate is a midpoint, so too will these be the
        % midpoint between the left and right 3/4 points
        x_point = (x_34l + x_34r)/2;
        z_point = (z_34l + z_34r)/2;

        % Set the colocation points
        x(i,j) = x_point;
        y(i,j) = y_point;
        z(i,j) = z_point;
    end
end



end
```

```
% determine_coloc_norms.m
%
% Description:
%   This function calculates the panel normals for wing, which during later
%   analysis are treated as being placed at the colocation point.
%
% Written by:   Julian Bardin
% Date:         2021-02-13
%


function [x,y,z] = determine_coloc_norms(wing_calc)
% Determine how many panels are present in the wing_calc object
dims = size(wing_calc.panel_x);
x_panels = dims(1)-1;
y_panels = dims(2)-1;

% Initialize the coordinate matrices
x = NaN(size(x_panels));
y = NaN(size(y_panels));
z = NaN(size(x_panels));

% Extract the panel locations
xp = wing_calc.panel_x;
yp = wing_calc.panel_y;
zp = wing_calc.panel_z;

% Go through each panel and determine normal for that panel, which for
% later ananlysis is located at the colocation points
for j = 1:y_panels
     for i = 1:x_panels
         % Define the points
         p1 = [xp(i,j)      yp(i,j)     zp(i,j)];
         p2 = [xp(i,j+1)    yp(i,j+1)   zp(i,j+1)];
         p3 = [xp(i+1,j+1)  yp(i+1,j+1) zp(i+1,j+1)];
         p4 = [xp(i+1,j)    yp(i+1,j)   zp(i+1,j)];

         % Calculate the normal
         n = calc_panel_norm(p1,p2,p3,p4);

         % Store the vector lengths for the normal
         x(i,j) = n(1);
         y(i,j) = n(2);
         z(i,j) = n(3);
     end
end

end
```

```
% determine_panel_area.m
%
% Description:
%   This function calculates the area of a panel in 3D space.
%
% Written by:   Julian Bardin
% Date:         2021-02-12
%


function A = determine_panel_area(wing_calc)
% Determine how many panels are present in the wing_calc object
dims = size(wing_calc.panel_x);
x_panels = dims(1)-1;
y_panels = dims(2)-1;

% Extract the panel locations
xp = wing_calc.panel_x;
yp = wing_calc.panel_y;
zp = wing_calc.panel_z;

% Initialize the area matrix
A = NaN(size(x_panels));

% Go through all the panels and calculate their area.
for j = 1:y_panels
    for i = 1:x_panels
        % Define the points for the panel
         p1 = [xp(i,j)      yp(i,j)     zp(i,j)];
         p2 = [xp(i,j+1)    yp(i,j+1)   zp(i,j+1)];
         p3 = [xp(i+1,j+1)  yp(i+1,j+1) zp(i+1,j+1)];
         p4 = [xp(i+1,j)    yp(i+1,j)   zp(i+1,j)];

         % Calculate the area by splitting the shape into two triangles
         A124 = calc_area_triangle(p1,p2,p4);
         A243 = calc_area_triangle(p2,p4,p3);

         % Store the area.
         A(i,j) = A124 + A243;
    end
end
end

% This function calculates the area of a triangle in 3d space
function A = calc_area_triangle(p1,p2,p3)
% Define the two position vectors
r12 = p1 - p2;
r32 = p3 - p2;

% Calculate the area
A = 0.5*norm(cross(r12,r32));
end
```

```
% determine_panel_coordinates.m
%
% Description:
%   This function is responsible for generating the coordinates for the
%   vertices of the panels which will be used for VLM computation
%
% Written by:   Julian Bardin
% Date:         2021-02-09
%

function [x,y,z] = determine_panel_coordinates(wing_param,x_panels,y_panels)
% Get the leading and trailing edge points
[x_LE,y_LE,x_TE,y_TE] = gen_planform_LE_TE(wing_param);

% Of the y-panels (spanwise direction), determine how many as being
% allocated to the left of the kink and to the right of the kink
y_panels_left = floor(y_panels*wing_param.dr_kink);
if y_panels_left < 1 && wing_param.dr_kink ~= 0
    y_panels_left = 1;
end
y_panels_right = y_panels - y_panels_left;

% Initialize the coordinate matrices
x = NaN(x_panels+1,y_panels+1);
y = NaN(x_panels+1,y_panels+1);
z = NaN(x_panels+1,y_panels+1);

% Determine the y coordinates
y_lines_l = linspace(0,wing_param.semi_span*wing_param.dr_kink,y_panels_left+1);
y_lines_r =
linspace(wing_param.semi_span*wing_param.dr_kink,wing_param.semi_span,y_panels_right+1);
y_lines = [y_lines_l y_lines_r(2:end)];

% Populate the coordinate matrices
for j = 1:y_panels+1
    % Set the y-coordinates for the chord
    y(:,j) = y_lines(j);

    % Determine the points that comprise the planview chord. Note that
    % x_LE, y_LE, etc are swapped from that used for the wing
    % dimensions.
    chord_x = [interp1(x_LE,y_LE,y_lines(j)) interp1(x_TE,y_TE,y_lines(j))];
    chord_len = abs(chord_x(1) - chord_x(2));

    % Split up the chord into the requested number of points
    x_points = linspace(chord_x(1),chord_x(2),x_panels+1);

    % Calculate the z-height based on the airfoil camber
    z_camber = calc_mean_camber(wing_param.airfoil,linspace(0,1,x_panels+1))*chord_len;

    for i = 1:x_panels+1
        x(i,j) = -x_points(i);
        z(i,j) = z_camber(i);
    end
end
end
```

```
% determine_panel_frame.m
%
% Description:
%   This function is used to compuite the points, angles, and displacements
%   within the panel reference frame.
%
% Written by:   Julian Bardin
% Date:         2021-02-16
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Fig 11.29
%


function [p_p,p1_p,p2_p,th1,th2,r1,r2] = determine_panel_frame(p,p1,p2,alpha_i)
% Define the rotation matrices
rot_g2p = [cos(alpha_i) sin(alpha_i) ; -sin(alpha_i) cos(alpha_i)];

% Since p1 is used as the origin for the panel reference frame, set its
% coordinates to [0,0]
p1_p = [0 0];

% Calculate the p2_p and p_p location in the panel reference frame using
% the rotation matrix.
p2_p = (rot_g2p*(p2-p1)')';
p_p  = (rot_g2p*(p-p1)')';

% Calculate the magnitude of the displacement vectors r1 and r2 which point
% to the given arbitrary point p
r1 = norm(p_p);
r2 = norm(p_p-p2_p);

% Calculate the theta angles using arctan (not arctan2)
th1 = atan2(p_p(2),p_p(1));
th2 = atan2(p_p(2),(p_p(1) - p2_p(1)));
end
```

```
% determine_ring_coordinates
%
% Description:
%   This function is responsible for calculating the vertex coordinates of
%   the vortex rings which will be used for VLM computation of wing lift
%   and inducted drag performance
%
% Written by:   Julian Bardin
% Date:         2021-02-09
%

function [x,y,z] = determine_ring_coordinates(wing_calc)
% Determine how many panels are present in the wing_calc object
dims = size(wing_calc.panel_x);
x_panels = dims(1)-1;
y_panels = dims(2)-1;

% Initialize the coordinate matrices
x = NaN(size(x_panels));
z = NaN(size(x_panels));

% Since, y-coordiantes don't change, just use the same y-coordinate matrix
% from the wing_calc object. We do need to add an additional row to
% compensate for the zero-strength vortex data points
y = [wing_calc.panel_y ; wing_calc.panel_y(1,:)];


for j = 1:y_panels+1 % For each chord
    % Extract the point which fall along this chord
    x_coords = [wing_calc.panel_x(:,j)];
    z_coords = [wing_calc.panel_z(:,j)];

    % Going along the chord, adjust x positions to place rings on the
    % quarder line of each panel
    for i = 1:x_panels
        x(i,j) = x_coords(i)+0.25*abs(x_coords(i)-x_coords(i+1));
        z(i,j) = interp1(x_coords,z_coords,x(i,j));
    end

    % Account for the trailing edge zero-strength vortex line
    x(i+1,j) = x_coords(end) + 0.25*abs(x_coords(end-1)-x_coords(end));
    z(i+1,j) = 0;

    % Add an extra point at 1E6 for "infinity"
    x(i+2,j) = 1E6;
    z(i+2,j) = 0;

end


end
```

```
% halffoil_solver_viscous.m
%
% Description:
%   This function computes the viscous boundary layer quantities of the
%   airfoil. This include consideration for the laminar potion and
%   turbulent portion, with a transition in between.
%
% Based on: An Integral Boundary Layer Direct Method... by G. Fijiwara and
% N. Nguyen
% Based on: Stanford University Viscous Flow Presentation
%
% Written by:   Julian Bardin
% Date:         2021-03-20
%
%

function [del_str,theta,Cf] = halffoil_solver_viscous(s,ue,rho,mu,M_inf,V_inf)
% Determine the number of points
num_points = length(s);

% Initialize the output vectors
del_str = NaN(1,num_points); % Boundary layer displacement thickness
theta   = NaN(1,num_points); % Boundary layer momentum thickess
Cf      = NaN(1,num_points); % skin friction coefficient

% Compute the free-stream speed of sound
if M_inf ~= 0
    a_inf = V_inf/M_inf;
else
    a_inf = 1E6;
end

% Go through the laminar points
H = 0;
for i = 1:num_points
    % Compute the local reynolds number
    Re_l = rho*ue(i)/mu;

    % Calculate the momentum thickness
    theta_sqrd = laminar_momentum_thickness(mu,ue,s,i);
    theta(i) = sqrt(theta_sqrd);
    if ~isreal(theta(i))
        error('Computation error');
    end

    % Calculate the momentum reynolds number
    Re_th = Re_l*theta(i);

    % If the flow has transitioned, exit this loop
    if check_if_transition(Re_l,s(i),Re_th)
%         disp('Transition')
        break;
    end

    % Compute the speed gradient
    if i ~= 1
        spd_grad = (ue(i) - ue(i-1))/(s(i) - s(i-1));
    else
        spd_grad = (ue(2) - ue(1))/(s(2) - s(1));
    end

    % Compute the value of lambda
    lambda = theta_sqrd*rho/mu*spd_grad;

    % Compute the values of H and l
    [H,l] = laminar_thwaites_approximations(lambda);

    % Compute the displacement thickness
    del_str(i) = H*theta(i);

    % Compute the friction coefficient
```

```
    Cf(i) = 2*l/Re_th*(1+0.2*(ue(i)/a_inf)^2);
end

% Override the friction coefficient at the first point to zero, since the
% computed value only exists due to the algorithm structure
Cf(1) = 0;

% Decrement i to redo that point as it is turbulent rather than laminar
i = i-1;

% Go through turbulent points
if i < num_points
    if i == 1
        for i = 1:2
            % Compute the local reynolds number
            Re_l = rho*ue(i)/mu;

            % Calculate the momentum thickness
            theta_sqrd = laminar_momentum_thickness(mu,ue,s,i);
            theta(i) = sqrt(theta_sqrd);
            if ~isreal(theta(i))
                error('Computation error');
            end

            % Calculate the momentum reynolds number
            Re_th = Re_l*theta(i);

            % Compute the speed gradient
            if i ~= 1
                spd_grad = (ue(i) - ue(i-1))/(s(i) - s(i-1));
            else
                spd_grad = (ue(2) - ue(1))/(s(2) - s(1));
            end

            % Compute the value of lambda
            lambda = theta_sqrd*rho/mu*spd_grad;

            % Compute the values of H and l
            [H,~] = laminar_thwaites_approximations(lambda);

            % Compute the displacement thickness
            del_str(i) = H*theta(i);

            % Compute the friction coefficient
            Cf(i) = 0;
        end
        end
    % Go through the remaining turbulent points
    for i = i:num_points-1
        % Extract the previous poiints
        theta_0 = theta(i);
        H_0     = H;
        if ~isreal(H_0)
            error('Computation error');
        end

        % Compute the value of H1
        H1_0 = 3.0445 + 0.8702/(H_0-1.1)^1.2721;

        % Compute the ue*theta*H1 quantity
        UTH_0 = ue(i-1)*theta_0*H1_0;

        % Step the UTH value
        UTH_1 = step_UTH(UTH_0,s(i),s(i+1),ue(i-1),H1_0);

        % Compute the friction coefficient
        Cf(i) = 0.246/(10^(0.678*H_0)*Re_th^0.268)*(1+0.2*(ue(i-1)/a_inf)^2);

        % Compute the speed gradient
        if i ~= 1
            spd_grad = (ue(i) - ue(i-1))/(s(i) - s(i-1));
```

```
        else
            spd_grad = (ue(2) - ue(1))/(s(2) - s(1));
        end

        % Determine the next value of theta
        theta(i+1) = step_theta(theta_0,ue(i),spd_grad,H_0,Cf(i),s(i),s(i+1),ue(i-1)/a_inf);
        if ~isreal(theta(i+1))
            error('Computation error');
        elseif theta(i+1) < 0
            error('Computation error');
        end

        % Determine the new H1 value from theta, ue, and UTH
        H1_1 = UTH_1/(ue(i)*theta(i+1));

        % Determine the value of H
        H = H1_to_H(H1_1);

        % Compute the boundary layer displacement thickness
        del_str(i+1) = H*theta(i+1);
        if ~isreal(del_str(i+1))
            error('Computation error');
        end

        % Compute the local reynolds number
        Re_l = rho*ue(i)/mu;

        % Calculate the momentum reynolds number
        Re_th = Re_l*theta(i+1);
        if ~isreal(Re_th)
            error('Computation error');
        end
    end

     % Compute the friction coefficient for the last point
     Cf(end) = 0.246/(10^(0.678*H)*Re_th^0.268)*(1+0.2*(ue(i)/a_inf)^2);
end

% Discard the boundary layer thickness at the last 5% of chord
idx = find(s > 0.95*max(s));
del_str(idx) = del_str(idx(1)-1);
end

%% Laminar Functions
% This function computes the moment thickness in laminar flow over the
% airfoil surface.
function theta_sqrd = laminar_momentum_thickness(mu,ue,s,idx)
% First compute the integral
sum = 0;
for i = 2:idx
    sum = sum + (ue(i))^5*(s(i)-s(i-1));
end

% Compute the moment thickness
if idx ~= 1
    theta_sqrd = (0.441*mu./(ue(i))^6)*sum;
else
    theta_sqrd = 0;
end
end

% This function computes the values of l and H based on Thwaites
% approximation
function [H,l] = laminar_thwaites_approximations(lambda)
if lambda >= -0.1 && lambda < 0
    l = 0.22 + 1.402*lambda + (0.018*lambda)/(lambda + 0.107);
    H = 2.088 + 0.0731/(lambda + 0.14);
elseif lambda >= 0 && lambda < 0.1
    l = 0.22 + 1.57*lambda - 1.8*lambda^2;
    H = 2.61 - 3.75*lambda + 5.24*lambda^2;
elseif lambda < -0.1
```

```
    lambda = -0.1;
    l = 0.22 + 1.402*lambda + (0.018*lambda)/(lambda + 0.107);
    H = 2.088 + 0.0731/(lambda + 0.14);
elseif lambda >= 0.1
    lambda = 0.1;
    l = 0.22 + 1.57*lambda - 1.8*lambda^2;
    H = 2.61 - 3.75*lambda + 5.24*lambda^2;
end
end

%% Turbulent Functions
% Use Euler's Method to compute the next  step of UTH
function UTH = step_UTH(UTH_0,s1,s2,ue_0,H1_0)
UTH = 0.0306*(ue_0/(H1_0-3.0)^0.6169)*(s2-s1) + UTH_0;
end

% Use Euler's Method to compute the next step of theta
function theta = step_theta(theta_0,ue,spd_grad,H_0,Cf_0,s1,s2,M_inf)
m = Cf_0/2 - (2+H_0-M_inf^2)*theta_0/ue*spd_grad;
theta = m*(s2-s1) + theta_0;
if theta < 0
    theta = theta_0;
end
end

function H = H1_to_H(H1)
if H1 < 3.3
    H = 3.0;
elseif H1 >=3.3 && H1 < 5.3
    H = 0.6778 + 1.1536*(H1-3.3)^(-0.326);
else % if H >= 5.3
    H = 1.1 + 0.86*(H1-3.3)^(-0.777);
end
end
%% Transition Functions
% This function determines if the transition criterion has been met
function bool = check_if_transition(Re_l,s,Re_th)
% Compute the distance reynolds number
Re_s = Re_l*s;

% Compute the transition reynolds number
Re_transition = 1.174*(1 + 22400/Re_s)*Re_s^0.46;

% Return the boolean
bool = Re_th > Re_transition;
end
```

```
% vortex_line.m
%
% Description:
%   This function calculates the induced velocity at an arbitrary point by
%   the a vortex line.
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Eq. 10.116
%


function q = vortex_line(p, p1, p2,vorticity)
% Calculate the cross product between r1 and r2
cross12(1) =  (p(2) - p1(2))*(p(3) - p2(3)) - (p(3) - p1(3))*(p(2) - p2(2));
cross12(2) = -(p(1) - p1(1))*(p(3) - p2(3)) + (p(3) - p1(3))*(p(1) - p2(1));
cross12(3) =  (p(1) - p1(1))*(p(2) - p2(2)) - (p(2) - p1(2))*(p(1) - p2(1));

% Calculate the absolute of the above cross product
abs_cross12 = sum(cross12.^2);

% Calculate the distances between p and 1, then p and 2
dist_rp1 = norm(p-p1);
dist_rp2 = norm(p-p2);

% Check for singular conditions
if dist_rp1 == 0 || dist_rp2 == 0 || abs_cross12 == 0
    q = [0 0 0];
    return;
end

% Calculate the dot product quantities
dot_r01 = sum((p2-p1).*(p-p1));
dot_r02 = sum((p2-p1).*(p-p2));

% Calculate the velocity
q = vorticity/4/pi/abs_cross12*(dot_r01./dist_rp1 - dot_r02./dist_rp2)*cross12;
end
```

```
% vortex_ring.m
%
% Description:
%   This function calculates the induced velocity at an arbitrary point
%   caused by a vortex ring of four vortex lines
%
% Based on: Low Speed Aerodynamics by Katz and Plotkin Eq. 10.117
%

function [q, qt] = vortex_ring(p,p1,p2,p3,p4,vorticity)
% Calculate the components for the trailing vortex segments
qt = vortex_line(p, p2, p3,vorticity) + vortex_line(p, p4, p1,vorticity);

%Calculate q as the sum of all the induced velocities of all four lines
q = vortex_line(p, p1, p2,vorticity) + vortex_line(p, p3, p4,vorticity) + qt;
end
```

Appendix F: Structural Functions

```
% calc_angle_distribution.m
%
% Description:
%   This function is used to determine the deflection angles caused by the
%   bending moment along the wingspan.
%
% Written by:   Julian Bardin
% Date:         2021-03-27
%
% Developed using definitions from Mechanics of Materials by Hibbeler
%

function th = calc_angle_distribution(box_ref,panel_x,y_lift,M_lift,theta_p,E,twist)
% Determine how many points are being analysed
num_points = length(y_lift);

% Initialize the output
th = NaN(1,num_points);
th(1) = 0;

% Compute the spanwise chord distribution
chord_lens = NaN(1,num_points);
for i = 1:num_points
    chord_lens(i) = abs(panel_x(1,i) - panel_x(end,i));
end

% Go across the wing in the spanwise direction computing the value for the
% change in theta divided by distance
for i = 2:num_points
    % Determine the average chord length of the section
    c_mean = mean([chord_lens(i) chord_lens(i-1)]);

    % Resize the parameters for the wingbox for this size
    box = scale_wingbox(box_ref,c_mean);

    % Determine the maximum z-value of the wing box. The computation will
    % proceed using this value above the centroid
    z_max = max(box.z_vals) - box.z_centroid;

    % Convert this point into the principal reference frame
    [x_p,z_p] = rotate_wingbox_coords(z_max,theta_p+twist(i));

    % Compute the moment components along the principal axes
    Mx_p = M_lift(i)*cosd(-theta_p-twist(i));
    Mz_p = M_lift(i)*sind(-theta_p-twist(i));

    % Compute the strain at this point
    strain = calc_bending_normal_strain(Mx_p,Mz_p,x_p,z_p,E,box.Izz_p,box.Ixx_p);

    % Compute the theta derivative
    dth_dy = -strain/(z_max);

    % Compute the next step of theta
    th(i) = th(i-1) + dth_dy*(y_lift(i)-y_lift(i-1));
end

% Convert the angles into degrees
th = th*180/pi;
end

% This function converts a given point along the global
% z-axis centered on the centroid of the wingbox into a coordainte in the
% principal reference
function [x_p,z_p] = rotate_wingbox_coords(z,theta_p)
x_p = z*sind(theta_p);
z_p = z*cosd(theta_p);
end

% This function computes the normal strain at a point
```

```
function strain = calc_bending_normal_strain(Mx_p,Mz_p,x_p,z_p,E,Izz_p,Ixx_p)
strain = Mz_p*x_p/(E*Izz_p) - Mx_p*z_p/(E*Ixx_p);
end
```

```
% calc_bend_loading.m
%
% Description:
%   This function computes the spanwise shear distribution caused by the
%   lifting distribution.
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%

function [V,M,y] = calc_bend_loading(L,dL,panel_y)
% Determine how many points there are in the spanwise direction
dims = size(dL);
num_sections = dims(2);

% Determine the y-positions
y = panel_y(1,:);

% Initialize the output variable
V = NaN(1,num_sections+1);
M = NaN(1,num_sections+1);
V(1) = -L;
M(end) = 0;

% Determine the spanwise lift distribution
for i = 1:num_sections
    % Determine the shear force at the next point
    V(i+1) = V(i) + sum(dL(:,i));
end

% Determine the moment distribution
for i = num_sections:-1:1
    % Determine panel length
    len = (y(i+1) - y(i));

    % Determine the constant force distribution over the section
    w = sum(dL(:,i))/len;

    % Determine the moment at the point
    M(i) = M(i+1) + 0.5*w*len^2;
end

end
```

```
% calc_bending_deflection.m
%
% Description:
%   This function approximates the vertical deflection of the wing, centered
%   at its centroid,
%
% Written by:   Julian Bardin
% Date:         2021-03-27
%

function dz = calc_bending_deflection(y_lift,th)
% Initialize the output variable
dz = NaN(1,length(y_lift));
dz(1) = 0;

% Go through the points and compute the deflection
for i = 2:length(y_lift)
    dz(i) = dz(i-1) + tand(th(i))*(y_lift(i)-y_lift(i-1));
end
end
```

```
% calc_moments_of_inertia.m
%
% Description:
%   This function is used to compute the moments of inertia and product of
%   inertia for the wingbox based on the area concentration distribution
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%

function [Ixx,Izz,Izx] = calc_moments_of_inertia(box)
% Extract the relevant wingbox data
x_vals = [box.x_vals box.x_strs] - box.x_centroid;
z_vals = [box.z_vals box.z_strs] - box.z_centroid;
A_vals = [box.A_vals box.A_strs];

% Compute the moments of inertia
Ixx = sum(x_vals.*x_vals.*A_vals,'all');
Izz = sum(z_vals.*z_vals.*A_vals,'all');
Izx = sum(z_vals.*x_vals.*A_vals,'all');
end
```

```
% calc_principal_angle.m
%
% Description:
%   This function computes angle that the principal x-axis makes with
%   the global x-axis.
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%
% Based on: Mechanics of Materials by Hibbeler Page 795
%

function theta_p = calc_principal_angle(box)
% Compute the angle of the principal x-axis
theta_p = 0.5*atand(-2*box.Izx/(box.Ixx - box.Izz));
end
```

```
% calc_principal_inertia
%
% Description:
%   This function rotates the moments of inertia to the values
%   corresponding to the principal reference frame. It also sets the
%   product of inertia to zero, as occurs with the principal axis.
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%
% Based on: Mechanics of Materials by Hibbeler, Page 794
%

function [Ixx_p,Izz_p,Izx_p] = calc_principal_inertia(Ixx,Izz,Izx,theta_p)
% Compute the value of Ixx in the principal reference frame
Ixx_p = 0.5*(Ixx + Izz) + 0.5*(Ixx - Izz)*cosd(2*theta_p) - Izx*sind(2*theta_p);

% Compute the value of Izz in the principal reference frame
Izz_p = 0.5*(Ixx + Izz) + 0.5*(Ixx - Izz)*cosd(2*theta_p) + Izx*sind(2*theta_p);

% Set the product of inertia to zero (always true along principal axes
Izx_p = 0;
end
```

```
% calc_shear_center.m
%
% Description:
%   This function is used to compute the location of the shear center of
%   wing box section corresponding to a unit-length chord. This is done by
%   applying arbitrary unit shear loads to compute the shear flow and
%   afterwards determine the location of the shear center.
%
% Written by:   Julian Bardin
% Date:         2021-04-03
%

function [x_shearcen,z_shearcen] = calc_shear_center(box)
% Extract relevant information from the wingbox object
x_vals  = box.x_vals - box.x_centroid;
z_vals  = box.z_vals - box.z_centroid;
t_vals  = box.t_vals;
theta_p = box.theta_p;
Ixx_p   = box.Ixx_p;
Izz_p   = box.Izz_p;
A_mean  = box.A_mean;

% Convert the x and z coordinates into the principal reference frame
x_p = x_vals.*cosd(theta_p) + z_vals.*sind(theta_p);
z_p = z_vals.*cosd(theta_p) - x_vals.*sind(theta_p);

% Compute the basic shear flow for the Sz case
qb = compute_basic_shear(0,1,t_vals,x_p,z_p,Ixx_p,Izz_p);

% Compute the constant shear flow for the Sz case
qs0 = compute_const_shear(qb,t_vals,x_p,z_p);

% Compute the x-coordiante of the shear center in the principal axes
x_shearcen_p = -find_shear_center_coord(-1,x_p,z_p,qb,qs0,A_mean);

% Compute the basic shear flow for the Sx case
qb = compute_basic_shear(1,0,t_vals,x_p,z_p,Ixx_p,Izz_p);

% Compute the constant shear flow for the Sx case
qs0 = compute_const_shear(qb,t_vals,x_p,z_p);

% Compute the z-coordiante of the shear center in the principal axes
z_shearcen_p = -find_shear_center_coord(1,x_p,z_p,qb,qs0,A_mean);

% Convert the shear center coordinates to the centroidal reference frame
x_shearcen = x_shearcen_p*cosd(theta_p) - z_shearcen_p*sind(theta_p);
z_shearcen = z_shearcen_p*cosd(theta_p) + x_shearcen_p*sind(theta_p);

% Convert the shear center into the global reference frame
x_shearcen = x_shearcen + box.x_centroid;
z_shearcen = z_shearcen + box.z_centroid;
end

% This function determines the basic shear distribution across the wingbox
function qb = compute_basic_shear(Sx,Sz,t_vals,x_p,z_p,Ixx_p,Izz_p)
% Append the first point to the end of the matrix to form a complete loop
x_p = [x_p x_p(1)];
z_p = [z_p z_p(1)];

% Compute the section lengths for the wingbox discretization
area_sum_x = NaN(1,length(x_p)-1);
area_sum_z = NaN(1,length(x_p)-1);
area_sum_x(1) = 0;
area_sum_z(1) = 0;
for i = 2:length(x_p)-1
    % Determine the z and x distance travaled between this point and the
    % next one
    dz = z_p(i+1) - z_p(i);
    dx = x_p(i+1) - x_p(i);

    % Determine the distance between the two points
```

```
    ds = sqrt(dx^2 + dz^2);

    % Calculate the value of this iteration's area x and z area sums
    area_sum_x(i) = area_sum_x(i-1) + t_vals(i)*x_p(i)*ds;
    area_sum_z(i) = area_sum_z(i-1) + t_vals(i)*z_p(i)*ds;
end

% Compute the shear flow matrix
qb = -Sx/Izz_p.*area_sum_x - Sz/Ixx_p*area_sum_z;
end

% This function determines the cosntant reference shear flow
function qs0 = compute_const_shear(qb,t_vals,x_p,z_p)
% Append the first point to the end of the matrix to form a complete loop
x_p = [x_p x_p(1)];
z_p = [z_p z_p(1)];

% Go along the border the wingbox, completing a numerical line integral to
% find the numerator value and denominator value
numerator = 0;
denominator = 0;
for i = 1:length(x_p)-1
    % Determine the z and x distance travaled between this point and the
    % next one
    dz = z_p(i+1) - z_p(i);
    dx = x_p(i+1) - x_p(i);

    % Determine the distance between the two points
    ds = sqrt(dx^2 + dz^2);

    % Add the value to the numerator
    numerator = numerator + qb(i)/t_vals(i)*ds;

    % Add the denominator value
    denominator = denominator + ds/t_vals(i);
end

% Compute the constant shear flow
qs0 = -numerator/denominator;
end

% This function computes one of the coordinates of the shear center
function coord = find_shear_center_coord(S,x_p,z_p,qb,qs0,A_mean)
% Append the first point to the end of the matrix to form a complete loop
x_p = [x_p x_p(1)];
z_p = [z_p z_p(1)];

% Go through all the points completing a numerical line integral
torque_sum = 0;
for i = 1:length(x_p)-1
    % Determine the z and x distance travaled between this point and the
    % next one
    dz = z_p(i+1) - z_p(i);
    dx = x_p(i+1) - x_p(i);

    % Determine the distance between the two points
    ds = sqrt(dx^2 + dz^2);

    % Depending on if the section if vertical or not, the analysis is a bit
    % different
    if dx ~= 0
        % If dx is no zero, there is no worry of infinity appearing,
        % compute the slope normally
        m = dz/dx;

        % Compute the value of p
        p = abs((z_p(i) - m*x_p(i))/(1-m^2))*sqrt(m^2+1);

    else
        % If the slope is infinity, then p is parallel to the x-axis and
        % equal in length to the x-coordinate
```

```
        p = abs(x_p(i));
    end

    % Add the valye to the torque_sum
    torque_sum = torque_sum + p*qb(i)*ds;
end

% Solve for the shear center coordinate
coord = (torque_sum + 2*A_mean*qs0)/S;
end
```

```
% calc_torsion_loading.m
%
% Description:
%   This function is used to compute the torque distribution along the
%   wing. This is done by first computing the torque applied to each
%   section, then finding the reaction force at the root from this. The
%   spanwise distribution is found by adding the section torque at each
%   point to the previous torque, finding the net torque generated by lift
%   along the wing.
%
% Written by:   Julian Bardin
% Date:         2021-03-27
%

function [T_lift,y_lift,T_span] = calc_torsion_loading(dL,wing_calc,box_ref)
% Determine the y-positions
y_lift = wing_calc.panel_y(1,:);

% Determine how many points are being analysed
num_points = length(y_lift);

% Initialize the output
T_lift = NaN(1,num_points);

% Determine the chordlengths
% Compute the spanwise chord distribution
chord_lens = NaN(1,num_points+1);
for i = 1:num_points
    chord_lens(i) = abs(wing_calc.panel_x(1,i) - wing_calc.panel_x(end,i));
end

% Go through the wing slices and determine the torque being applied at each
T_span = NaN(1,num_points-1);
for i = 1:num_points-1
    % Determine the average chord length for the section
    chord = (chord_lens(i) + chord_lens(i+1))/2;

    % Determine the scaled centroid location
    x_pivot = box_ref.x_shearcen*chord;

    % Extract the lift distirbution
    lift_dist = dL(:,i);

    % Extract the colocation points where the lift force is applied
    coloc_x = wing_calc.coloc_x(:,i)-(wing_calc.panel_x(1,i) + wing_calc.panel_x(1,i+1))/2;

    % Compute the sectional torque
    T_span(i) = sum(lift_dist.*(x_pivot-coloc_x));
end

% Go along the wing and apply the torques as they are encountered
T_lift(1) = -sum(T_span);
for i = 1:num_points-1
    T_lift(i+1) = T_lift(i) + T_span(i);
end
end
```

```
% calc_twist_distribution.m
%
% Description:
%   This function computes the twist distribution in the spanwise direction
%   along the wing.
%
% Written by:   Julian Bardin
% Date:         2021-03-27
%

function twist = calc_twist_distribution(panel_x,box_ref,T_lift,y_lift,G)
% Determine how many points are being analysed
num_points = length(y_lift);

% Initialize the output
twist = NaN(1,num_points);
twist(1) = 0;

% Compute the spanwise chord distribution
chord_lens = NaN(1,num_points);
for i = 1:num_points
    chord_lens(i) = abs(panel_x(1,i) - panel_x(end,i));
end

% Go along the span and compute the wing twist
for i = 2:num_points
    % Compute the mean chord
    c_mean = (chord_lens(i) + chord_lens(i-1))/2;

    % Scale the wingbox
    box = scale_wingbox(box_ref,c_mean);

    % Compute the next twist value
    twist(i) = twist(i-1) - T_lift(i)*abs(y_lift(i)-y_lift(i-
1))/(4*(box.A_mean)^2*G)*sum(box.st_vals);
end

% Convert the twist into degrees
twist = twist*180/pi;
end
```

```
% calc_wingbox_centroid.m
%
% Description:
%   This function is used to compute the centroid of the wingbox based on
%   the area concentration distribution.
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%

function box = calc_wingbox_centroid(box)
% Extract the relevant wingbox data
x_vals = box.x_vals;
z_vals = box.z_vals;
A_vals = box.A_vals;

% Extract the stringer properties
x_strs = box.x_strs;
z_strs = box.z_strs;
A_strs = box.A_strs;

% Determine the location of the centroid
box.x_centroid = (sum(x_vals.*A_vals,'all') +
sum(x_strs.*A_strs,'all'))/(sum(A_vals,'all')+sum(A_strs,'all'));
box.z_centroid = (sum(z_vals.*A_vals,'all') +
sum(z_strs.*A_strs,'all'))/(sum(A_vals,'all')+sum(A_strs,'all'));
end
```

```
% determine_wingbox_data.m
%
% Description:
%   This function populates the basic information which defines the wingbox
%   arrangement. Mainly, this sets the thickness values and outline
%   coordinates for the skin, as well as the stringer information
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%

function box =
determine_wingbox_data(foil,box_chords,N_strs,t_skin,t_spar,A_stringers,res_skin,res_spar)
% Initialize the wingbox object
box = WingBox();

% Pass on relevant parameters into the wingbox object
box.t_skin = t_skin;
box.t_spar = t_spar;

% Find where the airfoil is split
idx_split = find(foil.x_coords == 0);

% Split up the surfaces
x_lo = foil.x_coords(1:idx_split);
z_lo = foil.z_coords(1:idx_split);
x_up = foil.x_coords(idx_split:end);
z_up = foil.z_coords(idx_split:end);

% Determine which vectors need to be flipped
if x_up(1) > x_up(end)
    x_up = fliplr(x_up);
    z_up = fliplr(z_up);
else
    x_lo = fliplr(x_lo);
    z_lo = fliplr(z_lo);
end

% Compute the coordinates for the wingbox outline
[box.x_vals,box.z_vals] = calc_wingbox_outline(x_up,x_lo,z_up,z_lo,res_skin,res_spar,box_chords);

% Compute the coordinates of the stringers
[box.x_strs,box.z_strs,box.A_strs] =
calc_stringers(x_up,x_lo,z_up,z_lo,N_strs,A_stringers,box_chords);

% Compute the area concentrations
[box.A_vals,box.st_vals,box.t_vals] =
calc_area_concentrations(box.x_vals,box.z_vals,t_skin,t_spar);
end

% This function computes the coordinates of the skin/spar outline of the
% wingbox
function [x_vals,z_vals] = calc_wingbox_outline(x_up,x_lo,z_up,z_lo,res_skin,res_spar,box_chords)
% Determine the x-coordinates for the airfoil skin surfaces
x_skins = linspace(box_chords(1),box_chords(2),res_skin);

% Compute the coordinates of the skin portion of the outline
z_skins_up = NaN(1,res_skin);
z_skins_lo = NaN(1,res_skin);
for i = 1:res_skin
    z_skins_up(i) = interp1(x_up,z_up,x_skins(i),'makima');
    z_skins_lo(i) = interp1(x_lo,z_lo,x_skins(i),'makima');
end

% Determine the coordinates of the forward spar (left)
z_left = linspace(z_skins_up(1),z_skins_lo(1),res_spar);
x_left = box_chords(1)*ones(1,res_spar);

% Determine the coordiantes for the rear spar (right)
z_right = linspace(z_skins_lo(end),z_skins_up(end),res_spar);
x_right = box_chords(2)*ones(1,res_spar);
```

```
% Assemble the coordinates so that the points start at the bottom left and
% wrap around the wingbox in counter clockwise direction
x_vals = [x_skins    x_right(2:end-1) fliplr(x_skins)    x_left(2:end-1)];
z_vals = [z_skins_lo z_right(2:end-1) fliplr(z_skins_up) z_left(2:end-1)];
end

% This function determines the locations of all the stringers. It also
% converts the given stringer area into a vector of the same length as the
% x and z coordinates, for easy indexing.
function [x_strs,z_strs,A_strs] =
calc_stringers(x_up,x_lo,z_up,z_lo,N_strs,A_stringers,box_chords)
% Determine the x-coordinates of the stringers
x_vals = linspace(box_chords(1),box_chords(2),N_strs+2);
x_vals = x_vals(2:end-1);

% Compute the coordinates of the skin portion of the outline
z_vals_up = NaN(1,N_strs);
z_vals_lo = NaN(1,N_strs);
for i = 1:N_strs
    z_vals_up(i) = interp1(x_up,z_up,x_vals(i),'makima');
    z_vals_lo(i) = interp1(x_lo,z_lo,x_vals(i),'makima');
end

% Assemble the coordinates to be output
x_strs = [x_vals x_vals];
z_strs = [z_vals_lo z_vals_up];
A_strs = A_stringers*ones(1,2*N_strs);
end

% This function computes the area concentrations of the wingbox, used for a
% numerical approximation of the area distribution of the geometry
function [A_vals,st_vals,t_vals] = calc_area_concentrations(x_vals,z_vals,t_skin,t_spar)
% Append the first point to the end of the matrix to form a complete loop
x_vals = [x_vals x_vals(1)];
z_vals = [z_vals z_vals(1)];

% Initialize the output variable
A_vals  = NaN(1,length(x_vals)-1);
st_vals = NaN(1,length(x_vals)-1);
t_vals  = NaN(1,length(x_vals)-1);
% Go through all the points and compute the area for each area
% concentration
for i = 1:length(x_vals)-1
    % Determine the z and x distance travaled between this point and the
    % next one
    dz = z_vals(i+1) - z_vals(i);
    dx = x_vals(i+1) - x_vals(i);

    % Determine the distance between the two points
    ds = sqrt(dx^2 + dz^2);

    % Determine which thickness is to be used. If the dx is 0, then it is a
    % spar, otherwise it is a skin. Compute the area as the
    % distance*thickeness
    if dx == 0
        A_vals(i)  = ds*t_spar;
        st_vals(i) = ds/t_spar;
        t_vals(i)  = t_spar;
    else
        A_vals(i)  = ds*t_skin;
        st_vals(i) = ds/t_skin;
        t_vals(i)  = t_skin;
    end
end
end
```

```
% scale_wingbox.m
%
% Description:
%   This function is used to scale the wingbox parameters from the section
%   corresponding to unit-length chord to the length of a given cross
%   section
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%

function box = scale_wingbox(box,chord)
% Scale the position of the centroid
box.x_centroid = box.x_centroid*chord;
box.z_centroid = box.z_centroid*chord;

% Scale the position of the shear centre
box.x_shearcen = box.x_shearcen*chord;
box.z_shearcen = box.z_shearcen*chord;

% Scale the wingbox outline
box.x_vals = box.x_vals*chord;
box.z_vals = box.z_vals*chord;
box.A_vals = box.A_vals*chord;
box.st_vals = box.st_vals*chord;

% Scale the mean area
box.A_mean = box.A_mean*chord^2;

% Scale the stringer positions
box.x_strs = box.x_strs*chord;
box.z_strs = box.z_strs*chord;

% Scale the moments of inertia
box.Ixx = box.Ixx*chord^3;
box.Izz = box.Izz*chord^3;
box.Izx = box.Izx*chord^3;
box.Ixx_p = box.Ixx_p*chord^3;
box.Izz_p = box.Izz_p*chord^3;
box.Izx_p = box.Izx_p*chord^3;
end
```

```
% struct_solver_bending.m
%
% Description:
%   This function encapsulates the functions used to compute the structural
%   bending of the wing. The important outputs include deflection angle and
%   vertical deflection distance.
%
% Written by:   Julian Bardin
% Date:         2021-03-27
%

function [dz,def_angle,V_lift,M_lift,y_lift] =
struct_solver_bending(wing_calc,box_ref,L,dL,E,twist)
% Compute the loading distribution
[V_lift,M_lift,y_lift] = calc_bend_loading(L,dL,wing_calc.panel_y);

% Compute the deflection angles across the wing span
def_angle =
calc_angle_distribution(box_ref,wing_calc.panel_x,y_lift,M_lift,box_ref.theta_p,E,twist);
dz = calc_bending_deflection(y_lift,def_angle);
end
```

```
% struct_solver_deformation.m
%
% Description:
%   This function encapsulates the structural solver of the wing and
%   wingbox. This first computes the torque loading, then the twist
%   distribution. The bending calculator is then run, utilizing the twist
%   as an input as well. This will rotate the point probed for bending
%   radius to compensate for the wing twist. The deflection and twist along
%   the spanwise direction are returned, as well as the loads for plotting.
%
% Written by:   Julian Bardin
% Date:         2021-03-27
%

function [y_lift,dz,twist,T_lift,V_lift,M_lift]...
    = struct_solver_deformation(wing_calc,box_ref,L,dL,E,G)
% Compute torsional loading distribution
[T_lift,y_lift,~] = calc_torsion_loading(dL,wing_calc,box_ref);

% Compute the twist angles
twist = calc_twist_distribution(wing_calc.panel_x,box_ref,T_lift,y_lift,G);

% Determine the bending deflection and loading, correcting for the twist
[dz,~,V_lift,M_lift,y_lift] = struct_solver_bending(wing_calc,box_ref,L,dL,E,twist);
end
```

```
% struct_solver_wingbox.m
%
% Description:
%   This function encapsultes the functions used to compute the
%   characteristics of a wingbox corresponding to an airfoil of unit
%   chord length.
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%

function box =
struct_solver_wingbox(foil,box_chords,stringers,t_skin,t_spar,A_stringers,res_skin,res_spar)
% Determine basic geometric and area data for the wing box
box =
determine_wingbox_data(foil,box_chords,stringers,t_skin,t_spar,A_stringers,res_skin,res_spar);

% Compute the centroid for the wingbox
box = calc_wingbox_centroid(box);

% Compute the moments of inertia in the global reference frame
[box.Ixx,box.Izz,box.Izx] = calc_moments_of_inertia(box);

% Determine the angle between the principal axes and the global axes
box.theta_p = calc_principal_angle(box);

% Determine the moments of inertia in the principal reference frame
[box.Ixx_p,box.Izz_p,box.Izx_p] = calc_principal_inertia(box.Ixx,box.Izz,box.Izx,box.theta_p);

% Determine the mean area of the wingbox
box.A_mean = polyarea([box.x_vals box.x_vals(1)],[box.z_vals box.z_vals(1)]);

% Compute the shear center of the wingbox
[box.x_shearcen,box.z_shearcen] = calc_shear_center(box);
end
```

Appendix G: Utility Functions

```
% Airfoil.m
%
% Description:
%   This class defines the airfoil object, used to store airfoil
%   information. This is analogous to the Wing_calc object, but for use
%   with 2D calculation
%
% Written by:   Julian Bardin
% Date:         2021-02-16
%


classdef Airfoil
    properties
        % Define the geometric properties which define the airfoil
        x_coords = []; % x-direction coordinates for airfoil geometry
        z_coords = []; % z-direction coordinates for airfoil geometry
        a_panels = []; % The angle of the panel relative to -ve freestream
        x_coloc  = []; % x-direction coordinates of the colocation points
        z_coloc  = []; % z-direction coordiantes of the colocation points

        % Define secondary parametere
        x_norm   = []; % x-direction component of each panel's normal
        z_norm   = []; % z-direction component of each panel's normal
        x_tang   = []; % x-direction component of each panel's tangent
        z_tang   = []; % z-direction component of each panel's tangent

    end

    methods
        % Constructor method
        function obj = Airfoil(coord_vectors)
            obj.x_coords = fliplr(coord_vectors(1,:));
            obj.z_coords = fliplr(coord_vectors(2,:));
        end
    end
end
```

```
% calc_airfoil_panel_angle.m
%
% Description:
%   This function computes the angle of each panel the comprises the
%   airfoil.
%   *** NOTE: ANGLES IN RADIANS, NOT DEGREES
%
% Written by:   Julian Bardin
% Date:         2021-02-16
%

function [alpha] = calc_airfoil_panel_angle(foil)
% Extract the airfoil geometry
x_foil = foil.x_coords;
z_foil = foil.z_coords;

% Initialize the output matrix
alpha   = NaN(1,length(x_foil)-1);

% Go through all the panels and compute the angle
for i = 1:length(x_foil)-1
    %  Define the points p1 and p2
    p1 = [x_foil(i)   z_foil(i)];
    p2 = [x_foil(i+1) z_foil(i+1)];

    % Define the vector pointing from p2 to p1, which lies on the x_p axis in
    % the negative direction
    r21 = p2-p1;

    % Calculate the angle of the panel. Note all angles within this function
    % are in radians, rather than degrees used in other sections of this
    % program.
    alpha(i)   = atan2(r21(2),r21(1));
end
end
```

```
% calc_curve_dist.m
%
% Description:
%   This fucntion computes the curve displacement from x(0), z(0) at each
%   point defined by the given x and z vectors.
%
% Written by:   Julian Bardin
% Date:         2021-03-19
%

function s = calc_curve_dist(x,z)
% Initialize the output matrix
s = NaN(1,length(x));

% Set the first value of s to 0
s(1) = 0;

% Go through all the coordinates and compute the curve distance
for i = 2:length(x)
    s(i) = sqrt((x(i)-x(i-1))^2 + (z(i) - z(i-1))^2) + s(i-1);
end
end
```

```
% calc_mean_camber.m
%
% Description:
%   This function is used to calcualte the mean camber line of an airfoil
%   at discrete given x-values
%
% Written by:   Julian Bardin
% Date:         2021-02-09
%

function [z_camber] = calc_mean_camber(airfoil,x_camber)
% Extract the coordinates of the airfoil
x = airfoil(1,:);
z = airfoil(2,:);

% Find the point that corresponds to the leading edge
idx_split = find(x == 0);

% Split the airfoil geometry into upper and lower surface using the
% idx_split point found above. It is assumed that the first geometry
% interval is the upper surface, but if it is reversed the generation of
% the camber line will remain unaffected
x_up = x(1:idx_split);
z_up = z(1:idx_split);
x_lo = x(idx_split:end);
z_lo = z(idx_split:end);

% Determine which vectors need to be flipped
if x_up(1) > x_up(end)
    x_up = fliplr(x_up);
    z_up = fliplr(z_up);
else
    x_lo = fliplr(x_lo);
    z_lo = fliplr(z_lo);
end

% Determine the length of the x_camber vector
x_len = length(x_camber);

% Initialize variables to store the interpolated points
z_up_interp = NaN(1,x_len);
z_lo_interp = NaN(1,x_len);

% Calculate the values of z for the camber
for i = 1:x_len
    z_up_interp(i) = interp1(x_up,z_up,x_camber(i),'makima');
    z_lo_interp(i) = interp1(x_lo,z_lo,x_camber(i),'makima');
end
z_camber = 0.5*(z_up_interp + z_lo_interp);

end
```

```
% calc_vel_vector.m
%
% Description:
% This function is used to convert the free-stream velocity and angle of
% attack to a velocity vector Q_inf which is equivalent to a wing at 0 AoA
%
% Written by:   Julian Bardin
% Date:         2021-02-13
%

function Q_inf = calc_vel_vector(V_inf,AoA)
Q_inf = [V_inf*cosd(AoA) 0 V_inf*sind(AoA)];
end
```

```
% compute_new_z_from_disp
%
% Description:
%   This functin is used to adjust the airfoil surface geometry to account
%   for the boundary layer displacement thickness.
%
%

function z_new = compute_new_z_from_disp(x,z,disp)
% Initialize the variables to store x and z coordiantes with the
% displacement thickness
x_d = NaN(1,length(x));
z_d = NaN(1,length(x));
x_d(1) = x(1);
z_d(1) = z(1);

% Go through all the points (starting from 2 as the displacement thickness
% at the first point is always zero), and compute the adjusted value for x
% and z based on the dispacement thickness
for i = 2:length(z)
    % Compute the angle of this panel in the global reference frame
    panel_alpha = atan2d(z(i)-z(i-1),x(i)-x(i-1));

    % If the angle is equal to zero, then just use the displacement
    % thickness
    if panel_alpha == 0
        z_d(i) = z(i) + disp(i);
        x_d(i) = x(i);
        continue;
    end
    % Otherwise use trigonometry to adjust the coordinates
    z_d(i) =  disp(i)*cosd(panel_alpha) + z(i);
    x_d(i) = -disp(i)*sind(panel_alpha) + x(i);
end

% Interpolate to find the values of z that match with input x coordinates
z_new = interp1(x_d,z_d,x,'linear','extrap');

end
```

```
% convert_wing_obj.m
%
% Description:
%   This function is used to convert the parametrized wing into a
%   calculation-compatible equivalent form.
%
% Written by:   Julian Bardin
% Date:         2021-02-14
%

function wing_calc = convert_wing_obj(wing_param,x_panels,y_panels)
% Generate a calc-compatible wing object
wing_calc = Wing_Calc();

% Calculate the discrete panels that are used for VLM computation
[wing_calc.panel_x,wing_calc.panel_y,wing_calc.panel_z] =
determine_panel_coordinates(wing_param,x_panels,y_panels);

% Calculate the areas of the panels
wing_calc.panel_A = determine_panel_area(wing_calc);

% Determine the grid of vertices used for computing the vortex rings
[wing_calc.ring_x,wing_calc.ring_y,wing_calc.ring_z] = determine_ring_coordinates(wing_calc);

% Determine the colocation points and the normal vectors placed on them.
[wing_calc.coloc_x,wing_calc.coloc_y,wing_calc.coloc_z] = determine_coloc_coordinates(wing_calc);
[wing_calc.norm_x,wing_calc.norm_y,wing_calc.norm_z] = determine_coloc_norms(wing_calc);

end
```

```
% find_stagnation_point.m
%
% Description:
%   This function determines the location of the stagnation point on the
%   airfoil. If the stagnation point does not like on a node, then an index
%   value of X.5 is place between the two points in the tangential speed
%   matrix where the velocity flips signs.
%
% Written by:   Julian Bardin
% Date:         2021-03-17
%

function idx = find_stagnation_point(Qt)
% Convert the Qt vector into a vector of signs
Qt_sign = sign(Qt);

% First try and find a point where the sign is zero
idx = find(Qt_sign == 0,1,'first');

% If the idx is empty, it means no point equals 0, therefore set the
% stagnation point index to a midpoint between the two sign-flip points.
% The function that is used for drag computation will then round this value
% depending on which side of the airfoil is being computed.
if isempty(idx)
    % Find the first sign flip. Then subtract 0.5 from the index so that
    % the value returned is halfway between the two points.
    if Qt_sign(1) == 1
        idx = find(Qt_sign == -1,1,'first') -0.5;
    else
        idx = find(Qt_sign == 1,1,'first') -0.5;
    end
end
end
```

```
% gen_outline_2D.m
%
% Description:
%   This function returns two row vectors with coordinates for the wing
%   planform outline
%
% Written by:   Julian Bardin
% Date:         2021-02-09
%

function [x,y] = gen_outline_2D(wing_param)
% Define the origin at the leading edge at the root
x = 0;
y = 0;

% Define the leading edge at the kink
x(end+1) = x(1) + wing_param.semi_span*wing_param.dr_kink;
y(end+1) = y(1);

% Add point for the leading edge at the tip chord
x(end+1) = x(1) + wing_param.semi_span;
y(end+1) = y(1) - wing_param.semi_span*tand(wing_param.sweep_LE);

% Add point for the trailing edge the tip chord
x(end+1) = x(end);
y(end+1) = y(end) - wing_param.chord_root*wing_param.tr_tip;

% Add point for the trailing edge at the kink
x(end+1) = x(1) + wing_param.semi_span*wing_param.dr_kink;
y(end+1) = y(1) - wing_param.chord_root;

% Add point for the trailing edge at the root chord
x(end+1) = x(1);
y(end+1) = y(1) - wing_param.chord_root;

% Complete the loop and add the origin at the end of the row vectors
x(end+1) = x(1);
y(end+1) = y(1);

end
```

```
% gen_planform_LE_TE.m
%
% Description:
%   This function parses through the wing planform outline and isolates
%   which points belong to the leadinge edge, and which points belong to
%   the trailing edge.
%
% Written by:   Julian Bardin
% Date:         2021-02-09
%

function [x_LE,y_LE,x_TE,y_TE] = gen_planform_LE_TE(wing_param)
% First get the overall planform outline
[x,y] = gen_outline_2D(wing_param);

% Determine which points correspond to the tip chord.
tip_points = find(x == wing_param.semi_span);

% Coordinates from the first entry up until the first tip chord coordinate
% are taken as belonging to the leading edge
x_LE = x(1:tip_points(1));
y_LE = y(1:tip_points(1));

% Coordinates belonding to the trailing edge are taken as the range of
% values from teh last tip-chord points until the second-last outline points
x_TE = x(tip_points(end):end-1);
y_TE = y(tip_points(end):end-1);
end
```

```
% ij2K.m
%
% Description:
%   This function converts an i,j combination into its corresponding K
%   identifier.
%
% Written by:   Julian Bardin
% Date:         2021-02-13
%

function K = ij2K(i,j,y_panels)
K = (i-1)*y_panels + j;
end
```

```
% K2ij.m
%
% Description:
%   This function converts a given K identifier into its corresponding i
%   and j indices.
%
% Written by:   Julian Bardin
% Date:         2021-02-13
%

function [i,j] = K2ij(K,y_panels)
j = mod(K-1,y_panels)+1 ;
i = ceil(K/y_panels);
end
```

```
% read_selig_foil
%
% Description:
%   This function is responsible for loading the selig airfoil data file.
%   The output is a matrix, where the first row is a vector of x values,
%   and the second row is a matrix of z values.
%
% Written by:   Julian Bardin
% Date:         2021-02-09
%

function [points] = read_selig_foil(filename)
% Open the file
fileID = fopen(filename);

% Load the contents of the file into memory
contents = [];
while ~feof(fileID)
    contents = [contents ; {fgetl(fileID)}];
end

% Go through the loaded data and convert the strings into numbers
points = [];
for i = 1:length(contents)
    % Convert the string values to doubles
    vals = str2num(contents{i});

    % If the length is less than 2, assume this is a comment
    if length(vals) < 2
        continue
    end

    % Otherwise append the numbers to the points matrix
    points = [points ; vals];
end

% Transpose the points matrix to provice two row vectors.
points = points';

end
```

```
% render_airfoil.m
%
% Description:
%   This function generates a figure showing the airfoil with its mean
%   camber line.
%
% Written by:   Julian Bardin
% Date:         2021-02-09
%

function render_airfoil(foil)
% Create the figure
figure();
axis equal;
hold on;

% Plot the airfoil outline
plot(foil.x_coords,foil.z_coords,'-ok');

% Get the coordinates of the mean camber line
x_camber = linspace(0,1,100);
[z_camber] = calc_mean_camber([foil.x_coords ; foil.z_coords],x_camber);

% Plot the camber
plot(x_camber,z_camber,'-r');

% If the colocation points exist, plot them
if ~isempty(foil.x_coloc) && ~isempty(foil.z_coloc)
    plot(foil.x_coloc,foil.z_coloc,'kx');
end

% If the normals exist, plot them
if ~isempty(foil.x_norm) && ~isempty(foil.z_norm) && ~isempty(foil.x_coloc) &&
~isempty(foil.z_coloc)
    for i = 1:length(foil.x_norm)
        plot([foil.x_coloc(i) foil.x_coloc(i)+0.01*foil.x_norm(i)],[foil.z_coloc(i)
foil.z_coloc(i)+0.01*foil.z_norm(i)],'-k','LineWidth',1.3);
    end
end

% Output a legend
h = 0;
h(1) = plot(NaN,NaN,'kx');
h(2) = plot(NaN,NaN,'-ok');
h(3) = plot(NaN,NaN,'-r');
legend(h,'Colocation Points','Panel Nodes','Mean Camber Line');

end
```

```
% render_airfoil2.m
%
%

function render_airfoil2(foil,z_BL)
% Create the figure
figure();
grid on;
axis equal;
hold on;

% Plot the airfoil outline
plot(foil.x_coords,foil.z_coords,'-k');

% Get the coordinates of the mean camber line
x_camber = linspace(0,1,100);
[z_camber] = calc_mean_camber([foil.x_coords ; foil.z_coords],x_camber);

% Plot the camber
plot(x_camber,z_camber,'-r');

% Plot the boundary layer
plot(foil.x_coords,z_BL,'--r');

% Output a legend
h = 0;
h(1) = plot(NaN,NaN,'-k');
h(2) = plot(NaN,NaN,'--r');
legend(h,'Airfoil Outline','Boundary Layer Displacement');
end
```

```
% render_airfoil_CP.m
%
%

function render_airfoil_CP(foil,CP)

figure();
hold on;
axis equal
grid minor;

% CP(abs(CP) > 1E2) = 0;

plot(foil.x_coloc,CP);

set(gca, 'YDir','reverse')

plot(foil.x_coords,-foil.z_coords+1.5)


end
```

```
% render_loading.m
%
% Description:
%   This function renders the shear and moment diagrams for the spanwise
%   loading distribution.
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%

function render_loading(y,V,M,T)
% Create the figure
figure();
grid on;
hold on;

% Set the subplot
subplot(3,1,1);

% Plot axis lines
xline(0);
yline(0);

% Plot the shear distribution
plot(y,V,'-b');

% Add labelling
xlabel('Spanwise distance (m)');
ylabel('Shear Force (N)');
title('Shear Distribution');

% Set the subplot
subplot(3,1,2);

% Plot the moment distribution
plot(y,M,'-r');

% Set the labelling
ylabel('Bending Moment (Nm)');
xlabel('Spanwise distance (m)');
title('Moment Distribution');

% Set the subplot
subplot(3,1,3)

% Plot the torsion diagram
plot(y,T,'-r')

% Set the labelling
ylabel('Torsion (Nm)');
xlabel('Spanwise distance (m)');
title('Torsion Distribution');



end
```

```
% render_structure_response
%
% Description:
%   This function is used to generate plots for the twist and deflection
%   distribution along the wing.
%
% Written by:   Julian Bardin
% Date:         2021-03-27
%

function render_structure_response(y_lift,dz,twist)
% Create the figure
figure();
hold on;

% Set the subplot
subplot(2,1,1);

% Plot the data
plot(y_lift,dz,'-k');

% Label the plot
title('Spanwise Bending Deflection');
xlabel('Spanwise Distance (m)');
ylabel('Deflection (m)');

% Set the subplot
subplot(2,1,2);

% Plot the data
plot(y_lift,twist,'-k');

% Label the plot
title('Spanwise Twist');
xlabel('Spanwise Distance (m)');
ylabel('Twist (^o)');

end
```

```
% render_wing_2D.m
%
% Description:
%   This function is reponsible for generating a 2D planform view of the
%   wing.
%
% Written by:   Julian Bardin
% Date:         2021-02-09
%

function render_wing_2D(wing_param, wing_calc)
% Open the figure
figure();
grid on;
axis equal;
hold on;

% Draw a line for x = 0
xline(0);

% Generate the outline coordinates
[x,y] = gen_outline_2D(wing_param);
[x_LE,y_LE,x_TE,y_TE] = gen_planform_LE_TE(wing_param);

% Set the axes (x and y based on MATLAB, not on aircraft standard
% nomenclature)
x_range = wing_param.semi_span;
y_range = max(y)-min(y);
xlim([min(x)-0.1*x_range max(x)+0.1*x_range]);
ylim([min(y)-0.1*y_range max(y)+0.1*y_range]);

% Output the outline of the wing planform
plot(x,y,'-k','LineWidth',2);

% Render the panels if data for them exists
if ~isempty(wing_calc.panel_x) && ~isempty(wing_calc.panel_y)
    % Extract the panel locations
    x_panels = -wing_calc.panel_x;
    y_panels = wing_calc.panel_y;

    % Plot the panel borders
    plot(y_panels,x_panels,'-b');
    plot(y_panels',x_panels','-b');

end

% Output lines on the leading and trailing edge
plot(x_LE,y_LE,'-g','LineWidth',1.2);
plot(x_TE,y_TE,'-g','LineWidth',1.2);

% Render the vortex rings if they exist
if ~isempty(wing_calc.ring_x) && ~isempty(wing_calc.ring_y)
    % Extract the ring locations
    x_rings = -wing_calc.ring_x;
    y_rings = wing_calc.ring_y;

    % Plot the vortex right boundaries
    plot(y_rings,x_rings,'-r','LineWidth',1.5);
    plot(y_rings',x_rings','-r','LineWidth',1.5);
end

% Render the colocation points
if ~isempty(wing_calc.coloc_x) && ~isempty(wing_calc.coloc_y)
    x_coloc = -wing_calc.coloc_x;
    y_coloc = wing_calc.coloc_y;


    dims = size(wing_calc.coloc_x);
    for j = 1:dims(2)
        for i = 1:dims(1)
            plot(y_coloc(i,j),x_coloc(i,j),'kx');
```

```
        end
    end
end

% Output a legend
h = zeros(1,1);
h(1) = plot(NaN,NaN,'-g');
h(2) = plot(NaN,NaN,'-b');
h(3) = plot(NaN,NaN,'-r');
h(4) = plot(NaN,NaN,'xk');
legend(h,'LE and TE','Panels','Vortex Rings','Collocation Points');
end
```

```
% render_wing_3D.m
%
%

function render_wing_3D(wing_param, wing_calc,render_norms)
% Create the figure
figure();

% First render the root airfoil
x_f1 = -wing_param.airfoil(1,:)*wing_param.chord_root;
z_f1 = wing_param.airfoil(2,:)*wing_param.chord_root;
y_f1 = zeros(1,length(x_f1));
plot3(x_f1,-y_f1,z_f1,'-k','LineWidth',2);

% Now that the plot is 3D, set hold and axis
hold on;
axis equal;

% Generate the outline
[y_ol,x_ol] = gen_outline_2D(wing_param);
z_ol = zeros(1,length(x_ol));
% plot3(x_ol,-y_ol,z_ol,'-k','LineWidth',2);

% Get the lines for the leading and trailing edge
[y_LE,x_LE,y_TE,x_TE] = gen_planform_LE_TE(wing_param);
z_LE = zeros(1,length(x_LE));
z_TE = zeros(1,length(x_TE));
plot3(x_LE,-y_LE,z_LE,'-g','LineWidth',1.5);
plot3(x_TE,-y_TE,z_TE,'-g','LineWidth',1.5);

% Render the tip airfoil
x_f2 = x_LE(end)-wing_param.airfoil(1,:)*wing_param.chord_root*wing_param.tr_tip;
z_f2 = wing_param.airfoil(2,:)*wing_param.chord_root*wing_param.tr_tip;
y_f2 = ones(1,length(x_f2))*wing_param.semi_span;
plot3(x_f2,-y_f2,z_f2,'-k','LineWidth',2);

% Set axes limits
x_range = abs(max(x_ol) - min(x_ol));
y_range = abs(max(y_ol) - min(y_ol));
z_range = abs(max(z_f1) - min(z_f1));
xlim([min(x_ol)-0.2*x_range max(x_ol)+0.2*x_range]);
ylim([min(-y_ol)-0.2*y_range max(-y_ol)+0.2*y_range]);
zlim([min(z_ol)-2*z_range max(z_ol)+2*z_range]);


% If the panel data exists, plot it
if ~isempty(wing_calc.panel_x) && ~isempty(wing_calc.panel_y) && ~isempty(wing_calc.panel_z)
    plot3(-wing_calc.panel_x,-wing_calc.panel_y,wing_calc.panel_z,'-b');
    plot3(-wing_calc.panel_x',-wing_calc.panel_y',wing_calc.panel_z','-b');
end

% If the vortex ring data exits, plot it
if ~isempty(wing_calc.ring_x) && ~isempty(wing_calc.ring_y) && ~isempty(wing_calc.ring_z)
    % Extract the ring_x information
    ring_x = -wing_calc.ring_x;

    % Remove points at -1E6, and replace it with closer points
    ring_x(ring_x == -1E6) = min(x_ol)-0.2*x_range;

    % Plot the vortex rings
    plot3(ring_x,-wing_calc.ring_y,wing_calc.ring_z,'-r');
    plot3(ring_x',-wing_calc.ring_y',wing_calc.ring_z','-r');
end

% If the colocation point data exists, plot it
if ~isempty(wing_calc.coloc_x) && ~isempty(wing_calc.coloc_y) && ~isempty(wing_calc.coloc_z)
    x_coloc = wing_calc.coloc_x;
    y_coloc = wing_calc.coloc_y;
    z_coloc = wing_calc.coloc_z;
```

```
    dims = size(wing_calc.coloc_x);
    for j = 1:dims(2)
        for i = 1:dims(1)
            plot3(-x_coloc(i,j),-y_coloc(i,j),z_coloc(i,j),'kx');
        end
    end
end

% If the normals are calculated, plot them
if ~isempty(wing_calc.coloc_x) && ~isempty(wing_calc.coloc_y) && ~isempty(wing_calc.coloc_z)
&&...
        ~isempty(wing_calc.norm_x) && ~isempty(wing_calc.norm_y) && ~isempty(wing_calc.norm_z) &&
render_norms

    % Determine how many panels are present in the wing_calc object
    dims = size(wing_calc.panel_x);
    x_panels = dims(1)-1;
    y_panels = dims(2)-1;

    % Go through each panel and plot the normals
    for j = 1:y_panels
        for i = 1:x_panels
            % Determine the vector for each normal
            x_n = [wing_calc.coloc_x(i,j)
wing_calc.coloc_x(i,j)+wing_calc.norm_x(i,j)*0.05*wing_param.chord_root];
            y_n = [wing_calc.coloc_y(i,j)
wing_calc.coloc_y(i,j)+wing_calc.norm_y(i,j)*0.05*wing_param.chord_root];
            z_n = [wing_calc.coloc_z(i,j)
wing_calc.coloc_z(i,j)+wing_calc.norm_z(i,j)*0.05*wing_param.chord_root];

            % Plot the normals
            plot3(-x_n,-y_n,z_n,'-k');
        end
    end
end


% Output a legend
h = zeros(1,1);
h(1) = plot(NaN,NaN,'-g');
h(2) = plot(NaN,NaN,'-b');
h(3) = plot(NaN,NaN,'-r');
h(4) = plot(NaN,NaN,'xk');
legend(h,'LE and TE','Panels','Vortex Rings','Collocation Points','Location','best');

end
```

```
% render_wing_3D_lift.m
%
% Description:
%   This function renders the pressure distribution over the wing camber
%   determined from the wing's lift distribution.
%
% Written by:   Julian Bardin
% Date:         2021-02-13
%

function render_wing_3D_lift(wing_param, wing_calc,dL)

% Create the figure
figure();

% First render the root airfoil
x_f1 = -wing_param.airfoil(1,:)*wing_param.chord_root;
z_f1 = wing_param.airfoil(2,:)*wing_param.chord_root;
y_f1 = zeros(1,length(x_f1));
plot3(x_f1,-y_f1,z_f1,'-k','LineWidth',2);

% Now that the plot is 3D, set hold and axis
hold on;
axis equal;

% Generate the outline
[y_ol,x_ol] = gen_outline_2D(wing_param);
z_ol = zeros(1,length(x_ol));
plot3(x_ol,-y_ol,z_ol,'-k','LineWidth',2);

% Get the lines for the leading and trailing edge
[y_LE,x_LE,y_TE,x_TE] = gen_planform_LE_TE(wing_param);
z_LE = zeros(1,length(x_LE));
z_TE = zeros(1,length(x_TE));

% Render the tip airfoil
x_f2 = x_LE(end)-wing_param.airfoil(1,:)*wing_param.chord_root*wing_param.tr_tip;
z_f2 = wing_param.airfoil(2,:)*wing_param.chord_root*wing_param.tr_tip;
y_f2 = ones(1,length(x_f2))*wing_param.semi_span;
plot3(x_f2,-y_f2,z_f2,'-k','LineWidth',2);



x = wing_calc.panel_x;
y = -wing_calc.panel_y;
z = wing_calc.panel_z;

% dL(1,:) = dL(2,:);
c = dL./wing_calc.panel_A;

surf(-x,y,z,c,'EdgeColor','none')
colorbar


end
```

```
% render_wingbox.m
%
% Description:
%   This function is used to render a 2D cross section of the structural
%   wingbox.
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%

function render_wingbox(box)
% Create the figure
figure();
hold on;
axis equal;

% Set the plot limits
xlim([min(box.x_vals)-0.05,max(box.x_vals)+0.05]);

% Plot the wingbox outline and stringers
plot(box.x_vals,box.z_vals,'-k','LineWidth',1.5)
% plot([box.x_vals box.x_vals(1)],[box.z_vals box.z_vals(1)],'-k','LineWidth',1.5)
plot(box.x_strs,box.z_strs,'or')

% Plot the area concentrations
A_max = max(box.A_vals);
for i = 1:length(box.x_vals)
    plot(box.x_vals(i),box.z_vals(i),'.k','MarkerSize',box.A_vals(i)/A_max*20,'Color',[0.3 0.3
0.3]);
end

% Plot the centroid if the point exists. Additionally, plot some lines to
% indicate the unrotated non-principal axes
if ~isempty(box.x_centroid) && ~isempty(box.z_centroid)
    plot(box.x_centroid,box.z_centroid,'.k','MarkerSize',12);
    plot([box.x_centroid box.x_centroid+0.1],[box.z_centroid box.z_centroid],'--k');
    plot([box.x_centroid box.x_centroid],[box.z_centroid box.z_centroid+0.1],'--k');

end

% If the angle of the principal angle exists, plot axes to show the
% principal axes
len_p = 0.2;
if ~isempty(box.theta_p)
    [x,z] = principal_axes(box.x_centroid,box.z_centroid,box.theta_p,len_p);
    plot(x,z,'-k');
    [x,z] = principal_axes(box.x_centroid,box.z_centroid,box.theta_p+90,len_p);
    plot(x,z,'-k');
end

% If the shear center exists, plot it
if ~isempty(box.x_shearcen) && ~isempty(box.z_shearcen)
    plot(box.x_shearcen,box.z_shearcen,'xr');
end

% Output a legend
h = zeros(1,1);
h(1) = plot(NaN,NaN,'or');
h(2) = plot(NaN,NaN,'.k','Color',[0.3 0.3 0.3]);
h(3) = plot(NaN,NaN,'--k');
h(4) = plot(NaN,NaN,'-k');
h(5) = plot(NaN,NaN,'xr');
legend(h,'Stringers','Area Concentrations','Centroid Axis','Principal Axis','Shear Center');
end

% This function computes the points to plot the principal axes
function [x,z] = principal_axes(x_centroid,z_centroid,theta,len)
% Initialize the output variables
x = NaN(1,2);
z = NaN(1,2);
```

```
% Set he first point to be at the centroid
x(1) = x_centroid;
z(1) = z_centroid;

% Compute the following point
m = tand(theta);
if theta < -90 || theta > 90
    x(2) = x(1) - len/sqrt(m^2 + 1);
else
    x(2) = x(1) + len/sqrt(m^2 + 1);
end
z(2) = m*(x(2)-x(1)) + z(1);
end
```

```
% split_airfoil.m
%
% Description:
%   This function splits up the airfoil surface into the upper and lower
%   surface, based on the stagnation point index. This is pre-requisite for
%   computation of 2D viscous drag
%
% Written by:   Julian Bardin
% Date:         2021-03-18
%

function [x_1,z_1,Qt_1,x_2,z_2,Qt_2] = split_airfoil(foil,Qt,idx)
% Extract the airfoil geometry
x = foil.x_coords;
z = foil.z_coords;

% Extract the data for the upper surface
x_1  = fliplr(x(1:floor(idx)));
z_1  = fliplr(z(1:floor(idx)));
Qt_1 = fliplr(-Qt(1:floor(idx)));

% Extract the data for the lower surface
x_2  = x(ceil(idx):end);
z_2  = z(ceil(idx):end);
Qt_2 = Qt(ceil(idx):end);
end
```

```
% wing_bend_and_twist.m
%
% Description:
%   This function is used to deform the VLM mesh in response to the twist
%   (torsion) and z-deflection (bending) of the wing. First, each section
%   is rotated about the shear center by the supplied twist angle.
%   Afterwards, all points are deflected.
%
% Written by:   Julian Bardin
% Date:         2021-04-03
%

function wing_new = wing_bend_and_twist(wing_old,box_ref,twist,dz)
% Extract the panel coordiantes of the old wing planform
panel_x = wing_old.panel_x;
panel_y = wing_old.panel_y;
panel_z = wing_old.panel_z;
y_vals  = panel_y(1,:);

% Initialize the new wing object
wing_new = Wing_Calc();

% Set the y-values to be the same as before
wing_new.panel_y = wing_old.panel_y;

% Go through each y-slice and adjust the x- and z-coordinates based on the
% input distance dz values
for i = 1:length(twist)
    % Extract the coordinates of the slice
    x_vals = panel_x(:,i);
    z_vals = panel_z(:,i);

    % Determine the chord length
    chord = x_vals(end) - x_vals(1);

    % Resize the coordinates of the shear center centroid (pivot point of
    % rotation)
    x_pivot = box_ref.x_shearcen*chord;
    z_pivot = box_ref.z_shearcen*chord;

    % Extract the leading edge coordinates
    x_LE = x_vals(1);
    z_LE = z_vals(1);

    % Convert the coordinates from the global reference frame to the
    % centroidal-centered reference frame
    x_vals = x_vals - x_pivot - x_LE;
    z_vals = z_vals - z_pivot - z_LE;

    % Go through each point, and rotate it about the centroid
    for j = 1:length(x_vals)
        % Compute the angle that this point makes with the x-axis
        ref_angle = atan2d(z_vals(j),x_vals(j));

        % Compute the distance between this point and the origin
        r = sqrt(z_vals(j)^2 + x_vals(j)^2);

        % Compute the new coordinates
        x_vals(j) = r*cosd(ref_angle + twist(i));
        z_vals(j) = r*sind(ref_angle + twist(i));
    end

    % Convert the coordinates back to the global reference frame
    x_vals = x_vals + x_pivot + x_LE;
    z_vals = z_vals + z_pivot + z_LE;

    % Add the z-direction deflection caused by the bending
    z_vals = z_vals + dz(i);

    % Set the values in the output object
    wing_new.panel_x(:,i) = x_vals;
```

```
    wing_new.panel_z(:,i) = z_vals;
end

% Calculate the areas of the panels
wing_new.panel_A = determine_panel_area(wing_new);

% Determine the grid of vertices used for computing the vortex rings
[wing_new.ring_x,wing_new.ring_y,wing_new.ring_z] = determine_ring_coordinates(wing_new);

% Determine the colocation points and the normal vectors placed on them.
[wing_new.coloc_x,wing_new.coloc_y,wing_new.coloc_z] = determine_coloc_coordinates(wing_new);
[wing_new.norm_x, wing_new.norm_y, wing_new.norm_z]  = determine_coloc_norms(wing_new);
end
```

```
% Wing_Calc.m
%
% Description:
%   This class defines the wing in a form more convenient for aerodynamic
%   calculation. It is expected that objects of this type are populated by
%   converting from a Wing_Param object.
%
% Written by:   Julian Bardin
% Date:         2021-02-06
%

classdef Wing_Calc
    properties
        % Define the quantities used to calcualte the 3D viscous effects
        foil_geom   = {}; % Airfoil geometry at each foil_y point. This is in true coordinates
        foil_y      = []; % Distance from root for each geometry/AOA point
        foil_AoA    = []; % Angle of attack for each airfoil at each y
        foil_points = 0;  % The number of data points used to represent the wing

        % Define the quantities used for VLM to predict lift and induced
        % drag
        panel_x     = []; % x-coordinates for the panel vertices
        panel_y     = []; % y-coordinates for the panel vertices
        panel_z     = []; % z-coordinates for the panel vertices
        panel_A     = []; % Area of each panel
        ring_x      = []; % x-coordinates for the vortex ring vortices
        ring_y      = []; % y-coordinates for the vortex ring vortices
        ring_z      = []; % z-coordinates for the vortex ring vortices
        coloc_x     = []; % x-coordinates for the colocation points
        coloc_y     = []; % y-coordinates for the colocation points
        coloc_z     = []; % z-coordinates for the colocation points
        norm_x      = []; % x-distance for the panel normals
        norm_y      = []; % y-distance for the panel normals
        norm_z      = []; % z-distance for the panel normals

    end
    methods
    end
end
```

```
% Wing_Param.m
%
% Description:
%   This class defines a wing in its parametrized form. It also includes
%   derivative quantities which are calcualted from the main defining
%   parameters.
%
% Written By:   Julian Bardin
% Date:         2021-02-06
%

classdef Wing_Param
    properties
        % Properties which are parametrized and therefore define the
        % geometry of the wing
        airfoil     = []; % Selig format. X on first row, Z on second
        aoa         = 0;  % Angle of attack for the airfoil.
        chord_root  = []; % Length of the root chord in m
        tr_tip      = 1;  % Taper Ratio between tip and root chord
        semi_span   = []; % Length of the wing from the root to the tip in m
        sweep_LE    = 0;  % The sweep angle at the leading edge, in deg
        dr_kink     = 0;  % Location of the kink as a proportion of semispan

        % Properties calculated from the above definition
        qchord_line = []; % Coordinates of the quarter chord. X is 1st row, Y is 2nd
        sweep_QC    = []; % The sweep angle at the quarter chord
        area        = []; % Wing planform area
        tr_kink     = 1;  % Taper Ratio between kink and root chord

    end
    methods
        % This function calculates the secondary properties of the wing,
        % which are derived from teh parametrizing quantities
        function obj = calc_secondary_quantities(obj)

        end

    end
end
```

```
% WingBox.m
%
% Description:
%   This function is used to define an object for storing wing box data.
%
% Written by:   Julian Bardin
% Date:         2021-03-26
%

classdef WingBox
    properties
        % Define thicknesses
        t_spar = [];     % Thickness of the forward and rear spars in m
        t_skin = [];     % Thickness of the upper an lower skin

        % Define the centroid location relative to the airfoil
        x_centroid = []; % x location of the wingbox centroid with the airfoil coordinate system
        z_centroid = []; % z location of the wingbox centroid with the airfoil coordinate system

        % Define the shear center of the wingbox relative to the airfoil
        x_shearcen = []; % x location of the wingbox shear center
        z_shearcen = []; % z location of the wingbox shear center

        % Define the mean area of the wingbox
        A_mean = [];     % Mean area encased by the wing box cross section

        % Define the coordinate system of the wingbox
        x_vals = [];     % x values for the wing box outline
        z_vals = [];     % y values for the wing box outline
        A_vals = [];     % Area value for each area concentration set at each x,z point
        st_vals = [];    % Sum of the ratios of displacement/thickness
        t_vals = [];     % Thickness for each section of the wingbox outline

        % Define the stringer locations and area
        x_strs = [];     % x coordinates for the stringers
        z_strs = [];     % y coordinates for hte stringers
        A_strs = [];     % Area of each stringer

        % Define the moments of inertia
        Ixx    = [];     % Moment of Inertia along the x-axis
        Izz    = [];     % Moment of Inertia along the z-axis
        Izx    = [];     % Product of Inertia

        % Define the angle of the principal axis relative to the global
        % aircraft reference frame
        theta_p = [];    % Angle in degrees

        % Define the moments of inertia in the principal reference frame
        Ixx_p   = [];    % Moment of inertia along principal x-axis
        Izz_p   = [];    % Moment of inertia along principal z-axis
        Izx_p   = [];    % Product of inertia in principal reference frame
    end

    methods

    end
end
```